Semi-Symmetric Graphs of Valence 5

Berkeley Churchill

March 2012

Abstract

A graph is semi-symmetric if it is regular and edge transitive but not vertex transitive. The 3- and 4-valent semi-symmetric graphs are well-studied. Several papers describe infinite families of such graphs and their properties. 3-valent semi-symmetric graphs have been completely classified up to 768 vertices. The goal of this project is to extend this work to 5-valent semi-symmetric graphs. In this paper I present work on searching for these graphs, and the construction of such a graph.

Contents

1 Introduction

The study of semi-symmetric graphs started with Folkman's 1967 paper [Fol67] where he proved the smallest semi-symmetric graph had 20 vertices

and 40 edges, the so called Folkman graph. Additionally he showed that any semi-symmetric graph must not have $2p$ or $2p^2$ vertices for p prime. Since then, cubic and 4-valent semi-symmetric graphs have been studied extensively. Cubic graphs have recently been completely classified up to 768 vertices [CMMP06]. Some authors have discovered 5-valent semi-symmetric graphs. For example, in the process of constructing an infinite family of semi-symmetric graphs with varying valences, Lazebnik and Raymond discovered three 5-valent semi-symmetric graphs [LV02]. However, it does not appear that any author has studdied valence 5 graphs in particular.

In this project I investigate the properties of semi-symmetric graphs of valence 5. Sections 2 and 3 cover background material on group theory and graph theory. This material may be safely skipped by any reader already comforable with this material. Additional background in group theory may be found in "Algebra" by Grove [Gro83]. More on graph theory can be found in [AG07]. Section 4 includes background material on how graphs may be analyzed via group theory; here group actions are applied to the graph, and semi-symmetric graphs are defined. Most readers will benefit from the material in this section, but again, it may be skipped by readers who are already familiar with it. The remainder of the thesis is in two parts. First, I describe computer search techniques used to find some semi-symmetric graphs of valence 5, and then I provide a construction of such a graph.

2 Groups

Groups are of fundamental important throughout mathematics, especially in the study of symmetry. To understand semi-symmetric graphs, we must be able to describe relationships between different symmetries. Group theory provides the necessary tools to do this. In the following discussion, I define groups, group actions, and a few theorems necessary for our work. It is assumed that the reader has a basic understanding of group theory and its terminology, but it is covered here for reference.

Definition 2.1 (Groups)**.** A *group* is a set equipped with a binary operation which is associative, has a two-sided identity and under which every element has a two-sided inverse. This product is often called "multiplication" regardless of how it is actually defined. Multiplication is typically denoted by concatenation. Subsets which are themselves groups under the restriction of the binary operation are called *subgroups*. The notation $H \leq G$ is used to indicate that H is a subgroup of G.

Examples of groups include the integers under addition and the nonzero reals under multiplication. The finite *cyclic groups* $\mathbb{Z}_n$ are an important family of examples which are defined as the integers reduced modulo n under addition modulo n.

Another important example is the *symmetric group* of a set X. This is the group Sym_X formed by considering all bijections $\pi : X \to X$ under the operation of function composition. Symmetric groups are used to define group actions. Group actions describe the way a group can permute a set of elements. For this research it is used to describe the way that a "symmetry" or "automorphism" of a graph acts on its vertices.

The following theorem is useful in a wide variety of situations to bound the size of subgroups of a group. This is an indispensable tool for determining the size of a group, or its subgroups.

Theorem 2.2 (Lagrange)**.** *If G is a group and H is a subgroup of G, then $|H|$ divides $|G|$.*

Definition 2.3 (Cosets)**.** Given a group G, a subgroup H and an element $g \in G$, the set $gH = \{gh : h \in H\}$ is called a *left coset of H in G*. Similarly $Hg = \{hg : h \in H\}$ is a *right coset of H in G*. The set of all left cosets of H in G partition G into equivalence classes. The set is denoted by G/H. The size of G/H is denoted by $[G : H]$ and called the *index of H in G*. H is called a *normal subgroup* if $gH = Hg$ for all $g \in G$. In this case, G/H is a group under the operation $g_1H \cdot g_2H = (g_1g_2)H$, and this operation is well-defined. The resulting group $(G/H, \cdot)$ is called the *quotient group*.

Definition 2.4 (Kernels)**.** Given two groups G and H, a function $f : G \to H$ is called a *group homomorphism* if $f(xy) = f(x)f(y)$ for all $g \in G$. A bijective group homomorphism is called a *group isomorphism*. If f is a group isomorphism from G to H then G and H are said to be *isomorphic*. This is written as $G \cong H$. The *kernel* of a morphism is the set $\ker f = \{g \in G : f(g) = 1_H\}$. The kernel of a morphism is trivial exactly when the morphism is injective.

Theorem 2.5 (Fundamental Homomorphism Theorem)**.** *Normal subgroups are exactly the groups that are kernels of homomorphisms. In particular, if $f : G \to H$ is a surjective group homomorphism, then $G/\ker(f) \cong H$.*

Definition 2.6 (Generating Set)**.** Let G be a group, and $S \subset G$. The *subgroup of G generated by S* is the intersection of all subgroups of G containing H. This subgroup is denoted by $\langle S \rangle$, and S is referred to as the *generating set* for $\langle S \rangle$. If $H_1, \ldots, H_n$ are subgroups of G then $\langle H_1, \ldots, H_n \rangle$ denotes the subgroup of G that is generated by their union.

The key application of groups to graphs is in the study of group actions. The Orbit-Stabilizer theorem establishes an invaluable relationship used to determine the sizes of sets under group actions.

Definition 2.7 (Group actions)**.** Any group homomorphism $\phi : G \to \mathrm{Sym}_X$ defines a *group action* of the group G on the set X. If $g \in G$ and $x \in X$, notation is typically condensed, and xg denotes $\phi(g)(x)$. Some authors prefer to use a left group action, in which case gx is used to denote $\phi(g)(x)$.

Definition 2.8 (Orbits and Stabilizers)**.** If G acts on a set X and $x \in X$ then the *stabilizer* of x in G is the set $\mathrm{Stab}_G(x) = \{g \in G : xg = x\}$. This set is a subgroup of G. The *orbit* of x in G is the set $\mathrm{Orb}_G(x) = \{xg : g \in G\}$.

Theorem 2.9 (Orbit-Stabilizer Theorem)**.** *If G is a group acting on a set X and $x \in X$, then $[G : \mathrm{Stab}_G(x)] = |\mathrm{Orb}_G(x)|$.*

An important consequence of this theorem is the size of an orbit or stabilizer always divides the size of the group. The orbits of X in G partition X into equivalence classes. A group action is said to be *transitive* if there is only one orbit. Equivalently, for all $x, y \in X$ there exists a $g \in G$ so that $y = xg$. The action is termed *regular* if it is transitive and $|G| = |X|$.

3 Graphs

Graphs are useful for depicting networks and relations. They have far-reaching applications within computer science and mathematics. It is assumed that the reader does have a working knowledge of graph theory. This section is especially in place to help control the plethora of definitions that permeate the literature.

Definition 3.1 (Graphs)**.** A *graph* is an ordered pair $\Gamma = (V, E)$ where V is any set and E is a collection of subsets of V so that $|e| = 2$ for each $e \in E$. The set V is called the *vertex set* and E is called the *edge set*. $V(\Gamma)$ and $E(\Gamma)$ denote the vertex- and edge sets for any graph Γ.

Note that many authors define a graph more generally to have loops, multiple edges and/or directed edges. They would call a graph by the above definition a "simple graph". If Γ is a graph, $v_1, v_2 \in V(\Gamma)$ then v_1 and v_2 are called "vertices". If $\{v_1, v_2\} \in E(\Gamma)$ then v_1 and v_2 are said to be "adjacent" or "neighbours". Of course, we typically think of graphs by drawing pictures vertices connected to other vertices by edges.

Definition 3.2 (Graph homomorphisms)**.** Let Γ and Δ be two graphs. Let $\phi : \Gamma \to \Delta$ be a function, by which we mean ϕ maps members of $V(\Gamma)$ to members of $V(\Delta)$. If ϕ preserves vertex-adjacency, then we say ϕ is a *homomorphism*. To be precise, ϕ is a homomorphism if and only if $v_1, v_2 \in V(\Gamma)$ are adjacent if and only if $\phi(v_1), \phi(v_2)$ are adjacent in Δ. A *graph isomorphism* ϕ is a bijective homomorphism. If $\phi : \Gamma \to \Delta$ is an isomorphism, we say that Γ and Δ are *isomorphic*.

Like in many branches of mathematics, the notion of a homomorphism of graphs corresponds to maps that are structure preserving. Similarly, we have the notion that two graphs are identical, and share all the same properties, if they are isomorphic. The following terms are part of the standard vocabulary in graph theory.

Definition 3.3 (Regular Graph)**.** A graph Γ is *k-valent* or *k-regular* if every vertex has exactly k neighbours. Any such graph is called a *regular* graph. A vertex v with k neighbours is said to have degree k.

Definition 3.4 (Bipartite)**.** A graph Γ is *bipartitie* if each vertex may be colored "black" or "white" so that no two vertices of the same color are adjacent.

Example 3.5. The canonical example of a regular graph is the family of complete graphs. The *complete graph on n vertices* is denoted K_n. Its vertex set has n elements, and there is an edge between every pair of vertices.

Example 3.6. Another family of graphs are the complete bipartite graphs. The graph $K_{n,m}$ has n white vertices and m black vertices such that every white vertex u is adjacent to every black vertex v.

Definition 3.7 (Paths and Cycles)**.** A *path* in Γ is a sequence of unique vertices $v_0, v_1, \ldots, v_n$ so that each vertex is adjacent to its successor and predecessor. The vertex v_0 is the *initial vertex* while v_n is the *final vertex*. If p is a path with initial vertex u and final vertex v then we say p is a u, v-path. The graph Γ is *connected* if for each vertex u, v there is a u, v-path. A *cycle* in Γ is a path except that the initial vertex and the final vertex are the same. A path containing n vertices is called an *n-cycle*.

The following definitions and constructions are less standard.

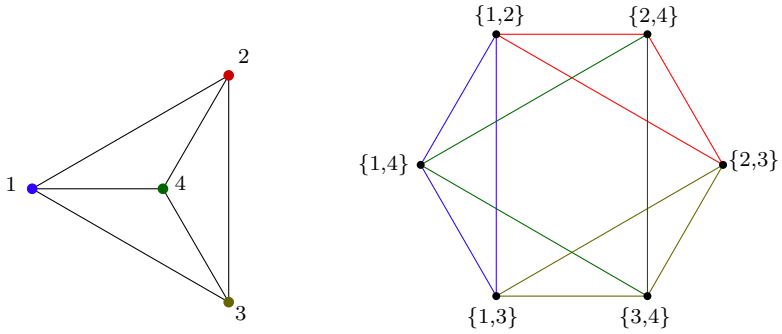
Definition 3.8 (Worthy)**.** A graph Γ is *worthy* if there are no two vertices with the same neighbors. With the exception of K_1 and $K_{1,1}$, all complete and complete bipartite graphs are worthy.

Given a graph Γ, its line graph is defined by turning its edges into vertices, and its vertices into edges. The following definition makes this notion precise.

Definition 3.9 (Line graphs)**.** Suppose $\Gamma = (V, E)$ is a graph. Let $V' = E$. For any pair $\{e_1, e_2\}$ of distinct edges in Γ, they are adjacent (as vertices) in the line graph if and only if they were adjacent (as edges) in the original graph. Using notation, $\{e_1, e_2\} \in E'$ if and only if $|e_1 \cap e_2| = 1$. The graph $\mathcal{L}(G) = (V', E')$ is called the *line graph of* Γ.

Contrary to intuition, if Γ is a graph then generally (and in fact, usually) $\mathcal{L}(\mathcal{L}(G)) \neq G$, as this next example demonstrates.

Example 3.10. In the figure below, on the left is the complete graph on four vertices, K_4. Its line graph, $\mathcal{L}(K_4)$ is the octahedral graph on the right. The vertices of K_4 have been colored to match the corresponding edges in its line graph.



In particular, notice that $\mathcal{L}(\mathcal{L}(K_4))$ has 12 vertices, and so is not isomorphic to K_4. In general, if Γ is a graph, then $\mathcal{L}(\Gamma)$ has more edges than Γ has vertices, even though $\mathcal{L}(\Gamma)$ has as many vertices as Γ has edges.

Voltage graphs are useful for describing a larger, derived graph, in terms of a smaller graph. This is a very general construction which can be used to construct large families of graphs easily.

Definition 3.11 (Voltage Graphs)**.** A *directed graph* is a graph where each edge has a direction. Formally, such an edge is represented by an ordered pair of vertices. A *multigraph* is one where the edges form a multiset, so there can be more than one edge between a pair of vertices. A *weighted graph* is a graph Γ along with a function $w : E(\Gamma) \to \mathbb{Z}$.

Let $n > 1$ be a positive integer, and suppose Δ is a directed, weighted multigraph, where all the weights on the edges belong to the set $\mathbb{Z}_n$. Such a

graph is called a *voltage graph*. From voltage graphs we construct a *derived graph*, Γ, as follows. Take $V(\Gamma) = V(\Delta) \times \mathbb{Z}_n$. The vertices (v, s) and (w, t) are adjacent if v and w are adjacent in Δ and there is a directed edge $(v, w) \in E(\Delta)$ with weight $t - s$ (computed in $\mathbb{Z}_n$). Following this construction, we say that Δ is a voltage graph for Γ, or equivalently, that Γ is the derived graph of Δ. More generally, $\mathbb{Z}_n$ may be replaced with any group, however this is unnecessary for this work.

Definition 3.12 (Cayley Graphs)**.** Let G be a group and $S \subset G$ be an arbitrary subset. Define $\Gamma = \mathrm{Cay}(G, S)$ to be a graph with $V(\Gamma) = G$ and $E(\Gamma) = \{\{g, gs\} | g \in G, s \in S\}$. An edge $\{g, gs\}$ is said to be *colored by* s.

4 Groups and Graphs

The key relationship between groups and graphs is symmetry. Graphs have symmetry, and groups study it! The following construction is a starting point for their interaction.

Definition 4.1 (Automorphism group)**.** Let Γ be a graph, and let S be the set of all graph isomorphisms $\phi : \Gamma \to \Gamma$. These isomorphisms are called *graph automorphisms* or sometimes *symmetries*. $(S, \circ)$ is a group, where $\circ$ denotes function composition. This group is called the *automorphism group of* Γ and is denoted by $\mathrm{Aut}(\Gamma)$. If Γ is bipartite, we also define $\mathrm{Aut}^+(\Gamma)$ to be the set of all *color-preserving automorphisms*, that is automorphisms ϕ such that $\phi(v)$ is the same color v for all vertices in Γ.

Notice that $\mathrm{Aut}(\Gamma)$ naturally acts on both $V(\Gamma)$ and $E(\Gamma)$. For any $\phi \in \mathrm{Aut}(\Gamma)$, vertex $v \in V(\Gamma)$, the obvious action is to apply ϕ to v! The automorphism map can also be applied to an edge of Γ to get another edge of Γ. This action is adjacency preserving; that is exactly the definition of a graph homomorphism.

Therefore, we can consider the orbits and stabilizers of vertices and edges in Γ under the action of $G = \mathrm{Aut}(\Gamma)$. The graph is *vertex transitive* if G acts transitively on the vertices of Γ. Similarly, it is *edge transitive* if G acts transitively on the vertices. A graph is *dart transitive* if for any two pairs of adjacent vertices (s, t) and (u, v) there exists $g \in G$ so that $(s, t) = (ug, vg) = (u, v)g$. This is stronger than edge-transitivity.

Intuitively, two vertices belonging to the same orbit under G "look" the same. Specifically, if $\phi \in G$ and $v \in V(\Gamma)$ then $v\phi$ has all the same local properties that v does. For example, the valence of v is the same as $v\phi$. The

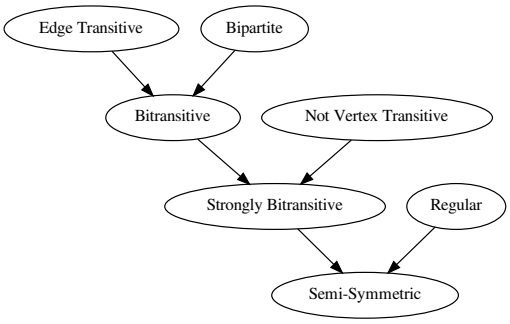


Figure 4.1: Relationships between different definitions related to transitivity and symmetry.

same comment applies to edges; if $e \in E(\Gamma)$ the $e\phi$ and e share properties local to those edges.

The Orbit-Stabilizer theorem (Theorem 2.9) guarantees that if Γ is vertex transitive, then the number of vertices of Γ divides the size of its automorphism group. Similarly, if Γ is edge transitive, then the number of edges must divide the size of the automorphism group. Essentially, being edge or vertex transitive guarantees the graph has many symmetries.

Let Γ be an edge transitive graph. Γ is called *symmetric* if it is dart and vertex transitive. It is $\frac{1}{2}$*-arc transitive* if it is vertex transitive but not dart transitive. It is *strongly bitransitive* if it is not vertex nor dart transitive. Strongly bitransitive graphs are exactly those bipartite and edge transitive graphs for which $\mathrm{Aut}(\Gamma) \neq \mathrm{Aut}^+(\Gamma)$. A graph is *semi-symmetric* if it is regular and strongly bitransitive.

Equivalently stated, a semi-symmetric graph is one that is regular and edge transitive but not vertex transitive. These relations are depicted in Figure 4.1.

Proposition 4.2. *Every edge transitive graph is symmetric,* $\frac{1}{2}$*-arc transitive or strongly bitransitive. In particular, there are no edge transitive graphs that are dart transitive but not vertex transitive.*

Definition 4.3 (Semisymmetric)**.** A graph Γ is *semi-symmetric* if it is regular and edge transitive but not vertex transitive.

Proposition 4.4. *Strongly bitransitive graphs, including semi-symmetric graphs, are bipartite.*

Theorem 4.5 (Folkman)**.** *The smallest semi-symmetric graph is the 4-valent Folkman graph on 20 vertices. See figure 4.2.*

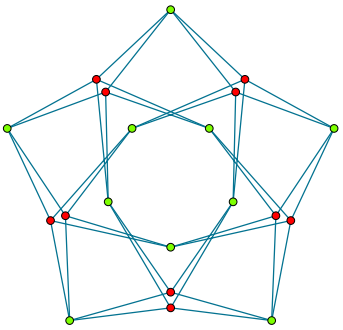


Figure 4.2: The Folkman Graph [Epp08]

In general, semi-symmetric graphs are sparse. Folkman initiated the study of these graphs with the Folkman graph [Fol67]. Many researchers since have sought to classify semi-symmetric graphs. One of the most remarkable accomplishments has been the classification of 3-valent semi-symmetric graphs with up to 768 vertices [CMMP06]. There are only 43 such graphs. The smallest is the *Grey graph* on 54 vertices. The goal of this project is to expand the work on semi-symmetric graphs to higher valences, starting with five valent graphs. Important questions include "what is the smallest 5-valent semi-symmetric graph?" and identifying infinite families of 5-valent semi-symmetric graphs.

4.1 The Bi-Coset Construction

Definition 4.6 (Bitransitive)**.** A graph Γ is *bitransitive* if it is edge transitive and bipartite.

By Proposition 4.4, we are guaranteed that all semi-symmetric graphs are bitransitive. Now I will introduce a particular method to construct bitransitive graphs from groups. This is called the bicoset construction. This construction is invaluable because all bitransitive graphs may be derived from it.

Definition 4.7. Let G be a group and H, K subgroups of G. The *bicoset graph*, $\mathrm{bcc}(G; H, K)$, is a bipartite graph defined as follows. Let the set G/H be the black vertices and G/K the white vertices. Two vertices (which are themselves equivalence classes) are adjacent if their intersection is nonempty. This construction has a number of nice properties, stated below as propositions, some of which are proved.

Proposition 4.8 (Bi-coset Construction)**.** *For every group G with subgroups H and K, the bicoset graph* $\mathrm{bcc}(G; H, K)$ *is bitransitive.*

Proposition 4.9 (Connectivity)**.** *Let G be a group with subgroups H and K. The bicoset graph* $\mathrm{bcc}(G; H, K)$ *is connected iff H and K generate G.*

Proposition 4.10. *The valence of a coset of H is given by $[H : H \cap K]$. Similarly the valence of a coset of K is given by $[K : H \cap K]$.*

Proof. Let $\Gamma = \mathrm{bcc}(G; H, K)$. Let $L = H \cap K$. One can construct a bijection f from cosets in G/L to the edges of Γ as follows. Let $f(Lg) = \{Hg, Kg\}$. This is well defined since if $Lg_1 = Lg_2$ then both $Hg_1 = Hg_2$ and $Kg_1 = Kg_2$. The map is surjective because each edge is of the form $\{Hg, Kg\}$ for some $g \in G$, so $f(Lg) = \{Hg, Kg\}$. To see that f is injective, suppose that $f(Lg_1) = f(Lg_2)$. Then $Hg_1 = Hg_2$ and $Kg_1 = Kg_2$. This implies $Hg_1 \cap Kg_1 = Hg_2 \cap Kg_2$ so $Lg_1 = Lg_2$. This bijection demonstrates that $|E(\Gamma)| = [G : L]$.

The valence of each black vertex (a coset of H in G) is constant since Γ is bitransitive. There are $[G : H]$ black vertices. Therefore each black vertex is adjacent to exactly $[G : L]/[G : H] = [H : L]$ edges. The same argument establishes that the valence of a coset of K is given by $[K : L]$. □

Corollary 4.11. $\Gamma = \mathrm{bcc}(G; H, K)$ *is regular if and only if $|H| = |K|$. Γ is 5-valent if $[H : H \cap K] = [K : H \cap K] = 5$.*

Proposition 4.12. *If Γ is bitransitive then there exist groups G, H, K such that $\Gamma \cong \mathrm{bcc}(G; H, K)$.*

Proof. Suppose Γ is a bitransitive graph, and let $G = \mathrm{Aut}^+(\Gamma)$. Pick a white vertex u adjacent to a black vertex v. Let H and K be the stabilizers of these two vertices, respectively, under the action of G. We will see that $\Gamma \cong \mathrm{bcc}(G; H, K)$.

If u' is any white vertex, there exists g such that $ug = u'$. For fixed u', the set of all g with this property is exactly a coset of H, namely $\{t \in G | ut = u'\} = Hg$. Similarly, if v' is any black vertex, there exists g' so that $vg' = v'$ and $\{t \in G | vt = v'\} = Kg'$.

Let $\phi : \Gamma \to \mathrm{bcc}(G; H, K)$ be defined as follows. If $u' \in V(\Gamma)$ is white, then $\phi(u') = \{t \in G | ut = u'\}$. If $v' \in V(\Gamma)$ is black then $\phi(v') = \{t \in G | vt = v'\}$. These are exactly cosets of H and K as we have just shown. To see ϕ is a graph isomorphism, suppose u' and v' are adjacent in Γ. Since Γ is edge transitive, there exists a $g \in G$ such that $u' = ug$ and $v' = vg$. Therefore,

$g \in \phi(u')$ and $g \in \phi(v')$, so $\phi(u') \cap \phi(v')$ is nonempty. This implies $\phi(u')$ and $\phi(v')$ are adjacent.

Finally, observe that ϕ is injective: if $\{t \in G | ut = u'\} = \{t \in G | ut = u''\}$ then certainly $u' = u''$. If Hg is some coset of H, then $\phi(ug) = \{t \in G | ut = ug\} = Hg$, so ϕ is surjective. Hence ϕ is a graph isomorphism. □

5 Searching for Semi-Symmetric Graphs of Valence 5

I searched for 5-valent semi-symmetric graphs programatically via the bicoset construction. The computer algebra system Magma comes with a database of groups suitable for this. The algorithm I used is as follows.

1. Pick a group G from a database of groups.
2. For each subgroup $H \leq G$, do:
   (a) For each conjugacy class of subgroups of G, pick a representative K so that $[H : H \cap K] = [K : H \cap K] = 5$ and $\langle H, K \rangle = G$, if possible.
   (b) For each K, construct $\Gamma = \mathrm{bcc}(G; H, K)$.
   (c) If Γ is not vertex transitive, then append it to the output.

Using this algorithm, I found three graphs of interest. The graphs had 120, 240 and 250 vertices. Of these, only the one with 250 vertices was previously known; it was constructed by Lazebnik in [LV02]. Details on these graphs, including complete descriptions via adjacency lists in various formats, may be found at `http://www.berkeleychurchill.com/research/census`. Source code I used is in appendix section 1.2.

I completed this procedure for every finite group with $|G| \leq 1200$. It is possible that the graphs I found were not the smallest ones. Indeed, the automorphism group of the Folkman graph has 3840 elements while the graph itself has only 20 vertices. In comparison, there are also two non-isomorphic semi-symmetric graphs on 40 vertices with valence four whose automorphism groups have only 80 elements. An obvious objective is to describe which graphs this computer search has missed.

One special case is of graphs whose color preserving automorphism groups contain a subgroup H so that H acts regularly on the edges. I say such a graph is *edge regular*. A graph Γ is edge regular if and only if there is a subgroup H of $\mathrm{Aut}(\Gamma)$ so that $\mathrm{Stab}_H(e) = 1$ for each $e \in E(\Gamma)$. If Γ were

semi-symmetric and edge regular, then the corresponding H could be used to construct Γ; namely, $\mathrm{bcc}(H, \mathrm{Stab}_H(v), \mathrm{Stab}_H(w)) \cong \Gamma$ where v is any white vertex and w is any black vertex. Thus, I have found all three of the edge regular semi-symmetric graphs of valence 5 with less than 1200 edges. The following theorem provides a description of the exact kind of graphs these are.

Theorem 5.1. *A connected bitransitive graph Δ is edge regular if and only if there exists a group G and a subset $S \subset G$ such that $\Gamma = \mathcal{L}(\Delta) \cong \mathrm{Cay}(G, s)$*

To prove this theorem, a few lemmas are necessary. The first lemma is used to show the first direction, while the second lemma shows the converse.

Lemma 5.2. *If G is a group with subgroups H and K so that $H \cap K = 1$ and $\langle H, K\rangle = G$, then $\mathcal{L}(\mathrm{bcc}(G; H, K)) \cong \mathrm{Cay}(G, H \cup K\backslash\{1\})$.*

Proof. Let $\Delta = \mathrm{bcc}(G; H, K)$ and $\Gamma = \mathcal{L}(\Delta)$. Then the vertices of Γ are the edges of $\mathrm{bcc}(G; H, K)$, which is exactly the set of cosets $G/(H \cap K)$. Hence, there is one vertex of Γ per member of G, and these vertices can be labeled as members of G. If g_1, g_2 are two distinct vertices, then they are adjacent if and only if g_1 and g_2 share a vertex in the bicoset construction. This happens when g_1 and g_2 either belong to the same coset of H in G or they belong to the same coset of K in G. That is, if $g_2 g_1^{-1} \in H \cup K\backslash\{1\}$. So, g_2 may be written as xg_1 for some $x \in H \cup K\backslash\{1\}$. Therefore, every edge of Γ is of the form $\{g, xg\}$ with $g \in G$. This set of edges is exactly the set of edges in $\mathrm{Cay}(G, H \cup K\backslash\{1\})$. The isomorphism is the identity. □

Lemma 5.3. *If Δ is a connected bipartite graph, $\Gamma = \mathcal{L}(\Delta)$ and G is a subgroup of $\mathrm{Aut}(\Gamma)$ then G acts as a group of automorphisms on Δ. Moreover, G acts on $V(\Gamma)$ exactly as it does on $E(\Delta)$.*

Proof. Since Δ is bi-partite its vertices may be colored black and white. This induces a non-proper 2-coloring of the edges of Γ. When discussing colored edges, these edges must be members of $E(\Gamma)$. When discussing colored vertices, these vertices are in $V(\Delta)$. Notice that every vertex v of Δ with degree $d > 1$ has a corresponding d-clique in Γ and every edge of this d-clique has the same color as v.

Consider any d-clique K in Γ for some $d > 1$. Suppose every edge of K is the same color. Then there are d edges in Δ that are each pairwise adjacent. Consider the subgraph H of Δ containing these edges and the vertices that are adjacent to more than one edge. H must be connected and bi-partite. Therefore if H contains more than one vertex there must be two

colors of vertices, so K contains two colors of edges. This contradicts our premise. Therefore if all the edges of K are the same color, these edges all correspond to a single vertex in Δ.

Now suppose that K has both black and white edges. This never happens when $d \leq 2$. Consider the case when $d \geq 3$. Pick a 3-clique subgraph of K and call it K'. It has three edges. By pigeonhole principle, at least two edges have the same color. Without loss of generality, suppose there is one white and two black edges. Again consider the connected bi-partite subgraph H of Δ consisting of the corresponding three edges and all vertices that are adjacent to more than one of them. There must be at least one black and one white vertex in this subgraph. Case 1: These are the only two vertices in H. However there are three edges, which implies that H has a multi-edge or H is a path graph, either of which is a contradiction. Case 2: There are three vertices in H, two of which are black and one is white. Then these is a 1-1 correspondence between these vertices and the edges of K. Hence the two black vertices must be adjacent, which contradicts the proper coloring of Δ. Therefore it is impossible for K to have both black and white edges.

Hence every clique, including 2-cliques, of Γ has a distinct color and corresponds to a subset of edges adjacent to a vertex of Δ. Let U be the set of maximal cliques in Γ, that is, the cliques which are not subgraphs of a larger clique. The black and white maximal cliques each form a block system. To see this, suppose $(u_1, u_2) \in K_1$ and K_1 is a maximal black d-clique. Then (u_1, u_2) is part of exactly one maximal d-clique, so if $g \in G$ then $(u_1, u_2)g = (v_1, v_2)$ must be part of another maximal d-clique, K_2. It follows that all the other edges and vertices in K_1 must map into K_2. Note that K_1 and K_2 need not have the same color if Δ is regular. Also observe that if K_1 and K_3 are vertex-adjacent then $K_1 g$ and $K_3 g$ are vertex-adjacent.

U is in one-to-one correspondence with the vertices of Δ. In addition, this correspondence preserves vertex-adjacencies in U with edge-adjacencies in Δ. Therefore G acts on Δ. □

Finally, the proof of theorem 5.1 follows.

Proof. First, suppose Δ is edge regular. Let G be a subgroup of $\mathrm{Aut}(\Delta)$ that acts regularly on the edges. Let H and K be the vertex stabilizers of an adjacent black and white vertex, repectively. By the argument in Proposition 4.12, $\Delta \cong \mathrm{bcc}(G; H, K)$. Moreover, by Proposition 4.9 we know that H and K generate G since Δ is connected. Finally, Lemma 5.2 establishes that $\Gamma = \mathcal{L}(\Delta) \cong \mathrm{Cay}(G, H \cup K \backslash \{1\}$.

In the other direction, suppose $\Gamma \cong \mathrm{Cay}(G, S)$. Then G acts regularly on the vertices of $\mathrm{Cay}(G, S)$. By the correspondence illustrated in Lemma 5.3, G acts on Δ and regularly on its edges. $\square$

6 Constructing a Semi-Symmetric Graph of Valence 5

The majority of my effort has gone toward providing a combinatorial description of the graph on 120 vertices, and proving it is semi-symmetric, and generalizing it to an infinite family. The most useful representation of this graph has been a voltage graph. This voltage graph may be constructed as follows. Start with the graph of an icosahedron. Replace each vertex of the icosahedron with a white vertex and a black vertex. In this new graph, two vertices are adjacent if and only if they are of different colors and they correspond to adjacent vertices in the original icosahedral graph. This is a standard construction, and is called the *bipartite double cover* of the icosahedron. Pick an orientation on this graph, that is, pick a direction for each edge so it is now a directed graph. Call this graph Δ.

Set $n = 5$. We will construct weights for the edges of Δ so that Δ is a voltage graph for the semi-symmetric graph on 120 vertices. For each edge $(u, v) \in E(\Delta)$ let $x_{(u,v)} \in \mathbb{Z}_5$ denote its weight. Note one may freely use $x_{(v,u)} = -x_{(u,v)}$. We will construct a system of equations to solve for these values. I call a pair of black and white vertices in Δ *friends* if they correspond to the same vertex in the icosahedral graph. Let A, B, C be black vertices with friends X, Y, Z, respectively, so that A, Y, C, X, B, Z is a 6-cycle in Δ. Re-name these vertices so that the cycle moves clockwise around the face of the icosahedron. This cycle corresponds to a triangle in the icosahedral graph. Introduce the equation

$$x_{(A,Y)} + x_{(Y,C)} + x_{(C,X)} + x_{(X,B)} + x_{(B,Z)} + x_{(Z,A)} = 1$$

into the system. Consider a 4-cycle A, X, B, Y where A and B are black vertices and X and Y are white vertices. Ensure that this cycle is traversed counter-clockwise. There are two cases. If A and B are adjacent as vertices in the icosahedral graph, then introduce the equation

$$x_{(A,X)} + x_{(X,B)} + x_{(B,Y)} + x_{(Y,A)} = 1.$$

Otherwise, X and Y will be adjacent in the icosahedral graph, and one should introduce

$$x_{(A,X)} + x_{(X,B)} + x_{(B,Y)} + x_{(Y,A)} = 2.$$

These equations are enough to fully determine the structure of the graph, as described in the following theorem.

Theorem 6.1. *Any solution to the above system yields a voltage graph on Δ, and the corresponding derived graphs are all isomorphic to the same semi-symmetric graph on 120 vertices.*

Proof. The proof of this theorem is in three parts. First I argue that any two solutions yield isomorphic graphs. Then I show that this graph is edge transitive using a symmetry argument. Finally, it is not vertex transitive because some white vertices are distance six from one another, while black vertices are at most four apart.

A particular assignment of voltages corresponding to the graph with 120 vertices may be written as a vector. The set of all such voltages forms an affine space where elements may be written as $a + w$, where a is a vector of starting voltages, and w is a member of a vector space W which represents a transformation to this set of voltages. I will show that for any voltage graph on Δ, the space W is at least 23 dimensional.

Suppose you have a set of voltages. Then, for a particular vertex v in the voltage graph, you can relabel ("pop") the derived vertices with the map $(v, i) \mapsto (v, i + 1)$. This corresponds to increasing the voltages on all the incoming edges in the voltage graph by one, and decreasing the voltage on the outgoing edges. Each of these is a transformation in W, so W contains at least 24 vectors. Of course, popping every vertex causes no change in the labeling of the graph, so these 24 vectors are not linearly independent. However, any 23 of them are.

To see that 23 of them are linearly independent, pick any minimal spanning tree of the voltage graph, and assign new weights to each edge. Pick one vertex to not pop. Then, working along each edge of the tree, each other vertex must be popped a unique number of times; i.e., there's exactly one linear combination of the other 23 vectors that gives this voltage. So, W has dimension at least 23.

Write the system of equations above in the form $Ax = b$. Every x satisfying $Ax = 0$ is a transformation of the voltages. Every "popping" vector found above satisfies the systems. This is easy to see because in each cycle there are as many in-edges as there are out-edges at each vertex. If each in-edge is incremented by one and each out-edge is decremented by one, then one can check that each equation stays invariant. These vectors form a 23

dimensional subspace of $\ker A$. It turns out that, by computer computation, that $\ker A$ has dimension exactly 23. The matrix corresponding to the system was explicitly computed. This code can be found in appendix section 1.1.

Let y and y' be solutions to $Ax = b$. Then $y - y' \in \ker A$. This means that y can be reached from y' by a series of popping vertices. Hence, the derived graph corresponding to y is isomorphic to the derived graph corresponding to y'. This concludes the first part of the proof.

Any rigid symmetry of the icosahedron is also a symmetry of the space of solutions to the voltage equations. This is clear because any symmetry of the icosahedron maps rhombi to rhombi and triangles to triangles while preserving orientation. Consider any two edges, e_1, e_2 in the derived graph, Γ. Write $e_1 = (f_1, i)$ and $e_2 = (f_2, j)$ where f_1 and f_2 are the corresponding edges in Δ. The graph Δ is edge transitive under rigid motions, so there is some automorphism ϕ of Δ which maps f_1 to f_2. This map permutes the voltages and yields a new voltage graph Δ'. Since the system of equations is preserved under rigid symmetries, the derived graph of Δ' is still isomorphic to Γ. This naturally induces an automorphism of Γ which sends (f_1, i) to (f_2, i). Composing this with the map that sends (f, t) to $(f, t + j - i)$ yields an automorphism sending e_1 to e_2. Hence, Γ is edge transitive.

Finally, one can manually check (or use a computer), that the distance between any two black vertices is no more than four. However, antipodal white vertices can be distance six apart. Therefore, there is no automorphism taking black vertices to white vertices, so the derived graph is not vertex transitive. Thus, the derived graphs are 5-valent and semi-symmetric. □

7 Conclusion

Hopefully one can generalize the construction provided in this paper to find a family of 5 valent semi-symmetric graphs. By solving systems of equations similar to that in Section 6, it should be easy to find several examples of candidate edge transitive graphs. However, determining which of these will not be vertex transitive will be harder. Future work on this project will hopefully include identifying an infinite family of these graphs.

References

[AG07] Geir Agnarsson and Raymond Greenlaw, *Graph theory: Modeling, applicatons and algorithms*, Pearson Prentice Hall, 2007.

[CMMP06] Marston Conder, Aleksander Malnic, Dragan Marusic, and Primz Potocnik, *A census of semisymmetric cubic graphs on up to 768 vertices*, Journal of Algebraic Combinatorics **23** (2006), no. 3, 255–294.

[Epp08] David Eppstein, *The Folkman graph, the smallest semi-symmetric graph, discovered in 1967 by J. Folkman.*, Public Domain Image, April 2008.

[Fol67] Jon Folkman, *Regular line-symmetric graphs*, Journal of Combinatorial Theory **3** (1967), no. 3, 215 – 232.

[Gro83] Larry C. Grove, *Algebra*, Academic Press, 1983.

[LV02] Felix Lazebnik and Raymond Viglione, *An infinite series of regular edge- but not vertex-transitive graphs*, Journal of Graph Theory **41** (2002), no. 4, 249–258.

A Source Code

Computations were vital to discovering 5-valent semi-symmetric graphs and providing their mathematical construction. A haskell program was used to transform the system of equations in Section 6 into a matrix. The resulting matrix was used to calculate the kernel using the computer algebra system Sage. Additionally, code to search for semi-symmetric graphs was written in magma. Included is the source code for both programs.

1.1 Haskell Source

```
{- Modules -}

import Data.List

{- Constructors -}

data Point = Point Expr Expr Expr Bool deriving (Eq,Show,Ord)
```

```
data Expr = Expr Rational Rational deriving (Eq,Show,Ord)

{- Expressions in Q[Phi] -}

zero = Expr 0 0
one = Expr 1 0
phi = Expr 0 1
mone = Expr (-1) 0
mphi = Expr 0 (-1)

times :: Expr -> Expr -> Expr
times (Expr a b) (Expr c d) = Expr (a*c + b*d) (b*c + a*d + b*d)

plus :: Expr -> Expr -> Expr
plus (Expr a b) (Expr c d) = Expr (a + c) (b + d)

neg :: Expr -> Expr
neg (Expr a b) = Expr (-a) (-b)

minus :: Expr -> Expr -> Expr
minus u v = plus u (neg v)

times_c :: Expr -> Rational -> Expr
times_c (Expr a b) c = Expr (a*c) (b*c)

divide :: Expr -> Expr -> Maybe Expr
divide _ (Expr 0 0) = Nothing
divide (Expr a b) (Expr c 0) = Just $ Expr (a/c) (b/c)
divide x y = divide (times x (conj y)) (times y (conj y))
    where conj :: Expr -> Expr
          conj (Expr u v) = Expr ((-u) -v) v

sq :: Expr -> Expr
sq u = times u u

to_flt :: Expr -> Rational
to_flt (Expr a b) = a + b*(1 + (toRational (sqrt 5)))*(1/2)

{- Points -}
same_color :: Point -> Point -> Bool
```

```
same_color (Point _ _ _ x) (Point _ _ _ y) = (x == y)

adjacent :: Point -> Point -> Bool
adjacent x y = (not ( same_color x y )) && (norm_dist x y == Expr 4 0)

adjacent_nc :: Point -> Point -> Bool
adjacent_nc x y = (norm_dist x y == Expr 4 0)

adjacency_list :: Point -> [Point]
adjacency_list x = [ y | y <- nodes, adjacent x y]

adjacency_list_large :: Point -> [Point]
adjacency_list_large x = [ y | y <- nodes, adjacent_nc x y]

triangle :: Point -> Point -> Point -> Bool
triangle x y z = (adjacent x y) && (adjacent y z) && (adjacent z x)

label_expr :: Expr -> String
label_expr (Expr 1 0) = "A"
label_expr (Expr (-1) 0) = "B"
label_expr (Expr 0 1) = "P"
label_expr (Expr 0 (-1)) = "Q"
label_expr (Expr 0 0) = "Z"

label :: Point -> Int
label p = locate nodes p
{-(foldr (++) "" (map label_expr [x,y,z])) ++
                            (if s then "-W" else "-B")-}

label2 :: [Point] -> [Int]
label2 t = map (\x -> label x) t

label3 :: [[Point]] -> [[Int]]
label3 t = map (\x -> label2 x) t

{- Geometry of Points -}

    -- functor for binary operations
efmap :: (Expr -> Expr -> Expr) -> (Point -> Point -> Point)
efmap f = \x -> (\y ->
```

```
        let Point x1 x2 x3 _ = x
            Point y1 y2 y3 _ = y
        in  Point (f x1 y1) (f x2 y2) (f x3 y3) False)

    -- pointwise binary operations
p_plus :: Point -> Point -> Point
p_plus = efmap $ plus

p_minus :: Point -> Point -> Point
p_minus = efmap $ minus

p_times :: Point -> Point -> Point
p_times = efmap $ times

    -- distances
norm :: Point -> Expr
norm (Point a b c _) = foldl plus (Expr 0 0)
    (map (\x -> sq x) [a,b,c])

norm_dist :: Point -> Point -> Expr
norm_dist u v = norm (p_minus u v)

    -- products
cross :: Point -> Point -> Point
cross (Point x1 x2 x3 s) (Point y1 y2 y3 t) = Point z1 z2 z3 (not (s == t))
    where z1 = (minus (times x2 y3) (times x3 y2))
          z2 = (minus (times x3 y1) (times x1 y3))
          z3 = (minus (times x1 y2) (times x2 y1))

dot :: Point -> Point -> Expr
dot u v = let (Point a b c _) = (p_times) u v
          in  plus a (plus b c)

    -- angles
mycos :: Point -> Point -> Point -> Expr
mycos u v w= times_c (dot (p_minus w v) (p_minus v u)) (1/4)

sin2 :: Point -> Point -> Point -> Expr
sin2 u v w= times_c (norm (cross (p_minus w v) (p_minus v u))) (1/16)
```

```
    -- there must be a better way to do this with a functor.
p_times_c :: Point -> Rational -> Point
p_times_c (Point x1 x2 x3 b) r = (Point (times_c x1 r)
                                  (times_c x2 r) (times_c x3 r) b)

average :: Point -> Point -> Point -> Point
average s t u | (mycos s t u) == Expr ((-1)/2) 0  =
                p_times_c (p_plus s (p_plus t u)) (1/3)
              | (mycos s t u) == Expr ((-1/2)) (1/2) =
                let adj = adjacency_list_large in
                head $ intersect (adj s) (intersect (adj t) (adj u))

cw :: Point -> Point -> Point -> Bool
cw s t u = let Point a1 a2 a3 _ = average s t u
               Point c1 c2 c3 _ = cross (p_minus t s) (p_minus u t)
               b1 = divide a1 c1
               b2 = divide a2 c2
               b3 = divide a3 c3
           in  foldl (&&) (True)
               (map (\x -> if x == Nothing then True else
                           (let Just y = x in (to_flt y > 0)))
                                          [b1,b2,b3])

{- Nodes -}

nodes :: [Point]
nodes = quicksort [ x y | x <- ico_node, y <- [True, False] ]
        where ico_node = type1 ++ type2 ++ type3
              type1 = [Point zero s t | s <- [one, mone], t <- [phi,mphi]]
              type2 = [Point t zero s | s <- [one, mone], t <- [phi,mphi]]
              type3 = [Point s t zero | s <- [one, mone], t <- [phi,mphi]]

rhombi4 :: Int -> [[Point]]
rhombi4 1 = map (\x -> [x]) nodes
rhombi4 n = [ old ++ [x] | old <- (rhombi4 (n-1)), x <- nodes,
                        (x > (head old)),
                        (adjacent (x) (last old)),
                        (not (elem x old)),
                        if n > 2 then cw (almost_last old) (last old) x
                                 else True]
```

```
rhombi4' = map (reverse) (filter (\x -> adjacent (head x) (last x))
            (rhombi4 4))

rhombi_black_adj :: [Point] -> Bool
rhombi_black_adj [a,b,c,d] = let (Point _ _ _ color) = a in
                             if color == False then adjacent_nc a c
                                                    --a,c black
                                               else adjacent_nc b d
                                                    --b,d black

rhombi4b :: [[Point]]
rhombi4b = filter (rhombi_black_adj) (rhombi4')

rhombi4w :: [[Point]]
rhombi4w = filter (not . rhombi_black_adj) (rhombi4')

triangles6 :: Int -> [[Point]]
triangles6 1 = map (\x -> [x]) nodes
triangles6 n = [ old ++ [x] | old <- (triangles6 (n-1)), x <- nodes,
                         (x > (head old)),
                            -- not the smallest one in the list
                         (adjacent (x) (last old)),
                            -- strongly adjacent to last thing
                         (not (elem x old)),
                            -- not something already there
                         if n > 2 then cw (almost_last old) (last old) x
                                  else True] --going cw

triangles6' :: [[Point]]
triangles6' =
            let triangles6'' = triangles6 6
                triangles6''' = (filter (\x -> adjacent (head x) (last x))
                                (triangles6''))
                triangles6'4 = (filter (\x -> cw (last x) (head x)
                                       (head (tail x))) triangles6''')
                triangles6'5 = (filter (\x -> cw (almost_last x)
                                       (last x) (head x)) triangles6'4)
            in  triangles6'5
```

```
{- Utility Functions -}

locate :: (Eq a) => [a] -> a -> Int
locate l v = locate' 0 l v
               where locate' n [] v = -1
                     locate' n (x:xs) v = if v == x then n
                                          else locate' (n+1) xs v

rotatel :: Int -> [a] -> [a]
rotatel 0 l = l
rotatel n (x:xs) = rotatel (n-1) (xs ++ [x])

rotate :: (Eq a, Ord a) => [a] -> [a]
rotate l = rotatel (locate l (find_min l)) l

find_min :: (Ord a) => [a] -> a
find_min [x] = x
find_min (x:xs) = min x (find_min xs)

quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = (quicksort start) ++ [x] ++ (quicksort end)
                 where start = [y | y <- xs, y <= x]
                       end = [y | y <- xs, y > x]

uniq :: (Ord a) => [a] -> [a]
uniq [] = []
uniq (x:xs) = if elem x xs then uniq xs else x : uniq xs

almost_last :: [a] -> a
almost_last l = head $ tail $ reverse l

{- Print the Answer ! -}

node_number :: Point -> Int
node_number p = locate nodes p

edges :: [ (Int, Int) ]
edges = [ (a, b) | a <- [0..23], b <- [(a+1)..23],
                   adjacent (nodes !! a) (nodes !! b) ]
```

```
nedges :: Int
nedges = length edges

edge_number :: Int -> Int -> Int
edge_number x y = locate edges ((x, y))

edge_number' :: Point -> Point -> (Int, Bool)
edge_number' x y = let a = node_number x
                       b = node_number y
                       c = edge_number (min a b) (max a b)
                   in if (a < b) then (c,False) else (c,True)

node_to_edge_list :: [Point] -> [(Int, Bool)]
node_to_edge_list (x:xs) = (edge_number' (last xs) x) : node_to_edge_list'
                                                                  (x:xs)
                   where node_to_edge_list' :: [Point] -> [(Int, Bool)]
                         node_to_edge_list' [x] = []
                         node_to_edge_list' (x:xs) = (let y = head xs in
                                                      (edge_number' x y) :
                                                      (node_to_edge_list'
                                                      (y:tail xs)))

edge_list_to_string :: [(Int, Bool)] -> String
edge_list_to_string l = (intercalate ","
        (map (\x -> if (lookup x l == Nothing) then "0"
                    else (if (lookup x l == Just True) then "1" else "4"))
                    [0..(nedges-1)]))

make_matrix :: [String] -> IO()
make_matrix l = putStr ( (foldl1 (\x y -> x ++ "," ++ y) l) )

main :: IO()
main = make_matrix $ map (edge_list_to_string . node_to_edge_list)
    (triangles6' ++ rhombi4w ++ rhombi4b)
```

1.2 Magma Source

```
/*
    Inputs: G, a groub
            H, K subgroups of G
    Outputs:
            The bicoset construction graph.
*/
bcc:=function(G,H,K)

    es := {};

    H_coset_count := Index(G,H);
    K_coset_count := Index(G,K);
    vertex_count := H_coset_count + K_coset_count;

    L := H meet K;
    if Order(L) ne 1 then
        LReps := Transversal(G,L);
    else
        LReps := G;
    end if;

    TH := CosetTable(G,H);
    TK := CosetTable(G,K);

    i:=0;
    // loop through representatives of G/L
    for g in LReps do
        Include(~es,{TH(1,g),H_coset_count+TK(1,g)});
    end for;

    X:=Graph< vertex_count | es>;
    return X;

end function;

/* Inputs: G, a group
           H,K, subgroups of G
```

```
   Outputs: 0 if bcc(G,H,K) is not regular, or
            k, if bcc(G,H,K) is k-regular. */
bcc_regular:=function(G,H,K)

    L := H meet K;
    H_index := Index(H,L);
    K_index := Index(K,L);

    if (H_index eq K_index) then
        return H_index;
    else
        return 0;
    end if;

end function;


/*finds subgroups of G (called H,K) so that bcc(G,H,K) gives semisymmetric graph.
    Assumes that H cap K = 1*/
brute_force:=function(G,valence)
    RF:=recformat<graph, G, H, K>;
    output := [];
    divisors := Divisors(IntegerRing() ! (Order(G)/valence));

    sizes := [ Floor(Order(G)/d) : d in divisors];
    //sizes := [valence];
    for e in sizes do
        sg_classes := Subgroups(G: OrderEqual:=e);
        subgroups := {};
        for coset in sg_classes do
            for subgroup in Conjugates(G,coset`subgroup) do
                Include(~subgroups,subgroup);
            end for;
        end for;
        for Hclass in sg_classes do
            H:=Hclass`subgroup;
            for K in subgroups do
                L := H meet K;
                if Index(H,L) eq valence then
                    graph := bcc(G,H,K);
```

```
                    if not IsConnected(graph) then
                        continue;
                    end if;
                    //not filtering for semi-symmetrics.
                    //necessarily bi-transitive already.
                    r := rec<RF | graph:=graph, G:=G, H:=H, K:=K>;
                    Append(~output,r);
                end if;
            end for;
        end for;
    end for;
    return output;
end function;

/* Performs the construction for all groups of a given order */
brute_force_size:=function(valence,order)
    results := [];
    s:=order;

    verts2:=(IntegerRing() ! (s/valence));
    if(IsPrime(verts2)) then
        return [];
    end if;
    bool,root:=IsSquare(verts2);
    if bool then
        if IsPrime(root) then
            return [];
        end if;
    end if;

    number_groups := NumberOfSmallGroups(s);
    for t in {1..number_groups} do
        G:=SmallGroup(s,t);
        output := brute_force(G,valence);
        for x in output do
            for orig in results do
                if IsIsomorphic(orig`graph,x`graph) then
                    continue x;
                end if;
            end for;
```

```
            Append(~results,x);
        end for;
    end for;
    return results;
end function;


/* Finds all semi-symmetric graphs of a given valence
   using groups with size between min_order and max_order. */
brute_force_all:=procedure(valence,min_order,max_order,~results,output_stuff)

    if(#results gt 0) then
        print "WARNING: results did not start empty!";
    end if;

    filename := output_stuff[1];
    System("rm " * filename);
    if #output_stuff gt 1 then
        system_command := output_stuff[2];
    else
        system_command := ";";
    end if;

    min_order := Max(10*valence,min_order);
    //all semisymmetric graphs have at least 20 vertices
    //this really helps to skip order 32...
    max_order := Floor(max_order/valence);
    min_order := Floor(min_order/valence);
    sizes := [valence*i : i in {min_order..max_order}];
    sizes;

    start_time := Realtime();

    for s in sizes do

        str1:="\nTime " * Sprint(Realtime(start_time)) * ":\n";
        str2:="Starting size " * IntegerToString(s) * ".\n";
        printf str1;
        fprintf filename, str1;
        printf str2;
```

```
        fprintf filename, str2;
        System(system_command);

        //do the work!
        if (s mod 64) eq 0 then
            continue s;
        end if;
        X:=brute_force_size(valence,s);
        for x in X do
            for y in results do
                if IsIsomorphic(x`graph,y`graph) then
                    continue x;
                end if;
            end for;
            Append(~results,x);
        end for;

        //Do I/O
        str1:= IntegerToString(#results) *
            " bi-transitive graphs founds so far.\n";
        str2:= "Finished size " * IntegerToString(s) * ".\n";

        printf str1;
        fprintf filename, str1;
        printf str2;
        fprintf filename, str2;

    end for;

        str1:="\nTime " * IntegerToString(Floor(Realtime(start_time))) *
            ": All done!\n\n";
    printf str1;
    fprintf filename, str1;
    System(system_command);

end procedure;

print_results:=procedure(~results,filename)
    for x in results do
        s:=Sprint(x`graph);
```

```
        fprintf filename, s;
    end for;
end procedure;
```