Brandeis University

Undergraduate Thesis in Computer Science

Graphs: Isomorphism, Cycles, Reconstruction and Randomness

Grady Ward

supervised by
Prf. James Storer

April 15, 2016

Contents

Acknowledgments

When I asked Professor Storer to supervise me on a thesis on the graph isomorphism problem, he was hesitant. He only acquiesced when I persuaded him that my future was secure with a fantastic job, and that my primary objective was the pursuit of questions that were of nothing but personal interest to me. In many respects, his skepticism proved well founded.

This work has been incredibly challenging, both in that the body of existing work on GI is so large, and in that few visible niches of it exist which are promising and not thoroughly explored. Over the past year I have poured my time and energy into this project, and have found it unbelievably energizing to do so. I have been thrilled to find interesting properties in problems surrounding GI, and have had an equal number of frustrations in finding that my results had been previously discovered. Moreover, it has been illuminating to begin to understand a hidden world of graph theory which had before seemed either trivial or intractably complex.

I would like to thank Professor Storer for the initial bout of skepticism about this project, as it shaped this project and experience for the better. But I would also like to thank him for the amount of advice and freedom that he has supported me with on this project. It has kept me on track to focus on my real goal for the semester, which was to learn and grow. I have learned advanced techniques in GPU calculation, proof techniques in abstract algebra, and gotten the chance to reason with established problems in new and interesting ways. My skill set has been broadened by this project which has deeply challenged me and always kept me on my toes.

I would like to thank my advisors (Prfs. Storer, Di Lillo and Torrey), the Computer Science department, and my friends and family for all of the different kinds of support and encouragement that have brought me to successful completion of this project.

TL;DR

Given a graph, count the number of closed paths (cycles) of every length that pass through each vertex in the graph. It turns out that we can do this very quickly, relative to how much information it gives us. Counting the number of cycles gives us a vector which describes a kind of local resonance, a description of the localized area around each vertex within the graph. This idea can be transformed into a numerical property, called an invariant, which is highly information dense–able to describe differences between graphs (or vertices) with high probability, but not sufficient to verify that two graphs (or vertices) are the same.

There are many graphs, even over a small number of vertices, but the number is much larger if you consider different labelings of the same graph to be distinct. This difference matters in many contexts, but is most evident through examining random graph generators, which treat different labelings of the same graph as if they are distinct graphs. This is not a problem in and of itself (it makes sense in many practical contexts), but it warps theoretic arguments about the runtime of algorithms over 'random' graphs. It turns out that making up our own random graph generators can actually improve upon this state of affairs in a quantifiable way.

An Open Source Project

An interesting aspect of this thesis has been that I have placed every incremental iteration of my work online, through the git version control system and GitHub. Every element, from my reading notes, to my mid-semester reports, to my code, results, and datasets: everything has been kept in a central repository, and every change has been committed and logged. Alongside this choice, I have decided that every line of code I have written is open source and publicly available for use; licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

There are three reasons to make this thesis as an open source, version controlled work.

1. Research in academia is far too often done in the dark, and only once the conclusion has been drawn and sufficiently polished does the general public get to learn about it. This falsely represents scientific inquiry as a lightning bolt, a blinding and stark progression from correct idea to correct idea.

   In reality we all explore ideas that fail, we all have intuitions that turn out to be incorrect. The reality of research is much more one of lightning's infinitesimally branching electrical charges (with eventual connection and brilliance). Though this report presents a well coordinated and rehearse set of conclusions, this is not a reflection of all of the work that was done. The GitHub source provides the other pieces of this puzzle.

2. I work on a personal laptop, a school desktop, and occasionally a public workstation. Transferring data (of all sorts) between these computers is tiresome and error prone without a VCS.

3. Everyone does better work when they know that their work could be observed.

The work is available on my personal github page, and will be there for the foreseeable future.

Chapter 1

Definitions, Syntax, and the Cycles Invariant

1.1 Graphs

This report describes undirected, unlabeled graphs with no self-loops or multi-edges. Such graphs are represented by an *adjacency matrix*: a square, symmetric, binary matrix with zeros along the diagonal. We will use G to refer to a graph, and A to refer to an adjacency matrix of the graph. When we use A, it will not refer to a specific adjacency matrix, as most graphs can be represented by many matrices.

We will refer to the set of vertices of a graph as $V(G)$, and the set of edges of a graph as $E(V)$. Graphs have N vertices and M edges where $N \in [0, \infty)$ and $M \in [0, E_{max}]$ where $E_{max} = \frac{1}{2}(N)(N-1)$. When discussing specific edges, tuples are symmetric: $(v_1, v_2) = (v_2, v_1)$.

A graph's *complement* or *inverse* is a new graph over the same set of vertices, but where adjacency and non-adjacency are inverted. In formal terms, the graph $\overline{G}$ is G's inverse if $V(\overline{G}) = V(G)$ and $(v_1, v_2) \in E(\overline{G}) \leftrightarrow (v_1, v_2) \notin E(G)$.

The complete graph over N vertices is K_N. The cycle graph over N vertices is C_N. The star graph over N vertices is S_N. The empty graph over N vertices is E_N.

1.2 Labeling and Representing Graphs

A labeling of a graph is a bijection which maps each vertex of $V(G)$ to an integer in the range $[1, N]$. For every possible labeling of a graph, there is a natural adjacency matrix which represents it, namely the matrix where the i^{th} labeled vertex is represented by the i^{th} column and row of the matrix. The number of distinct labelings of a graph and the number of distinct

matrices which represent that graph are equivalent. A graph can have as many as $V!$ labelings, or can have as few as 1 (consider K_N).

An important distinction in this report is on the *representation* of graphs. Most graphs can be represented by several distinct adjacency matrices, but a change to representation does not change the fundamental structure of the graph that the matrices represent. Different representations of graphs are akin to different labelings of the graph: neither mutate structure, and neither should change the results of our algorithms. When we discuss a set of graphs, or an algorithm over graphs, we will be treating graphs as objects which denote structure, and will in every way be blind to their representation. When we refer to a graph, we are referring to all of its representations. When we intend to discuss a graph within some physical reality of its representation (for example, when determining whether two given adjacency matrices refer to the same graph) we will use the term *graph instance* to denote that difference. This is an 'algebraic' understanding of the structure of graphs, not a semantic choice. It will have important implications throughout this report, particularly around ideas of random graph models.

When we are discussing the set of all graphs as algebraic objects, we will use the notation G_{Alg}. When we are discussing the set of all graph instances, we will use the notation G_{Inst}. To start thinking critically about this distinction, always remember that:

$$|G_{Alg}| <<< |G_{Inst}| = 2^{E_{max}}$$

but since the number of representations of a given graph is limited by its number of labelings that:

$$|G_{Alg}| * N! > |G_{Inst}| = 2^{E_{max}}$$

Finally, we will talk about the number of distinct matrices that describe isomorphic graph instances as being the 'number of representations' of the graph, or $M_{Reps}(G) \in [1, N!]$. Note that $M_{Reps}(G)$ is related to the differentiation we made between the set of all graphs (G_{Alg}) and the set of all graph instances (G_{Inst}).

1.2.1 Graph6 Encoding of Graphs

For this project, we frequently will use graph6 notation to encode a graph as a succinct string. This is an encoding of the adjacency matrix as described through the formal specification (http://users.cecs.anu.edu.au/ bdm/data/-formats.txt). Basically, we take the upper half of the adjacency matrix (not including the zeroed-out diagonal), as a bit-string, then compress that bit-string using UTF-8 character encoding. I optimized this code (preallocation, bit shifting, etc) in all of the languages I used, because when non-optimized, it can contribute to runtime in a non-negligible way.

1.2.2 Counting Graphs over N Vertices

The number of undirected, non-looped graphs over N vertices is an open question in computational theory. The first few values of this sequence were well known, however no closed form has been proven to successfully list the numbers of graphs of a given size. A large number of theoreticians have come up with proposed closed forms with associated margins of error, but all small values have been agreed upon. The values in this sequence grow exponentially, but to give you an idea of how rapidly, the first twelve values are shown:

1 1 2 4 11 34 156 1044 12346 274668 12005168 1018997864

Thus, when we describe (as we will throughout this project) the computational difficulty of going, say, from calculating a metric over all graphs on 10 to 11 vertices, remember that is about a 60-fold increase in the size of the examined set, not even beginning to grapple with the time complexity increase that an increase in N has on the running time of the algorithm. The values that I use for this thesis come directly from the Online Encyclopedia of Integer Sequences, which gives values up to graphs with 50 vertices. As a side note: OEIS is probably one of my favorite things that exists. It astounds me how good it is picking out the exact sequence you are looking for, regardless of whether it is shifted by a few places, might have missing gap, etc. A well cooked example of the beautiful hyper-connected and data-driven world we live in.

1.2.3 Counting Graphs over N Vertices and M Edges

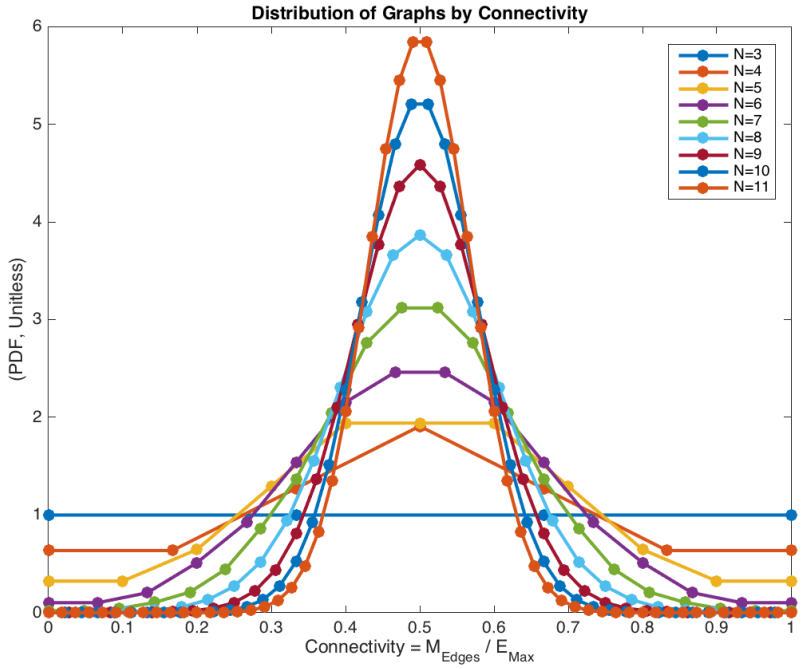
The number of graphs with a given number of nodes (N), has an internal distribution: the number of edges in each of the graphs. It turns out that this is an approximately normal distribution, which is (as we would anticipate) centered at $\frac{E_{Max}}{2}$. To see how this distribution changes over various values of N, we can normalize our notion of edges to one of connectivity: we will divide the number of edges by the number of possible edges (transforming an integer range $[0, E_{max}]$ to a fractional range $[0, 1]$.

This enables us to see how the distribution changes over multiple values of N. In figure 1.2.3 we make clear this connection by normalizing the distributions so that they are comparable (i.e. making the areas under each curve exactly one by dividing each by the total number of graphs associated with the value of N), we are able to capture the way that the distribution of graph connectivity changes for varying values of N and M.

When normalized in this manner, it becomes immediately clear that:

- Each of these distributions is normaly distributed

Figure 1.1: *Number of Graphs over N vertices with varying Connectivity*

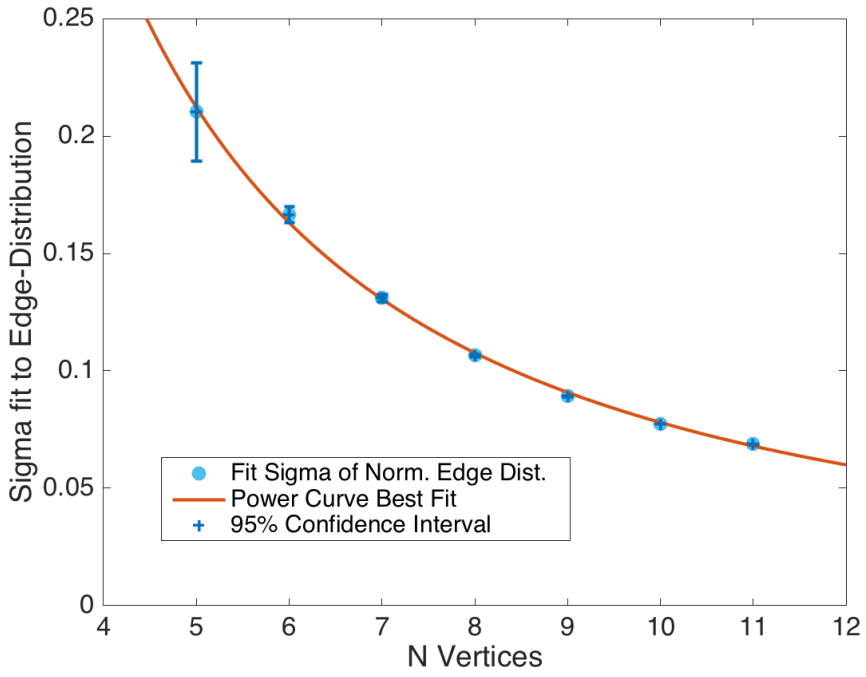


- The distributions share peaks (as we expect them to) at 1/2
- The standard deviation of each of these distributions is decreasing as N increases

To formalize these three observations, I first ran normality tests, which passed at the most strict levels possible for $N \geq 8$, and which were slightly less performant for smaller values of N (which makes sense, as we simply have fewer discrete bins on our x-axis in which to place observations, which are also many fewer in number). Secondly, I verified our thinking on the co-modality trivially: we know that if two graphs are non-isomorphic, then their inverses are non-isomorphic. Thus, our distribution must be symmetric about the half-connectivity point. Thirdly, we can formalize this by fitting a normal distribution to each of the observed relationships between population proportion and connectivity (allowing only sigma to vary) and come up with precise estimates for the normal fits for these distributions for $N \geq 5$. I opted not to use the fitted sigmas for $N < 5$, as the roughness of our data for those values meant that less than 90% of the variation in those observations was attributable to the normality. The results are shown below in figure 1.2.3, and there is a clear downward trend in sigma, which roughly follows a power curve which asymptotes to zero:

This establishes an important relationship: most graphs are 'somewhat

Figure 1.2: *Standard Deviation Fit to Distribution of Number of Edges for Graphs over N Vertices*



connected', neither sparse nor dense, with an increasing probability that they lie in the middle of the connectivity distribution. In fact, it appears that as N increases, sigma decreases, and we can conclude that the larger the value of N, the fewer and fewer graphs (as a proportion of the global set) are sparse or dense.

1.3 Graph Isomorphism and Automorphism

Two graph instances G and H are *isomorphic* if there exists a mapping M between $V(G)$ and $V(H)$ such that

$$\forall_{a,b \in V(G)}(a,b) \in E(G) \leftrightarrow (M(a), M(b)) \in E(H)$$

If an isomorphism exists between two graph instances, then the two instances represent the same graph; they have the same structure. An isomorphism preserves all adjacencies and all non-adjacencies, and the existence of an isomorphism between instances proves that they are the same graph. It may be possible for multiple isomorphisms to exist between two graph instances, but we are generally only concerned with the existence of such a mapping. We will use the notation $Iso(G,H)$ to be shorthand for a boolean predicate describing the existence of such a mapping.

The question of whether or not graph isomorphism as a decision problem (GI) can be computed in polynomial time is an open question in theoretical computer science. The advancement of Babai in November of 2015 proved that GI is computable in quasi-polynomial time, though no convincing arguments have placed it in NP-complete nor in P.

An *automorphism* is a mapping of the set of vertices of a graph onto itself ($V(G)$ to $V(G)$) which preserves adjacency and non-adjacency. If an automorphism M maps every element of $V(G)$ to itself, the automorphism is called the *trivial automorphism*. Though it will be taken as granted, the set of all valid automorphisms for a graph G forms a group. This group has at least one element (the identity element as the identity automorphism), but may have as many as $N!$ elements. This group will be referred to as $Aut(G)$, and the operation over the group is understood to be the *followed by* operation.

1.4 Graph Invariants

A *graph invariant* is an ordered property calculated over a graph which remains the same irrespective of the representation or labeling of the graph. More specifically, an algorithm or property is a graph invariant only if it produces output which is stable across all instances of the same graph. A graph invariant $Inv(G)$ can allow us to conclude that two graph instances (G_1, G_2) are *not* isomorphic if $Inv(G_1) \neq Inv(G_2)$. However, it is distinctly limited, in that the converse does not nescesarily hold (i.e. it is possible for non-isomorphic graph instances to share a value for a graph invariant).

1.4.1 Discriminatory Power

A graph invariant is *discriminating* if it can, with a certain probability, distinguish two non-isomorphic graphs as non-isomorphic. For example, an example of a graph invariant that is not very discriminatory is the vertex count of a graph. Two graphs are certainly not isomorphic if they differ in their vertex count, however, many graphs which are not isomorphic do have the same vertex count. In contrast, the chromatic polynomial of a graph is a highly discriminatory graph invariant, as the odds of having two non-isomorphic graph instances agree on their chromatic polynomial is relatively low.

To formalize this notion, we will discuss discriminatory power with a specific probabilistic meaning. A graph invariant Inv discriminates at a level α for N vertices and M edges if selecting two graphs G and H at random from some random graph generator:

$$P\big[Inv(G) = Inv(H) \wedge \neg Iso(G, H)\big] \leq \alpha$$

What we will find is that we can frequently discuss alpha as a function of M and N. Later in this report we will discuss how α fits in to a natural definition of a false positive an uncertain test without a false negative rate ($\beta = 0$).

1.5 Vertex Invariants

1.5.1 Vertex Similarity

Two vertices are *similar* if there exists a mapping in the automorphism group $Aut(G)$ such that the mapping maps one vertex to the other. Similarity is a transitive and commutative property. The vertex set $V(G)$ can be divided up into between 1 and N similar vertex sets, such that all of the vertices in each set are similar, and no two sets contains similar vertices. A discussion of these *similar vertex sets* (or SVSs) will be the primary focus of chapter three.

A graph (or subset of the vertices of a graph) is called *perfectly similar* or *perfectly automorphic* if, for every pair of vertices, there exists an automorphic mapping which maps one of the vertices to the other. This is not suggesting that every mapping of the graph is an automorphism (as is only the case in K_n and $\overline{K_n}$), but rather that any initial choice of pairing within a mapping is valid, even if it limits further choices. For example, C_n is a perfectly similar graph, as is Peterson's graph.

1.5.2 Vertex Invariants

Vertex invariants are numerical properties that we calculate over a specific vertex within a graph which identifies potentially similar vertex pairs. Similar vertices within a graph agree on all vertex invariants. However, like graph invariants, vertex invariants can only eliminate the possibility for vertex similarity, they are not sufficient to prove similarity.

Vertex invariants make the computation of graph isomorphism between two graphs markedly easier. Whereas a graph invariant can tell us about whether or not graph instances as a whole might be alike, it does nothing to suggest a proposed mapping between the vertices of the two graph instances. In contrast, a vertex invariant identifies potentially similar vertices not only within a graph, but also between graph instances. A *perfect* vertex invariant is one for which agreement on the value of the invariant is equivalent to establishing the existence of an automorphism that maps one vertex to the other.

A question discussed later in this report will be about the theoretical implications of a hypothetical perfectly discriminatory vertex invariant, and a proposed invariant that haven't been found to be imperfect.

1.6 Cycles Invariant

The focus of this report is an invariant which can function as an invariant over graphs or their vertices. It is called the 'Cycles' invariant, but is sometimes referred to as the 'Paths' Invariant in cases where cycle has other connotations. In either case, we will consistently capitalize to distinguish the invariant from its other denotations.

Our definition of cycles counts ones which repeat vertices and edges, and ones which can pass back through their place of origin. We are counting cycles which are directional, so $ABCA$ and $ACBA$ are distinct cycles.

Note that cycles (as a function and vertex invariant) is a property of a graph instance, only through sorting can we make it into a graph invariant.

1.6.1 Cycles as a Function

The $Cycles(A, p, v)$ invariant counts the number of closed paths of length p that pass through a given vertex, v. This information can be easily computed using A. Just as the entries of A^1 represent the existence of paths of length 1 between two vertices (edges), the entries of A^p represent the number of paths of length p between any two vertices (by examining the row and column corresponding to two vertices). Thus, to find the number of closed paths of length p that contain a given vertex v, we simply need to calculate:

$$Cycles(A, p, v) = A^p[v, v]$$

Where v is being used interchangeably here with its represented position within the adjacency matrix. Thus, calculating a specific value of $Cycles(A, p, v)$ can occur in the time it takes to exponentiate A to the power p.

Though it is a well known result that matrix multiplication can be done in faster than $O(n^3)$ time, we will be using the naïve assumption that matrix multiplication runs in $O(n^3)$ in order to make the computational complexity calculations more accessible. Under that simplifying assumption, it is clear that calculating $Cycles(A, p, v)$ will occur in $O(pN^3)$ time, but we can request as many values of v and $p_i < p$ 'for free' after a single calculation for A and p.

1.6.2 Cycles as a Vertex Invariant

Cycles as a vertex invariant describes a vector of length P, where the pth entry is the number of closed cycles of length p which pass through the vertex being described.

If a vertex is the ith row/column of an adjacency matrix A, then the cycles invariant for the vertex is the successive values of

$$Cycles(A, v_i) = [c_1, c_2, ...c_N] \text{such that} c_p = Cycles(A, v_i, p) = A^p[i, i]$$

for each of the values of p, forming a vector of length P.

The vector generated by this computation is a way of describing the local graph around the vertex v_i. We can consider many ways in which Cycles reflects a 'reverberation' about the local neighborhood of a vertex, and provides a noisy invariant, which is useful for distinguishing between vertices in a graph. It will be discussed why later, but we will prove that P will always be strictly less than N, and that further values of computation are not useful, as they cannot provide additional layers of differentiation between graphs or their vertices.

1.6.3 Cycles as a Graph Invariant

Extending this vertex invariant to be a graph invariant requires little imagination. We simply calculate the Cycles vertex invariant for every one of the vertices of the graph, and are given back N vectors of length P. We then sort the resultant vectors lexicographically, and arrange them in a $N \times P$ matrix. This matrix is a comparable object which is invariant to changes in labeling or representation of G.

Thus, paths as a graph invariant takes fewer inputs: Cycles(A, P).

Since vectors are comparable in linear time, we can sort them in log-quadratic time. This does not change the asymptotic nature of our running time.

Cycles(A, P) is invariant to changes in vertex relabeling or adjacency matrix ordering. A skeptical reader should convince themselves that this holds true, even when two of the vectors to be sorted are identical, the resulting *Paths* object is valid and deterministically constructed from the graph. The specific running time of calculating and comparing the *Cycles* functions and object is not the primary aim of this report, but there are clever methodologies that I have used to reduce the running time beyond these naïve estimates. Most notably, we can significantly improve upon the average case time complexity by using delayed evaluation techniques, and can improve overall running time by multi-threading the calculations across multiple cores or GPUs.

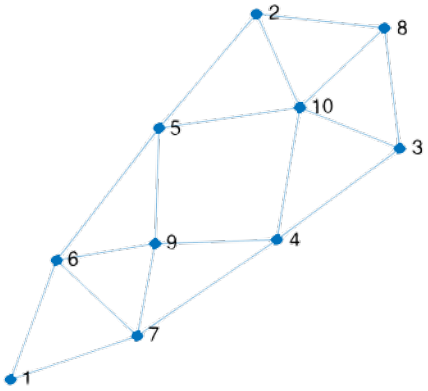
1.6.4 A Brief Example

The best way to establish intuition and understanding of the cycles invariant is to show its values on a sample graph. In figure 1.6.4, we see an example graph (in graph6 format: I@QKj?]]O), which has multiple automorphisms. The graph has been arranged to be aesthetically appealing and to make the automorphisms clear.

In figure 1.6.4 below, we also see the cycles matrix for the graph. The rows represent the individual vertex invariants associated with each value of cycles. The columns represent the length of the cycles being considered. The

reader should trace out the number of cycles for themselves (for small values of p) to see the connection, and understand how this function is translated to a vertex invariant, and then to a graph invariant (the full matrix).

Figure 1.3: *A Small Graph* - A labeling of the vertices is provided to correspond to the labeling of figure 1.6.4. This graph (like all of the ones we are describing) is unlabeled, undirected, and loop-less.



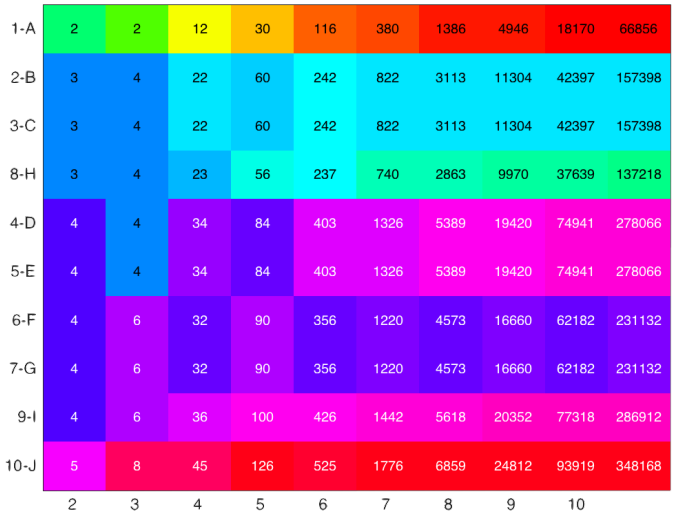
1.6.5 Invertibility

Note that if two graphs agree on the Cycles invariant, their complements (or inverses) likely agree on the Cycles invariant, but there is no proven reason to believe this is universally true. The existence of non-isomorphic, co-cycles graphs raises the question: are their inverses necessarily co-cycles? For all co-cycles graphs that were found, invertibility holds, but that is not a proof that it always holds.

1.6.6 Time Complexity

The running time of the paths invariant is the same whether we treat it as a vertex invariant or as a graph invariant. The one difference is whether or not we sort the resulting vectors. It is imperative to calculate A^P even if we are only interested in calculating it for a single vertex. Matrix multiplication can generally be accomplished in sub-cubic time CITE, but for the sake of simplicity and comprehensibility within this paper we will assume that it is an algorithm that runs in $O(N^3)$ time. This means that the computation of any paths invariant can be accomplished in $O(N^4)$ time. Though this sounds like a lot of time, in practice this is a fast computation. It is made asymptotically better by the fact that efficient algorithms for matrix multiplication are well studied and optimized for a variety of contexts. Matrix

Figure 1.4: *A Cycles Invariant Matrix* - The color coding corresponds to the numbers, and purely serve to give a visual differentiation between different integer values. We can see that many of the pairs of vertices in this graph share their values of the cycles invariant through them.



multiplication can be done efficiently by GPU arrays, and there are quick algorithms to do multiplication on sparse matrices for larger graphs.

1.6.7 Operating over Large Numbers

One element of the Cycles invariant that we will consistently need to be cognizant of is the fact that our numbers will grow very quickly, particularly for large values of N. This complicates the calculation of cycles for values as small as $N = 10$. Integer overflow occurs when the summed value of the elements in the dot product add to a number greater than 2^{32} (on our system). We can avoid this overflow if we guarantee that each of the N pieces of the summation are less than or equal to $2^{\lfloor 32-\log_2(N)\rfloor}$. Since we arrive at each element in this dot-product-summation via the product of two elements that were previously somewhere in our running matrix, we can avoid an overflow in a computation step if we have:

$$\forall_{i,j\in[1,N]} A[i,j] \leq \sqrt{2^{\lfloor 32-\log_2(N)\rfloor}} = 2^{\lfloor 16-0.5\log_2(N)\rfloor}$$

We can make sure that this equality holds if we establish $K = \lfloor 16 - 0.5\log_2(N)\rfloor$ and make the incorporation of modulo into our formula for the Cycles function explicit:

$$Cycles(A, v_i, p) = \begin{cases} A[i,i] & p = 1 \\ (Cycles(A, v_i, p-1) * A) \ \% \ 2^K & p > 1 \end{cases}$$

This preserves the properties of addition and composition that we are looking for, and maintains reasonably sized bit arrays. If we examine this methodology, we see that this operation is stable (if A and B map to the same value, they still will in this system) and that the odds of 'collisions' (where values of A and B differ in the old system, and are the same in the new system) are unlikely. The first claim is verified through the fact that if we break any integer z_i into the portion of it divisible by 2^K (x_i), and the piece that is the remainder (y_i), then our properties of multiplication and addition hold under the modulo transformation (denoted $T_K(z_i)$) is stable/consistent:

$$z_i = x_i(2^K) + y_i, \quad T_K(z_i) = y_i$$

$$z_i + z_j = (x_i + x_j)(2^K) + (y_i + y_j)$$
$$T_K(z_i + z_j) = 0 + T_K(y_i + y_j)$$

$$z_i * z_j = (x_i * x_j)2^{2K} + (x_i * y_j + x_j * y_i)2^K + (y_i * y_j)$$
$$T_K(z_i * z_j) = 0 + 0 + T_K(y_i * y_j)$$

The second claim is verified through a thought experiment about how cycles varies over multiple values of p. If we are attempting to distinguish between to cycles vectors, the most obvious comparison would be the degree of the vertices (the second element of a cycles vector). If two vertices don't agree on this value, we have distinguished them successfully, and any future point of overlap (or collision) is irrelevant. If they agree on this value, then we know that each will have some minimum number of paths for any even value of p (as we know enough about their local neighborhood to assume the star graph as a minimum). Similarly, past values within the cycles invariant can be used to determine a large percentage of future values of the cycles invariant, for larger values of p. The majority of the cycles described by the invariant are products of smaller cycles, so the 'rate' at which they are added is highly correlated to past events. To have a large number of newly emergent cycles in one but not the other (a precise enough one to exactly match the threshold for modulo) is incredibly unlikely.

Consider this argument by an analogy, one used to catch cheaters (people who cut courses) in looped Marathons and long track races. If two people are in a race, and their first many splits are at the same times, it is very unlikely (even impossible) that one will lap the other over the course of a single split. The same logic can be applied to think about the way that the number of closed cycles grows as p grows.

1.6.8 Space Complexity

Though the above simplification can certainly allow us to do our computations in a way that is likely to avoid collisions, if we are discussing properties of the invariant as a whole, our analysis ignores the nature of collisions. Thus, if we are attempting to draw conclusions about the algorithm from a theoretical perspective, we need to assume that we are fully calculating the cycles invariant without the use of the modulo trick. This means that we need to prove that computation of it can be done in polynomial space. Otherwise, the polynomial aspect of this computation could be misleading away from a gross inefficiency in space that masks the the true costs and limitation of the computation. For example, there is a cute algorithm that I designed, which solves the boolean satisfiability problem (an NPC-problem) in linear time, using only addition, multiplication and bitwise operations, but uses exponential space to do it. The flaw in this kind of algorithm is that it exploits a simplification that we make in algorithmic theory: we don't consider the time complexity on a bit-by-bit basis. With large enough numbers, algebraic operations are not constant, they are linear functions of the number of their bits. Infinitely sized registers are an abstraction of well designed systems, but cannot exist. Thus it is important for us to verify that the calculation of the cycles invariant uses a polynomial amount of space.

In the worst case, the largest value in the Cycles invariant will occur when we have a fully connected graph of size N. In a fully connected graph, any sequence of vertices that does not put the same vertex adjacent to itself is a valid cycle. Thus, for a length l, there are exactly $(N-1)^l$ valid paths, of which exactly $(N-1)^{(l-1)}$ of which start and end at the same location (and thus are cycles).

If we assume that the maximum length cycle we are interested in is of length N-1 (as is discussed in another chapter), then the number bits required to express an individual number within the Cycles matrix is at maximum:

$$O(Bits(N)) = O\Big(\log_2\left[(N-1)^{N-1}\right]\Big) = O\big((N-1)*\log_2[N-1]\big) = O(N\log_2 N)$$

Thus the total number of bits required to fully represent a Cycles matrix is bounded by $O(N^3 logN)$, a large upper bound, but certainly sub-exponential.

1.7 Reconstructability, Determination

Many of the graph theoretic discussions of the cycles invariant will describe it with respect to other invariants. We have some language to assist these comparisons. We will say that a property of a graph is 'reconstructable' from Cycles if we can construct a valid value for the property if we are given the

Cycles invariant of the graph. Similarly, a property is determined by Cycles if a given value of the invariant allows exactly zero or one values for the property in question. Reconstructability and determinability differ only in that reconstructability describes a procedure for the conversion, but doesn't guarantee a unique result, while determinability demands that a value for Cycles uniquely determines the property in question, but doesn't require us to show a technique by which to perform the determination.

Chapter 2

Cycles as a Graph Invariant

Cycles is a powerful graph invariant. In this chapter we will discuss properties of graphs that are reconstructable from Cycles. We will then show how Cycles is reconstructible from some other graph invariants, which leads us to the conclusion that Cycles is necessarily an incomplete invariant. We will then discuss manual calculations that find examples of Co-Cycles graphs, and the establishment of small datasets of Co-Cycles graphs as a measure by which to gauge the discriminatory power of graph invariants. Finally, we will examine the performance of the cycles invariant on different classes of graphs which typically show resistance to classification and differentiation via graph invariants.

2.1 Basic Cycles-Reconstructable Properties

From the Cycles graph invariant, we can easily deduce the number of vertices (the size of the resulting matrix's first dimension), and the number of edges (the sum of the second column of this matrix, divided by two). We can also deduce the degree sequence, as simply observing the second column of each vertex invariant vector, and sorting the result.

2.1.1 Triangles, Higher Order Polygons

Within the context of a graph, a polygon is similar to a cycle in that both denote closed paths, however, a polygon differs from a cycle in that a cycle is allowed to repeat both edges and vertices, while polygons do not allow repetitions of either (except to close the path to its starting vertex).

The number of triangles is also easily computed. Since triangles necessarily contain three distinct vertices (in a graph with no self-loops), we know that any cycle of length three on a graph will be a triangle. Thus, the number of triangles which pass through each vertex is simply the third column of the cycles invariant matrix, and summing them and dividing by

three yields the total number of triangles in the graph.

Things are not so simple for larger polygons. If we think very critically, we can deduce the number of of valid quadrilaterals, by considering the degree sequence of each of the nodes adjacent to a specific node.

Note that this logic requires us to take in an additional piece of information to augment the data that we get from the cycles invariant (i.e. the sum of the degrees of the adjacent nodes). Similarly, figuring out higher order polygons can be done, but it requires a larger amount of external information. Generally, the larger the 'neighbors-paths' supplemental information we require, the further we stray toward giving away the information that fully determines the graph.

However, this is a powerful observation, and fits into our idea of the cycles invariant as a 'neighborhood awareness mechanism'. A formal discussion of such mechanism is discussed in chapter three.

2.1.2 Chromatic Polynomial

One of the most discriminatory polynomial time algorithms for ruling out graph isomorphism is the *Chromatic Polynomial* of the graph's adjacency matrix A. It is a well proven result that shuffling the order of the vertices in the representation of a given graph as an adjacency matrix does not change the eigenvalues of the matrix (thus the eigenvalues, as a multiset, form a powerfully discriminatory graph invariant), and thus the chromatic polynomial (as it encodes all information from this multiset) is also a graph invariant. In practice, it is a highly discerning as a graph invariant, as a very small proportion of graph instances are "co-spectral" (having the same chromatic polynomial) and not isomorphic. However, there are not well established procedures to use the results of eigenvalue calculations to attempt to explicitly construct an isomorphism, rather, it is powerful in ruling out hypothetical isomorphisms.

The large downside of the chromatic polynomial (as a graph invariant) is that it is not computable in polynomial time (across all sets of graphs). Rather, it is in the computational class $\#P - Hard$ (Sharp-Polynomial Hard), by reducibility to #3SAT. There are algorithms which can deterministically solve it in $O(2^N N^r)$ For a constant r. This makes is calculation for graph isomorphism testing prohibitively expensive. What we will show in this section is objectively quite valuable: we will show that the Cycles invariant encodes the chromatic polynomial, and is actually *more* discriminatory than it. Thus, by calculating the Cycles invariant, we can discriminate between graphs with greater accuracy, in less time, than by using the chromatic polynomial. This result is significant, but requires a long proof.

In seeking out a relationship between the chromatic polynomial and the *Cycles* invariant, it helps to remember the helpful property of eigenvalues:

$$E(A) = \{\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_N\} \rightarrow E(A^P) = \{\lambda_1^P, \lambda_2^P, \lambda_3^P, \ldots, \lambda_N^P\}$$

Additionally, since we know that our original matrix A is diagonalizable, positive, and symmetric, we know that all of the eigenvalues are real and non-negative. Finally, we know that the sum of the diagonal of any matrix is the sum of its eigenvectors. Thus, using the $Cycles$ function to determine the entries of the diagonal, we know that for any adjacency matrix A,

$$\lambda_1 + \lambda_2 + \lambda_3 + \cdots + \lambda_N = \sum_{i=0}^{N} Cycles(A, 1, i)$$

Or more generally (leveraging our knowledge about the eigenvalues of A^P:

$$\forall_{p\in[1,P]} \sum_{i=0}^{N} \lambda_i^p = \sum_{i=0}^{N} Cycles(A, p, i)$$

This means that we can create N nonparallel (as not divisible in the ring $R[x]$) equations, each containing λ_1 to λ_N as variables. Additionally, since we have the guarantee that our eigenvalues are all non-negative, we know that we can assuredly solve any given equation in terms of one of the λ's. If we then use these substitutions, and substitute out all but one of the variables, what we will get is a continuous function whose zeros are well defined and positive.

Given the aggregated substitution, each one of the zeros is an eigenvalue, assign a selected value to any unassigned eigenvalue, and reduce our equations using that assumed value (think of a let statment). This kind of procedure is necessary because our system of equations has variables whose relationships are inherently symmetric. In other words, we cannot make any differentiation between λ_1 and λ_2 based on the relationships of the equations, but if we make a choice for either, it will lead to a valid conclusion, or a reduction which does not eliminate a valid conclusion. This is by virtue of the fact that our substitutions are without condition, and we don't need to worry about any domains of substitution.

From a physical perspective, these equations can be described as surfaces in V dimensions, whose intersection we are interested in isolating for. Graphing the equation $x^3 + y^3 + z^3 = \kappa_3$ yields an interesting quasi planar sheet which becomes highly regular at large scales. The equation $x^2+y^2+z^2 = \kappa_2$ describes a sphere with the radius $\sqrt{\kappa_2}$, and the equation $x + y + z = \kappa_1$ describes a plane. Each of these functions is axis symmetric, so their intersections are also axis symmetric. The intersection of all of the equations yields a set of points (or none), which represent the potential values for our λ's, which are axis invariant. We can generalize this to any number of requisite dimensions using the principle of equation and variable symmetry that these equations necessarily produce.

Thus, given the function $Cycles(A, p, v)$, we could calculate the eigenvalues of the matrix A using $Cycles(A, p, v)$ restricted to $1 \leq p \leq P = N$. This

is superb, as it shows that the *Cycles* function encodes at least as much information as the chromatic polynomial, and that any graphs which produce the same *Cycles* function up to (p = V) are co-spectral. A quick verification of the base non-isomorphic co-spectral graphs (the butterfly and the box) show that they produce *different* values of the Cycles function, showing that the *Cycles* function carries *more* information than graph spectrum does, even though both are computed in polynomial time. Also helpful: while co-spectral tests might rule out isomorphisms instead of providing a mechanism for their potential generation, the *Cycles* function suggests the isomorphism that it believes to exist with a natural sorting and comparison of the vectors within the *Cycles* object.

Again, because of the difference in the running time required to calculate Cycles ($O(N^4)$) versus the running time it takes to calculate the chromatic polynomial ($O(2^N N^r)$), this is an incredibly valuable result: rather than using the chromatic polynomial for discerning between non-isomorphic graphs, it is clear that the Cycles invariant is objectively better.

2.1.3 Reconstructing a Valid Graph from Cycles

If we are given full access to the *Cycles* function, could we create a graph that is in line with the Cycles invariant? Note that this question is fundamentally distinct from the one which asks if we could reconstruct the *only* graph that could generate those Cycles values. That is a separate question, one which the existence of co-cycles graphs makes irrelevant, but the question posed is significantly more general, and does not decide or influence the question of computational complexity of GI. The reader should note that this section is really a proof of concept and the work done for this was to see how we could go between modes of operation: between reasoning about graphs theoretically, and interacting with them on an algebraic level. A reader who is focused only on interesting results will not find them in this section.

The answer to this question is yes, given a Cycles invariant, we can reconstruct a graph that creates it. Over the next few paragraphs, I am going to detail a method of constructing such a graph by first constructing constituent integer-valued equations which describe the interrelationships between edges, then transform the integer edge equations into equally information dense boolean sentences. The final step is then to transform each of these sentences into CNF (Codd Normal Form), and use a satisfiability solver to find a solution to them.

It should be noted that a brute force search for a fitting graph would likely run in faster time, and that the final algorithmic step here (3-SAT) is in NP-Complete (thus has a non-polynomial running time). The distinction here lies in the fact that our algorithm produces a predicate which describes *all* graphs which fit the given Cycles invariant. The transformation to 3-SAT involves the parsing out of all invalid cases, so that any value that

we produce is guaranteed to satisfy the given Cycles invariant. Contrast this to a guess and check methodology. In this section we are not doing any guessing, nor do we need to do checking. Though both algorithms run in exponential time, the sophisticated one is employing a fundamentally different unit of computation.

Before we begin, lets examine three important points:

- Our reconstruction algorithm is going to run in exponential time, and *that is okay.* We have already demonstrated that our invariant, the $Cycles$ function, is calculable in polynomial time. The fact that we can reconstruct a valid graph from the $Cycles$ function is about exploring or debunking the invertible nature of the $Cycles$ object, and the time it takes to perform an inverse of that operation tells us nothing about the computational complexity of that operation.

- The equations in this section for all but the most trivial of graphs take up a large amount of space, and as such, it is not recommended that this reconstruction technique be used in practice. Realistically, a brute force search would likely yield faster running-solutions (if one were trying to find a graph fitting the Cycles function), but the reader should try to convince themselves that this approach would terminate, and would yield a valid solution, given a valid $Cycles$ object.

We know that the diagonals of A^p yield the $Cycles$ function. We also know that A is comprised of $E_{Max} = \frac{N*(N-1)}{2}$ boolean variables, and all entries of A^p must be some combination of the variables in the original adjacency matrix. We will refer to these variables as the x_i's. Generally, we will arrange them in the row primary pattern, and will number them starting at 1.

The x_i's also have the helpful property that $\forall k \geq 1, x_i^k = x_i$. This stems from the fact that each one of the x_i's is either zero or one, the two solutions to this identity. This allows us to reduce any polynomial degree in our resulting equations down to one, though multivariate linear terms may remain.

I have been discussing these 'equations' quite a bit; lets formally define them. We assume that we are given the $Cycles$ function for a graph, and we will call $Cycles(A, p, v) = k_{p,v}$ to simplify our work. In each equation, we will set the Cycles function for a specific v and p equal to the symbolic representation of the exponentiated adjacency matrix (which will solely be in terms expressed by the x_i's). We will refer to this specific equation as Equation p.v for v in the range $[1, N]$ and p in the range $[1, \infty)$. For each v and p in their respective ranges, our equation p.v is:

$$k_{p,v} = A^p[v, v]$$

Some example Equations are shown below for a five node graph (2.1, 3.1, 4.1). Note the rapid expansion in the number and complexity of the terms. Also note that we don't need any polynomials over one variable, so we have collapsed them down to their reduced terms.

$$k_{2,1} = x_1 + x_2 + x_3 + x_4$$

$$k_{3,1} = 2x_1x_2x_5 + 2x_1x_3x_6 + 2x_1x_4x_7 + 2x_2x_3x_8 + 2x_2x_4x_9 + 2x_3x_4x_{10}$$

$$\begin{aligned} k_{4,1} = x_1 + x_2 + x_3 + x_4 + 2x_1x_2 + 2x_1x_3 + 2x_1x_4 + 2x_2x_3 + x_1x_5 + 2x_2x_4 + x_1x_6 + x_2x_5 \\ +2x_3x_4 + x_1x_7 + x_3x_6 + x_2x_8 + x_2x_9 + x_3x_8 + x_4x_7 + x_3x_{10} + x_4x_9 + x_4x_{10} \\ +2x_2x_3x_5x_6 + 2x_1x_2x_6x_8 + 2x_1x_3x_5x_8 + 2x_2x_4x_5x_7 + 2x_1x_2x_7x_9 + 2x_1x_4x_5x_9 \\ +2x_3x_4x_6x_7 + 2x_1x_3x_7x_{10} + 2x_1x_4x_6x_{10} + 2x_2x_3x_9x_{10} + 2x_2x_4x_8x_{10} + 2x_3x_4x_8x_9 \end{aligned} \tag{2.1}$$

An interesting aspect of these equations actually has a cool natural cause. Notice that in equation 2.1 and equation 4.1, both have linear terms of four variables. (Those happen to be the four variables in the row that we chose, row 1, but if we chose row 4, we would have gotten the variables in that row). A nice property that arises out of these linear variables is that if we were to solve for the variable x_1 from equation 4.1, we would get an expression with the following denominator:

$$2x_2+2x_3+2x_4+x_5+x_6+x_7+2x_2x_6x_8+2x_3x_5x_8+2x_2x_7x_9+2x_4x_5x_9+2x_3x_7x_{10}+2x_4x_6x_{10}+1$$

This is of particular interest, because we know that each one of the terms in this statement has to be *non-negative*, as $\forall i \in v, x_i \in \{0, 1\}$. Thus, there is no possible valid input of x_i's which results in this denominator being zero, and our substitution is thus universally valid. That is really important because it means that we might be able to do the same thing for other variables, and potentially come up with a system of equations that fully describes the interactions of the vertices within our graph. Thus such a system could uniquely determine each of the x_i's, the graph could be determined by *Cycles*, but what what we find instead is that *a* reconstruction exists, but it is also possible that the reconstruction is not unique.

Lets explore this notion further. Lets say that we have an equation that describes the interactions of the binary relations in such a way that we can express the equality for of the variable x_i solely in terms of the other variables such that

$$x_i = \frac{C}{D + z}$$

Where z is an integer, $z \geq 1$, and C and D can be any number of positive expressions that are comprised of terms of the addition and multiplication of

variables in the set $\{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{(N(N-1))/2}\}$. Since we know that $x_i \in 0, 1$, we know that any positive term comprised of the multiplication and addition of these variables must be greater than or equal to zero. Thus, we know the denominator of the equation for x_i must be greater than or equal to one ($D + z \geq 1$). This is a helpful result, as it means that our substitution is valid, and we know that the equality holds, because our division of both sides by $D + z$ does not risk dividing by zero. We will call this a *valid substitution* for x_i over the equation for which it is generated (e.g. Equation 2.5).

It turns out that there is a clever way to use a *valid substitution* to construct a graph which would generate the Cycles functions that generated the equations which generated the valid substitutions. Since we know that the denominator of a valid substitution is non-zero, we know the substitution is valid. We also happen to know that the variable we are solving for is either equal to zero or one. Thus, if the numerator (C) of our substitution is equal to zero, then we know that x_i is equal to zero. Otherwise, we know that x_i must be equal to one (for all other values are not in its domain). If we allow ourselves to do an informal conversion to predicate logic, we can transform an equation of the form:

$$x_i = \frac{C}{D + z}$$

$$(x_i = 0) \text{ iff } (C = 0)$$

$$\neg(x_i) \text{ iff } (C = 0)$$

$$x_i \leftrightarrow \neg(C = 0)$$

Expressing that C, a summation of terms over the x_i's, is equal to zero, is actually quite a simple construct. We imagine that each of the terms in C has a positive, integer valued weight that corresponds to its coefficient. If C contains any integer valued negatives (which it does in real applications), we will call this the *target*. Given this (and the terms and their weights as corresponding lists), we can generate a simple procedure for generating a boolean statement that encodes the information that our equation does when C is set to zero.

```
exactlyKTrue(clauses, weights, target):
        if (target == 0) then
                return negatedConjuction(clauses)
        clause = clauses.pop()
        weight = weights.pop()
        caseF = exactlyKTrue(clauses, weights, target)
        newTarget = target - weight
        caseT = exactlyKTrue(clauses, weights, newTarget)
        return (clause & caseT) | (!clause & caseF))
```

Though this code doesn't totally cover all edge cases, it should convince the reader that there is an appropriate transform between C and a boolean expression of the x_i's which maintains domain validity across substitutions.

2.2 Other Forms of Reconstructability

2.2.1 EA Reconstructability

Just as Cycles can be used to reconstruct other properties of a graph, Cycles can be reconstructed from another invariant, the matrix describing the angles between the eignespaces of a graph and the standard basis of R^n. In their paper *A Study of Eigenspaces of Graphs*, D. Cvetković, P. Rowlinson, and S. Simić showed that the cycles invariant (which they described as the number of closed walks) can be reconstructed from the angle matrix of G. Just like the Cycles invariant, the angle matrix becomes a graph invariant when we lexicographically order its vectors.

However, the angle matrix of a graph is much more difficult to calculate than the Cycles function. As a prerequisite to describing the angles, we first need to know the eigenvalues of the graph, which as we have already discussed, is an exponential algorithm. If we think about the time-discrimination tradeoff, the angle matrix sits at a location that is more discriminatory and more costly than the Cycles invariant.

2.2.2 Deck Reconstructability

An open question that I have not been able to resolve is whether or not we can reconstruct the cycles invariant from the deck of a graph. A deck of a graph G over N vertices, $Deck(G)$ is a multi-set of N graphs such that each graph in the deck (called a card) is one of the vertex deleted subgraphs of G. The reconstruction hypothesis claims that a deck uniquely determines a graph.

Many graph invariants can be proven to be reconstructible from the deck of a graph. I was not able to prove that the Cycles invariant is reconstructible from a deck, but, if you look to some of the unfinished work in chapter six, there is a lengthy discussion about how the cycles invariant might be able to give us tools to solve the reconstruction hypothesis by setting up clever predicates about each of the card graphs and their interrelationships.

2.3 Placing Cycles within a Time/Power Tradeoff

A subjective task we have to undertake is the critical examination of Cycles within the invariant tradeoff between computational power and time complexity. On the one hand, cycles is highly discriminatory, and for the

majority of graphs, we can show that cycles fully distinguishes between different graphs. This computational power means that for the overwhelming majority of graphs (particularly large graphs), cycles can give us canonical labelings with a single pass.

However, on the other hand, cycles is heavy handed. There is a quartic amount of work that has to be done to calculate cycles for the number of vertices in a graph. This means that even if we had a graph where every node was distinguished only by its degree, we would still require an extensive process of matrix multiplication to even get to results that could have been achieved in less time by any other means.

2.4 Setting up a Computational Description

One way we can discuss this tradeoff is computationally. An invariant that is 'good' distinguishes between non-isomorphic graphs in a small amount of time, and fails to do so with a relatively low frequency Thus, we can evaluate an invariant in the following way.

An invariant, when given two graphs G and H, calculates its value of G and H, then returns 'accept' if the invariant values are the same, and 'reject' if the invariant values are not the same. We are going to assume that the invariant is an imperfect distinguisher: meaning that the probability of failing to distinguish between two graphs is some constant α, corresponding to a false positive rate. Note that the nature of an invariant means that β, a false negative rate, is inherently zero.

The way this tradeoff is going to be executed on is simple: we will generate a large set of paired graphs, some of which are isomorphic pairs, and some of which are not. We will calculate how well the invariant does in terms of its alpha rate, and also calculate the amount of time that it takes to complete the task of deciding 'accept/reject' on the full set.

This will enable us to show this tradeoff in a direct, graphical, way.

[IF TIME DO THIS]

2.5 Discrimination on Tough Graph Classes

2.5.1 Miyizaki Graphs

The Miyizaki Graphs are as interesting as they are inscrutable. They are best described by Gamkrelidze, Hotz and Varamashvili in their paper *New Invariants for the Graph Isomorphism Problem*, and can be summarized as being comprised of similar constructions that are occasionally twisted. The diagrams below are borrowed from their paper.

It does not come as a surprise to me, and it should not come as a surprise to you, that the cycles invariant is unable to distinguish in between any

Figure 2.1: *Internal Structure of Miyizaki Graphs*

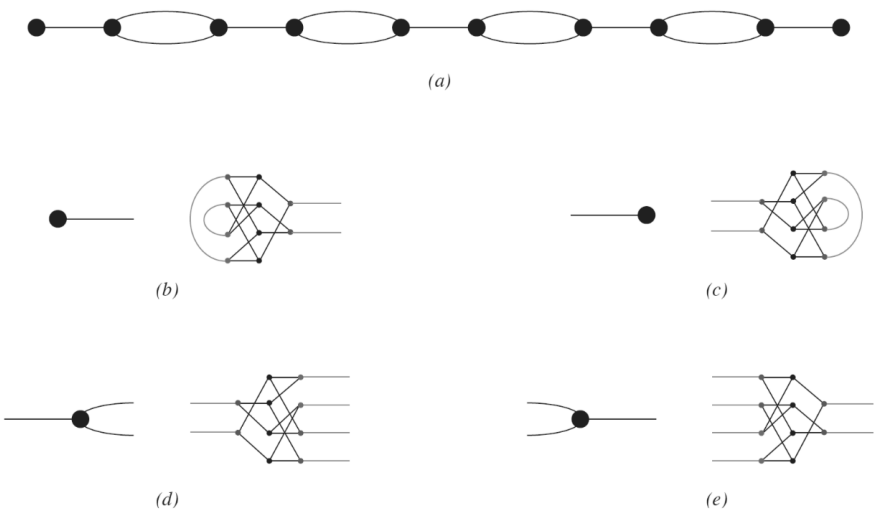
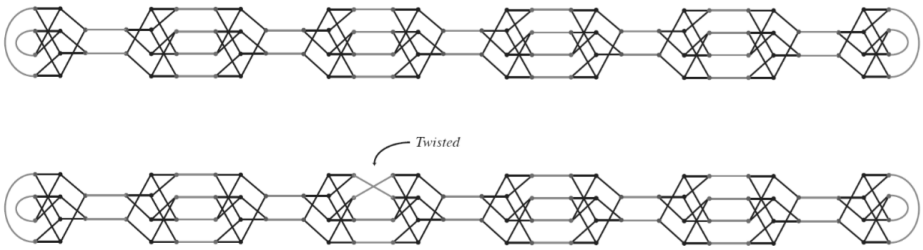


Figure 2.2: *A Twisted Miyizaki Graph (M_4vs.$MT_{4,2}$)*



Miyizaki Isomorphism pairs.

Miyizaki graphs are co-cycles graphs in partnership with their twisted halves. In testing this hypothesis, I tested it on the Miyizaki graphs:

- M_1 versus $MT_{1,1}$
- M_2 versus $MT_{2,1}$
- M_2 versus $MT_{2,2}$
- M_3 versus $MT_{3,1}$
- M_3 versus $MT_{3,2}$
- M_3 versus $MT_{3,3}$

and each were shown to be co-cycles with their partner.

However, differentiation with flagging (as described in the chapter on vertex invariants) was possible and did successfully discriminate between the structures internally (see Cycles as a Vertex Invariant), at enormous computational (though still polynomial) expense.

2.6 Imperfection, Co-Cycles Graphs

Though it has been alluded to previously, a key discovery of my thesis is that there exist co-cycles graphs: non-isomorphic graphs which have the same value for the Cycles invariant for all observed values of P. These graphs are very rare. None exist with fewer than ten vertices, and only about 100 of the 12 million graphs over 10 vertices satisfy this property.

In this section, we will discuss the initial discovery of co-cycles graphs, detailing the relatively advanced set of methodologies used. Then we will discuss how co-cycles graphs can be constructed intentionally on larger sets of vertices. Finally, we will discuss co-cycles graphs as a useful (and relatively small) set of graphs which are useful for invariant analysis and testing.

2.6.1 Discovering Co-Cycles Graphs, Cleverly

Discovering co-cycles graphs was not trivial. The way I went about looking for them was theoretically simple, but incredibly challenging to implement and process.

- First, I enumerated all graphs of size N with M vertices by using the NAUTY geng function
- I set up a mechanism which delivered the cycles values in sorted order, thus, the cycles invariant (which is an $N \times N$ matrix) could be parsed apart in order of its columns (all of the values of cycles of length 2, then 3, then 4, etc)
- I used a clever set of minimally memory intensive structures to store only the information needed for a given unit of this computation. This, in combination with the second step, yielded a delayed stream processing, where we have a structure (with minimal memory), which can be called upon for another, more differentiating value.
- These structures (one per graph), were placed in an enormous trie, each level of which was the level of the delayed evaluation. Only values which differentiated between different structures were evaluated (lazy evaluation on trie-collision, splitting into sub-tries).
- Co-Cycles graphs were then simply detected by depth. After all graphs of a given size (with a given number of edges) were placed into the trie, only those at depth N^2 could possibly be co-cycles.

The lazy evaluation (in a couple of benchmark tests on eight vertices) worked more efficiently on the order of 10^{-3} times the original running time. Additionally, a trick I employed with bulk memory allocation reduced running time on the order of 10^{-1} (allocated 100 trie-nodes at once, rather than 1 at a time). An additional optimization I made was the singular evaluation of a combination of N and M at a given time. We know that the number of vertices and edges in a graph is reconstructible from the cycles invariant, so we only need to check for set values of N and M if we want to discover co-cycles graphs. This enabled me to reduce the overall amount of memory allocated to approximately a tenth of what it would require to process all graphs of a given size simultaneously.

Additionally, a smart way that I set up my matrices allowed for insanely efficient memory allocation: since I had to maintain every single graph in my trie, for even a small number of vertices that could have been an incredible amount of space. The graphs themselves were stored as pointers to arrays of integers (int**), which leads to a costly price in memory for even a small N (consider $N = 10$), where so refers to the C function $sizeof$:

$$Memory = so(int) * N^2 * NGraphs + so(int*) * N * NGraphs$$

$$Memory = 4 * 10 * (10 + 2) * 12,005,168 = 5,762,480,640 \approx 6GB$$

However, I realized quite quickly that since the matrices themselves were not changing over the course of the computation (the were being used in multiplications and what not, and being used to change other values and matrices), we could share their vector representations across multiple matrices. I decided not to have my algorithm waste time trying to canonize the graphs or find similarities, or shrinking down the individual memory storage of the graphs to the bit level (which would have been highly inefficient for the frequent scale-up/down necessary for computations as integers). Instead, I simply pre-allocated all of the 2^N possible vectors that could be the rows of a binary matrix. Thus, the only thing that changed was that each of the individual entries in this great large computation is pre-allocated and shared amongst all 12 million graph representations as in $N = 10$. The memory use for this case is shown to be dramatically reduced:

$$Memory = so(int) * N * NVectors + Sizeof(Int*) * N * NGraphs$$

$$Memory = 4 * 10 * 2^{10} + 8 * 10 * 12,005,168 = 960,454,400 \approx 1GB$$

This six fold increase in memory efficency allowed me to calculate this value on my laptop, rather than relying on the public workstation, which had 24GB of RAM.

2.6.2 Constructing Co-Cycles Graphs

[DO IF TIME]

2.7 Trie Depth and the Power of Cycles Across Classes of Graphs

In chapter three, we will discuss a notion of $P^*(G)$, a value for which (within a graph G), cycles beyond length $P^*(G)$ do not yield additional discriminatory information. Related is the same idea across sets of graphs. For example, if we have some set of graphs S, there exists an integer $P^*(S)$, at which values beyond this length cycle are not informative in further discriminating between graphs in the set. Intuitively, for a set with one element, $P^*(S) = 0$, and not-so-intuitively, with a set with only graphs which are co-cycles with one another, $P^*(S) = 0$.

The way that $P^*(S)$ is calculated takes a natural interpretation within our co-cycles computations in C. $P^*(S)$ is simply the maximum depth of the trie created when searching for co-cycles graphs, excepting nodes of full (N^2) depth.

2.7.1 Expectations Borne out of Graph Counts

A natural cadre of sets to explore is $S_{N,M} = Graphs(N, M)$, where $S_{N,M}$ is the set of all graphs with N vertices and M edges. Before we discuss the results of what we observed $P^*(S)$ to be over these sets, lets first discuss what we anticipate, and discuss a detail of its implementation. Naturally, we would think that the larger the set S, the more difficult it will be to fully discriminate between all of the graphs in it. If we fix N, the distribution of graphs within that N over various values of M is approximately normal, and approximation that increases in accuracy as N increases (see chapter 1). Thus, if we intuit that $P^*(S)$ is likely to be a function of the size of S, we would expect $P^*(S)$ to be approximately normally distributed.

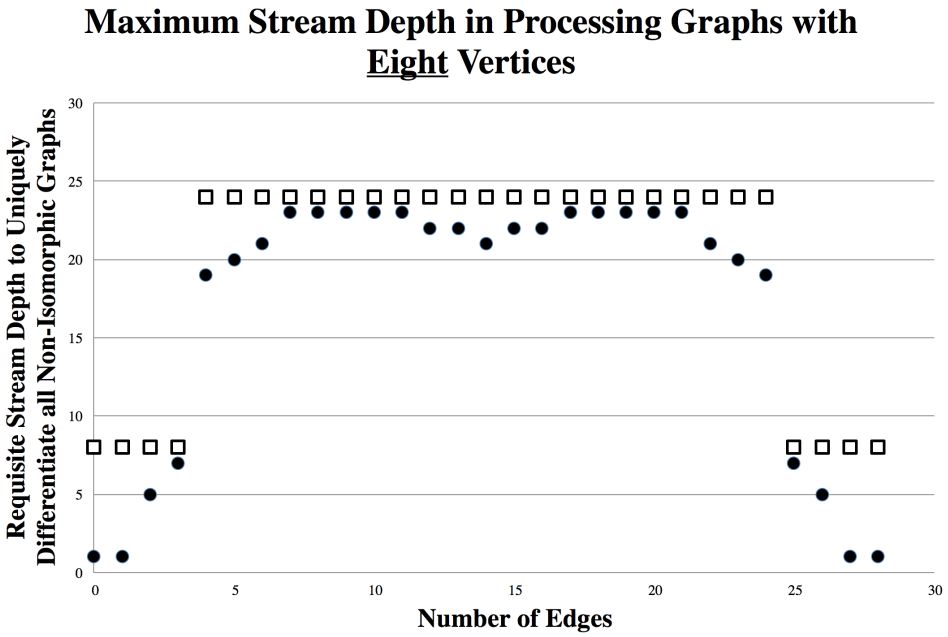
What we actually find is markedly different, in the next three diagrams, of this curve for N=8, 9 and 10:

In each diagram, the dots represent the maximum trie depth, while the squares above them round that value up to the nearest 'complete' depth. By complete depth, we are referring to a depth divisible by the number of vertices within the graph. This is relevant because these levels are the length of the cycles that correspond to all of the information gleaned from this length cycle and fewer.

2.7.2 An Unexpected Dip

What we observe is counter to our expectations. $P^*(S_{N,M})$ certainly is smaller at the fringes, but it actually dips in the middle, where the overwhelming majority of the graphs are located. Remember from chapter one that the number of graphs in each of these sets is maximized in the middle,

Figure 2.3:



while the ability to differentiate between them is actually maximized with connectivity around .25 and .75.

In figure $N = 10$, we see the same trend (slightly lower values for $M > 12$, than for $M \in (10, 12)$), despite the existence of co-cycles graphs, which push the middle up to 100 (maximal trie depth for N=10).

In figure $N = 9$, we see this trend exaggerated to the point that it distorts not only the granular values (the black dots), but the overall cycles length values (boxes).

It should be noted that the asymmetry in diagram for $N = 9$ was highly suspect to me. I looked into it, and was able reproduce it using Matlab (in addition to the calculations in C). It turns out that the asymmetry is due to the lack of invertible equivalence in the Cycles invariant, and for some graphs in $S_{N=9,M=25}$ can be differentiated by their cycles values, while their inverses cannot.

This small set of experiments suggests an interesting property of graphs: it may be easier to differentiate between graphs which are half connected than a quarter connected, despite the fact that there are significantly more graphs which are half connected.

Figure 2.4:

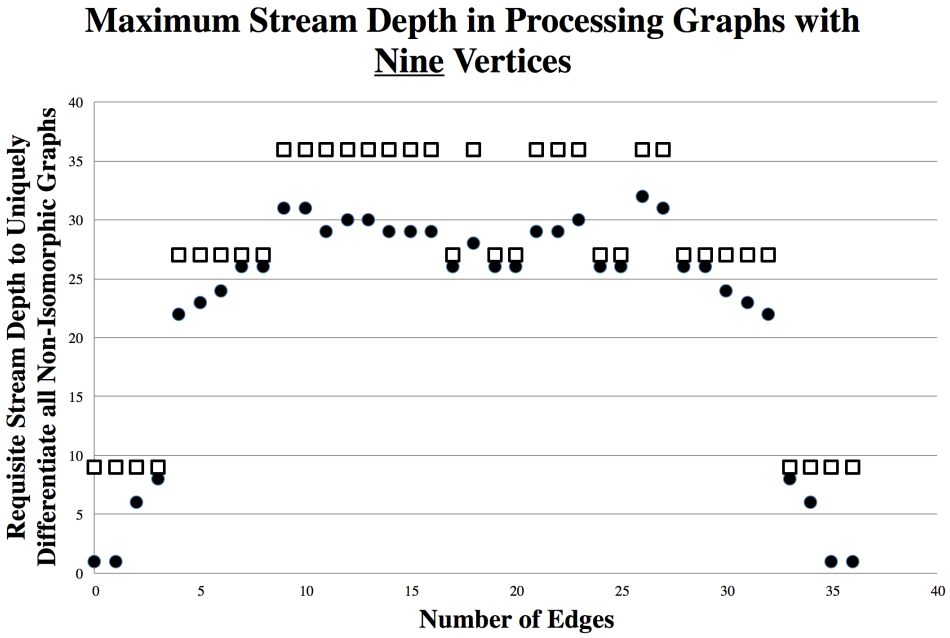
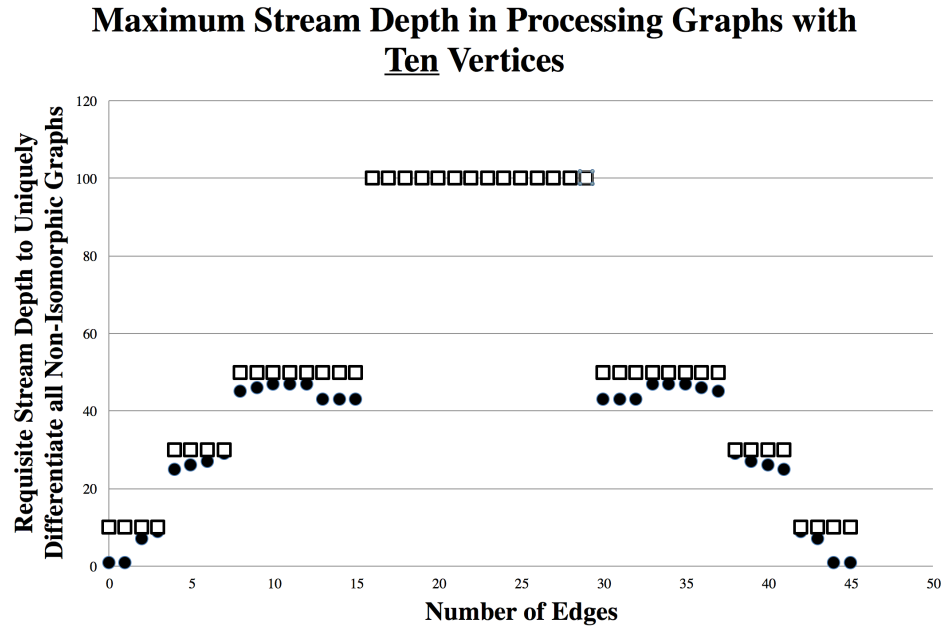


Figure 2.5:



Chapter 3

Cycles as a Vertex Invariant

A powerful feature of the Cycles invariant is that it cannot only discriminate between graphs, it has structure which naturally suggests mappings between their vertices which have the potential to be isomorphisms. This section will discuss exactly how isomorphisms can be constructed and evaluated by using cycles as a vertex invariant. Central to this chapter is the concept described earlier as Automorphism Equivalence Classes, or Similar Vertex Sets (SVS).

3.1 Similar Vertex Sets

A vertex-similar set is a subset of the vertices of a graph such that for any pair of vertices in the set, there exists an automorphism which maps one to the other. Note that vertex similarity is reflexive (as one-to-one mapping (such as an automorphism) is invertible). Also note that membership in vertex-similar sets is transitive (as we can simply use the 'followed-by' operator on the component automorphism mappings).

Thus, all vertices in a graph can be partitioned into some number of similarity-sets, where every set has at least one element, and the union of the sets is the vertex set. It is the structure of these disjoint, covering sets, each of which is internally vertex-similar which we will call Similar Vertex Sets, or SVSes for short. When we are referring to a single vertex-similar set, we will say 'an SVS', and when we are referring to the covering structure of multiple, disjoint, covering sets, we will refer to them as the 'SVSes'. For the sake of simplicity, we will assume that there is some well-ordering over these sets, so that we are able to assign an order between each SVS within the SVSes. In practice, this will be a practical function of how we go about computing the SVSes.

This construction is deeply tied to isomorphism checking and discovery. If we have two graphs G and H, and their Similar Vertex Sets are $SVS(G)$ and $SVS(H)$, based on the size of each set, we have a maximum number of potential isomorphisms. First off, if we examine every ith paired set

between $SVS(G)[i]$ and $SVS(H)[i]$, if the sizes of the sets differ, then we can immediately reject the possibility of isomorphism between G and H. If the size of each component set are the same, we can check for isomorphism by brute force, by calculating every possible permutation between the paired sets, and then every combination of those permutations across all of the SVSes.

One should note that this is simply a way of describing the end goal of a vertex coloring (where like-colored vertices are automorphic to one another). SVSes are easier to discuss concretely within the context of Cycles, and have the additional feature that we allow them to be imperfect, or defined by a particular facet of the graph/invariant.

Though we can make many practical arguments which greatly simplify the number of these possibilities we have to test for isomorphism, it should be noted that in the overwhelming majority of cases, this simple algorithm only has to check *one* mapping for an isomorphism. This is by virtue of the fact the the overwhelming majority of graphs have only a single automorphism.

3.2 Most Graphs Have One Automorphism

This is by virtue of a simple fact: we can talk broadly about the average number of automorphisms in a set of graphs, and even better, we can calculate precise means for the average number of automorphisms for graphs of a certain size. Remember that the number of undirected, non-looped, single-edge graphs (or as we have just been calling them, graphs), over N vertices has a known closed form. We also know that the number of graph instances over N nodes is simply the number of undirected matrices over N nodes, $2^{E_{max}}$. Finally, we know that the number of matrices

$$M_{reps}(g) = \frac{N!}{|Aut(g)|}$$

$$\sum_{g \in G_{alg}} M_{reps}(g) = \sum_{g \in G_{alg}} \frac{N!}{|Aut(g)|}$$

$$2^{\frac{N^2-N}{2}} = N! \sum_{g \in G_{alg}} \frac{1}{|Aut(g)|}$$

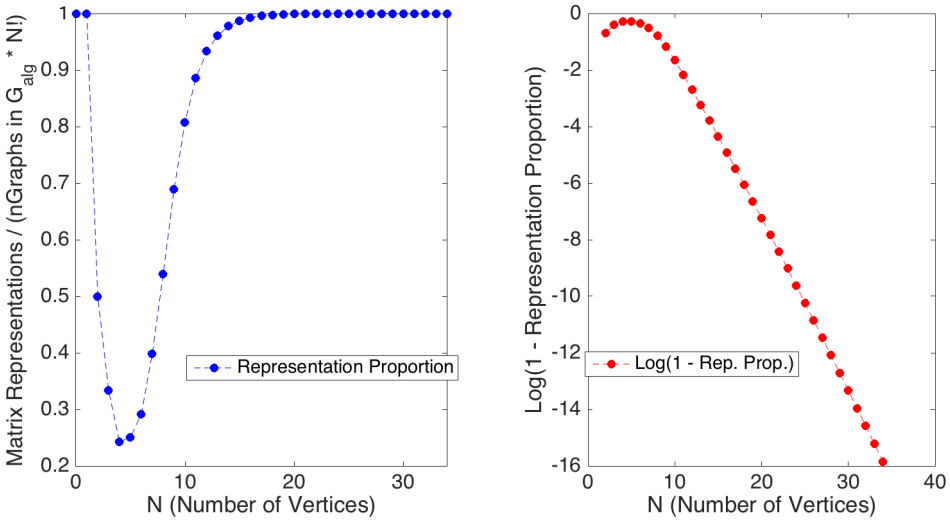
$$\frac{\sum_{g \in G_{alg}} \frac{1}{|Aut(g)|}}{|G_{alg}|} = \frac{2^{\frac{N^2-N}{2}}}{N! * |G_{alg}|}$$

$$|\overline{Aut^{-1}(g)}| = \frac{2^{\frac{N^2-N}{2}}}{N! * |G_{alg}|}$$

Though we don't have a closed form for this, a quick calculation pretty easy, and gives us a very clear trend. In Figure 3.2, there are two figures

shown. The first plot shows the right hand side of the final equation, graphed for different values of N. The second plot explores the difference between the first plot and one, and takes the logarithm to see how small the values get.

Figure 3.1: *Most Graphs Have Exactly One Automorphism* - It is clear that as N increases, the right hand side of our above equation approaches 1. Not only that, but it appears to do so exponentially fast. The plot on the right shows a logarithmic linear approach to one.



This should convince even a skeptical reader that for an average graph g (even when treating it like an algebraic object), as N increases, $|\overline{Aut(g)}| \to 1$.

3.2.1 Implications for Perfectly Similar Vertex Sets

Applying this back to our original purpose, we can surmise that the proportion of graphs which have only the trivial automorphism increases to one as N increases. In this common case, the SVSes are a set of N distinct, one element sets, where each vertex is distinguishable from each other, and none share any automorphisms. This makes checking for isomorphism between two random graphs trivial (if we have the SVSes), as the overwhelming majority of the time, we will only need to try the isomorphism implied by the direct, one to one mapping between the SVSes of the two graphs.

However, checking for automorphisms between every single pair of vertices is clearly in a higher computational complexity class than testing for isomorphism. Thus, we will use smart heuristics to develop quazi-similar vertex sets, which will enable us to get the benefits of SVSes (i.e. implying the isomorphisms to try) without actually proving that the automorphisms that define the SVSes exist.

3.3 Quazi-Similar Vertex Sets

Quazi-Similar Vertex Sets (QSVSes) are constructed with respect to a vertex invariant, but have the same structure as SVSes. The vertex invariant is calculated for all vertices within the graph, and those with the same value are placed into the same set. Transitivity and reflexivity clearly both hold under this definition. Moreover, the ordering/comparability of vertex invariants gives us a natural way to order the sets within the QSVSes.

In this section, we will describe how using the cycles invariant to generate QSVSes is highly successful at mirroring the true SVSes, and suggest an augmentation to cycles which correctly differentiates the two for all observed cases.

3.4 Why $P^*(G) < N$: Limitations on Cycles Usefulness

3.4.1 What is P^*, why does it matter?

An ENORMOUS part of this thesis has thus far gone as assumed: what is P (the maximum length cycle we are considering)? When we are discussing cycles through a node, how long are the cycles we are considering? The vertex invariant for cycles is clearly a vector, with the length of cycles corresponding to the place within the vector (cycles of length 2, 3, 4, etc.), but where do we draw the line? To answer this question, I used a notion of 'usefulness', to maximize the amount of information that we get out of the cycles invariant for the computation that we put into it.

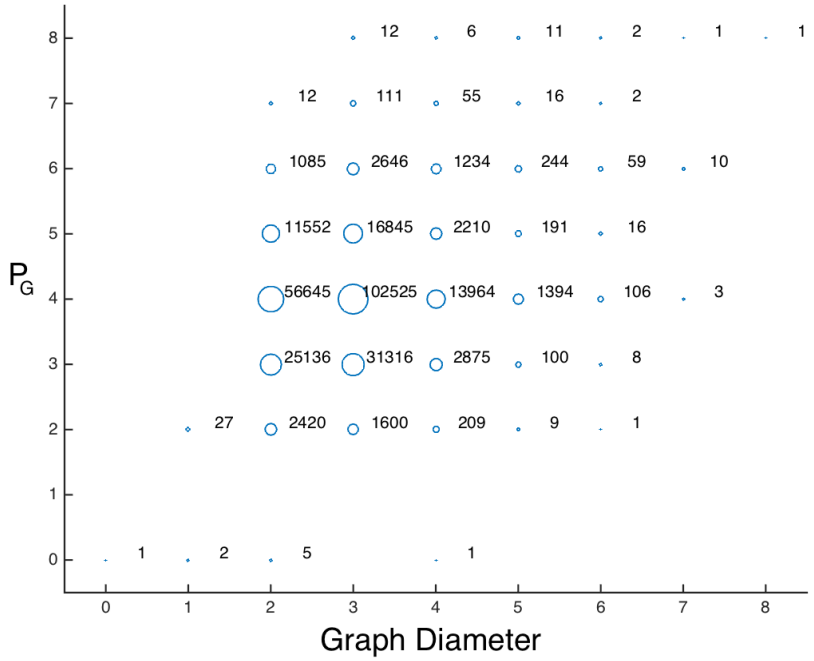
Specifically, within the context of cycles as a vertex invariant, I started out with the assumption that all vertices share a single Similar Vertex Set (i.e. all are similar unless we have proof that they are not). We then distinguish vertices by advancing P by one, we are in effect splitting this one class (or smaller classes) into even smaller classes, based on the information gleaned by increasing the length of all invariant vectors by one. This kinds of 'breaking-down' leads us to a concrete notion of usefulness: for a given graph G, cycles is useful at a power $P^*(G)$, where $P^*(G)$ is the minimum value for P at which all of the vertex sets in the SVSes are as broken down as they can get (for any larger values of P). Thus, if we take the maximum of $P^*(G)$ across the set of all graphs over N vertices and M edges, , we will find a value $P^*(S_{N,M})$ at which point computation beyond this point is redundant for graphs over this set of graphs. In this section we will be ignoring M, and examining the critical value for P over all values of M while fixing N. This would enable us to more concretely talk about running time, and its tradeoffs as a function of N alone. What we find (through both computational exploration and theoretical justification) is that $P^*(S_N)$ can be described in direct terms of N:

$$P^*(S_N) = \begin{cases} N-1 & \text{If N is odd} \\ N-2 & \text{If N is even} \end{cases}$$

3.4.2 Observational Data: Diameter vs. $P^*(G), G \in S_N$

I first examined $P^*(G)$ as a function of the individual graph G, rather than over the set of all graphs, to try to see how the overall distribution of $P^*(G)$ looked for all graphs in the set. Though this did not lead me anywhere directly, it did give me a clear idea of the upper bounds on $P^*(G)$ for an individual graph. Shown in figure 3.4.2 is a description of this relationship. By no means is it rigid, but the correlation is very strong between diameter and $P^*(G)$, labeled as P_G on the y-axis. It is also interesting (personally) just to see the diameter of a set of graphs laid out like this.

Figure 3.2: *Diameter Against* $P^*(G)$ *with* $N = 9$ - The size and number next to each circle tell us how many graphs (as algebraic objects) fit into different sizes of diameter and $P^*(G)$, labeled P_G



Four things are consistent across all of the alike graphs (shown in an appendix):

- There is consistently a maximum diameter for $P^*(G)$, implying a $P^*(S_N)$ for each as shown in the table below

- There are never any graphs with $P^*(G) = 1$, as we do not allow self loops
- The Path graph (P_N), is always the lone member of the far upper right hand corner
- The Cycle graph (C_N), always has $P^*(G)$ equal to that of the Path graph: $P^*(C_N) = P^*(P_N)$

The upper bound proposed by each one of the values for $P^*(S_N)$ follows the pattern:

$$P^*(S_N) = \begin{cases} N-1 & \text{If N is odd} \\ N-2 & \text{If N is even} \end{cases}$$

for $N <= 9$. This sets up a strong case for this as a target hypothesis, but we still must prove it.

3.4.3 Theoretical Justification: P_N versus C_N

Why does the proposed value for $P^*(S_N)$ make sense?

This proof consists of two parts, the first is why $P^*(P_N) =$ the proposed$P^*(S_N)$ for the Path graph (P_N), the second describes why all other graphs G have $P^*(G) \leq P^*(P_N)$.

Part 1: *Distinguishing between vertices in the path graph requires* $P^*(P_N) = N - 1 - ((N+1)\%2)$.

Start all vertices in the same assumed similarity set, and assume that $P = 0$. When we increase P to 2, all vertices have degree 2 (the second entry in the cycles invariant) except for the two terminuses, thus the two terminuses are split off from the original quazi-similarity set, as they have been distinguished at $P = 2$ by their cycles invariant. When we increase P to 4, the two nodes adjacent to the two terminuses are now distinguished (as they are now 'missing' a path of length four that they would have had if they were further in toward the center of the path graph). There are now $N - 4$ vertices remaining in the original quazi-similar vertex set. From this, it is clear to see that for an arbitrarily large path graph, for $P = 2K$, there will be $N - 2K$ elements in the original QSVS. We will have *fully distinguished* all of the vertices into distinct QSVSes which are true to the SVSes when the original QSVS contains only the 'middle' node(s).

In the even case $N = 2K, K \in Z$, this occurs when we have

$$P^*(P_N) = 2\lceil\frac{N-2}{2}\rceil = 2\lceil\frac{2K}{2} - 1\rceil = 2\lceil K-1\rceil = 2(K-1) = 2K-2 = N-2$$

In the odd case $N = 2K + 1, K \in Z$, this occurs when we have

$$P^*(P_N) = 2\lceil\frac{N-2}{2}\rceil = 2\lceil\frac{2K+1-2}{2}\rceil = 2\lceil K - 1/2\rceil = 2K = N-1$$

Thus, we know that

$$P_{P_N} = \begin{cases} N-1 & \text{If N is odd} \\ N-2 & \text{If N is even} \end{cases}$$

[TODO HERE PART 2]

[TODO HERE, PROBABILITY OF SVSes mirroring QSVSes created via the cycles invariant!!!]

3.5 Improving upon QSVS with Flagged Cycles

The quality of the QSVS is generally described as how close they are to the true SVSes. In other words, quality is a measure of how frequently does a QSVS contain two or more subsets that should be independent SVSes. Though Cycles as a vertex invariant produces QSVSes which mirror SVSes with high probability, we should examine the failures cases to see if we can do better.

We will accomplish this via a methodology I am calling 'flagging' after I have seen that a few times in the literature. Flagging (or Marking) is the process of taking a graph and appending a single vertex to it which is connected by a single edge to some target vertex in the graph. Flagging frequently modifies graph structure enough to see impacts which successfully differentiate between non-isomorphic/non-automorphic structures.

For example, if we have two co-spectral graphs, and we are curious whether or not an isomorphism exists between the two of them, we can append a flag on each one of the vertices which must be mapped together in a potential isomorphism, and then take the chromatic polynomial again. If the new (flagged) graphs differ in their chromatic polynomial, then the two vertices that were flagged cannot have an isomorphism to one another, as modifying two isomorphic graphs in a systematically consistent way should preserve isomorphism. We use this principle in the pursuit of fully differentiating a set of QSVS into a refined state that is more likely to equal the ground-truth SVSes.

3.5.1 Appending a Flag, Somewhat Predictable

Our methodology for modifying the QSVSes is simple: first, calculate the QSVSes using the Cycles invariant. For each of the sets within the QSVSes with more than one element, perform a flagging operation K times, if the set has K elements, which results in K different graphs. Calculate the cycles graph invariant over each of these graphs. If the cycles invariants for any two of the K graphs differs, then split apart the set of vertices into two (or more) new QSVSes, one for each different value of the graph invariant associated with the graph formed by flagging each vertex.

An interesting piece of this is that we can actually predict some of the values that the newly created, flagged graphs, will have for their cycles invariant values. Specifically, if we are given the previous value of the cycles invariant, and told which vertex in the cycles invariant is going to be appended to, we can predict the number of cycles for the flag vertex (the new vertex), as each one of the cycles that it is a part of must go to and from the flagged vertex (the vertex the new vertex was attached to), and we already knew how many of those there were from the original cycles function. Additionally, we can predict the change in the number of cycles for the flagged vertex, as we knew its cycle profile before, and the addition of a single vertex on to it can be interpreted as a recursive definition, where cycles are broken into components that travel back (and forth) from the flag vertex and those that exist within the graph. Though difficult to explain via formulae and words, I would encourage a skeptical reader to check out my code at (/Thesis/Matlab/constituentpaths/predictPathsOfAddingOneVertex.m).

Equally interesting (to the fact that we *can* predict paths for these two nodes when just given the cycles invariant) is that we *cannot* make any further predictions about the flagged cycles values based solely on the plain cycles values. We know this to be true because in co-cycles graphs, we frequently get different flagged cycles graph invariants when we flag vertices which are suggested to be isomorphic (but are not). This is a good piece of information to know because it gives us proof that flagging is not some kind of manipulation of information that we already have, it represents a way to get pieces of information that include additional structural information.

3.5.2 Intuitive Justification for Flagging

This section is not grounded in proof, but gives a conceptual reason that flagging works at further discriminating between vertices that we think might be automorphic. Cycles is really a rough description of the 'neighborhood' of a vertex within its context in the graph. Cycles tells us about the way that the structure reverberates, it is, in many ways, a description of resonance. One conceptual tool I like is to think about electricity flowing (bidirectionally) around a circuit, and measuring the self-connectivity or resistance between points based on the number of ways that electrons could flow.

Flagging a node changes that resonance. It changes it in a way that we understand and can predict for the flagged vertex and the flag vertex. However, it also changes the way that cycles 'reverberate' through the flagged vertex, and in doing so, changes the cycles invariant at other vertices. We can even think of this in more concrete terms: take the cycles invariant for the flagged graph (without sorting), then subtract off the cycles invariant for the non-flagged graph (again, without sorting).

This tells you exactly how many new cycles have been created which pass through the flagged node at least once. We call this modified matrix

the *excess cycles* matrix. Note that this is *not* the same as the row for the flagged cycles matrix, and actually encodes significantly more information (the constituents of those paths, not only that they exist). In doing so, we actually get tangible information about the connectivity of each of the nodes. From the excess cycles matrix, we can calculate the distance between the flagged vertex and each of the other vertices in the graph. We can calculate the relative makeup of each of the added cycles (how many pass through each vertex, and how many times does each double back on itself), among a large number of other features.

The intuitive justification for flagging is compelling, and if I had another month to puzzle away at this problem, this is probably the area I would spend the most time focusing on.

3.5.3 Analytical Support for Flagging, and an Open Question

The analytical support for flagging is strong: for all graphs of size $N \leq 10$, cycles with flagging as a mechanism for QSVS generation correctly generated QSVSes which perfectly estimated (without flaws) the accurate SVSes. No counterexamples were found where cycles with flagging fails to be anything short of a perfect vertex invariant.

This begs an open question: is it possible that cycles with flagging is complete as a vertex invariant? Is it possible that this metric fully determines the automorphism sets of vertices? Possibly. However, it is more likely (based on the raw number of graphs out there) that we have only kicked the proverbial can further down the road. For one, the raw amount of information required in Cycles with flagging is $O(N)$ times the amount of information required for cycles comparison. This alone should tell us to expect a higher threshold for failure.

That being said, there are properties that are reconstructable from excess cycles computations that make me think that flagging might have some higher order justification. I would love to spend another few months thinking of equations that I can use to reconstruct A (or better yet, determine A) from flagged cycles.

Chapter 4

Canonical Labeling Using Cycles

A canonical labeling of a graph is a labeling of edges in a way that is consistent across all isomorphic graph instances of the same graph. Given two graphs and their canonical labelings, graph isomorphism is then a simple quadratic-time check to make sure that alike-labeled vertices preserve all adjacencies and non-adjacencies. Thus, any canonical labeling algorithm cannot be in a complexity class smaller than GI, the time complexity class of the graph isomorphism problem.

Many parts of my coding projects required the use of a canonical labeler. Sometimes this was because I wanted to check for isomorphisms on a broad set of graphs, a problem that is possible to complete quickly given the lexicographical nature of canonical labels. Though there exist good algorithms for determining canonical labeling, I wanted to create my own, both as an exercise in implementation, and in algorithm design. The result of this process is an algorithm which is relatively slow, but whose time growth is small relative to faster algorithms, due to the relatively large amount of computational work that is done regardless of the difficulty of the graph.

4.1 Start With A Vertex Invariant

The fundamental unit of computation that drives a canonical labeler with this algorithm is a vertex invariant. For the canonical labeler that I created, I used the flagged cycles vertex invariant, which is described in the third chapter. Understanding the canonical labeler is not contingent upon an understanding of this vertex invariant, however it is a strategic choice that is rooted in finding a good use for the cycles invariant.

What this does for us is simple, it allows us to directly measure and compare the use of cycles as a vertex invariant to others within a time/power tradeoff. The result is a canonical labeling algorithm that has much higher

constants than its established counterparts, but has a comparably lower time growth rate. This is a direct reflection of the quality of Cycles with flagging as a vertex invariant.

4.2 A Consistent Algorithm

The algorithm has a simple interpretation. It divides all of the vertices of the graphs into disjoint (covering) sets (QSVSes) based on the value of the cycles vertex invariant. For each of these sets with more than one vertex, a second round of 'flagged cycles' is performed to further divide sets, if possible. It sorts each of these classes by its size, then by its cycles values. Each of these sorts is performed in a consistent way, which is irrespective of the original labeling of the graph.

These sorts optimize the interrelationships between vertex sets, so that we are likely to eliminate lexicographically small matrices before even testing them. This is a version of exponential decision tree pruning, and how the order of the decisions can make pruning easier or harder. While maintaining our lexicographical target, we are able to significantly reduce the number of checks that are necessary (on the order of N!, in the best case).

Once the order of the sets is sorted, we consider every possible permutation of the values within each of the sets so that every possible ordering of vertexes is possible, with the constraint that each set element remains in the higher level order established by the sets themselves. We start by examining a specific QSVS, and its associated permutations. We traverse through these permutations and select the matrix which is lexicographically the 'smallest', within the context of the already established graph, if some has already been established. We then add this piece to the growing graph (selecting all 'lexicographically maximal' representations), and move on to the next set within the QSVS. This step-wise pruning means we minimize the overall number of permutations that we test, though it is possible that we accept multiple permutations after a given step of this process is done. In the final step, we may have multiple permutations remaining (over all the vertices now), but in this context, every permutation will produce the same matrix representation.

I wrote and optimized the algorithm for Matlab (as that is the slowest of my use cases, so having fast code makes a big difference), but then worked on implementing it in C and Javascript (though I didn't go to the same lengths to do the tree pruning). All three versions of this code are available in my online repository documenting my thesis work. The Matlab code provides detailed explanations of each piece of the code, why I implemented it that way, and the thoughts that went into it.

The result is quite pleasing, and (alongside a memoization mechanism that saves all results that take longer than 1ms to calculate), canonization of

graphs does not bottleneck my computations (even when I am doing millions a minute), despite it being the only computation that I use that operates in exponential time.

4.3 Computational Complexity

There are several factors which contribute to the average case running time of this algorithm, but its worst case is simple to compute: a perfectly automorphic graph, where $|Aut(G)| = N!$. In such a case, we will perform the original Cycles computation ($O(N^2)$), the secondary flagged cycles calculations ($O(N^3)$) and then a factorial number of checks on the number of remaining isomorphisms ($O(N!)$). Thus the worst case running time is clearly in factorial time ($O(N!)$).

However, in the average case this is a much less grim scenario. If we assume (as we have reason to believe) that our flagging algorithm correctly determines SVSes, then our algorithm becomes a function of the number of automorphisms of our graph, in addition to the running time of the flagged paths algorithm ($\Theta((|Aut(G)| + 1) * N^2)$). As we have seen in chapter six, the average number of automorphisms in G approaches 1 as N increases. Thus, for sufficiently large N, the average running time is on the order of $\Theta(N^2)$.

4.4 Time Growth Comparisons to Faster Algorithms

Any discussion of a homegrown algorithm would be either incomplete or intentionally salacious without a discussion of how it compares to out of the box algorithms. For these benchmarks, I established two datasets, one of 10000 randomly generated graphs of size ten to fifty using an Erdos-Renyi/Gilbert model, and one using a graph model which is more likely to produce highly automorphic graphs (discussed in detail in chapter 6). For both of these models we used a probability of $p = 0.5$, as these are typically the graphs that are hardest to discriminate against.

To give perspective to my own time results, I used an established and well optimized piece of code, the canonical labeler from the NAUTY package. A summary of the results is shown below, alongside equations which estimate their overall running time with the above parameters. From these figures it is clear that the NAUTY package outperforms my canonical labeler, but that my labeler appears to have a lower growth rate. I would imagine that this difference can be attributed to the large lengths my labeler goes to to minimize the number of permutations that are checked. NAUTY's algorithm uses simpler heuristics (such as X and X) to eliminate permutations, while mine uses significantly more power to make likely better informed choices.

Chapter 5

Random Graph Generators and Automorphisms

5.1 Random Graph Models

The field of random graph theory was started with two independent papers in 1959, each defining models for generating and analyzing random graphs. A *random graph model* is any system of generating graphs with the influence of chance, with various constraints that describe the desired properties of the resulting graphs. These two papers began a trend in graph theory that allowed theoreticians and algorithm designers to think about efficiency differently, particularly on NP-Hard and NP-Complete graph algorithms. Rather than concerning themselves with worst case run time, theoreticians began describing and designing algorithms to satisfy ideas of average case running time over the 'class of random graphs'. Random graphs served as a new lens through which theory could pivot away from the worst case, instead handling common and general cases of problems that were either proven to be impossible in the worst case, or suspected to be so.

A few examples from the years following showed that random graph theory lent a new life to the challenging problems of CITE, CITE and CITE. In this chapter, we will discuss multiple models of random graph generation, examine the strengths and flaws of each, and propose alternative mechanisms by which graphs can be generated for more theoretical applications and algorithms.

It is important to recognize that there has been an increase in interest in alternative random graph models in the last ten years. The majority of these papers focus on exponential random graph models, which are centered around the study of network structure with local and global patterns of connection. For example, the internet (if sites are viewed as vertices and links as edges), has an exponential in-degree distribution (many more people link to cnn.com than gradybward.com) [**?**]. Another set of random graph models

tries to model social networks, which are categorized by cliques (groups of friends), with alternative connections that symbolize non-group friendships. For either of these models, algorithmic theory is inadequate if it treats all graphs as equally likely, or if it treats all graph instances as equally likely. Though this section is not focused on these models, it does adapt some of the characteristics and heuristics of this pursuit to a different goal, of establishing a random graph model for testing algorithms over the set of all graphs.

In this pursuit, we will question what we mean when we say 'random' graphs, and make explicit the assumptions, aims and valid applications of any one of our models.

5.1.1 The Erdös-Rényi Model(s)

In their seminal paper on random graph theory, Paul Erdös and Alfred Rényi proposed two different models for random graph generation. The first of these models will be outlined in this section, and will be referred to as the Erdös-Rényi Model. The second was laid out in the same 1959 paper, but was also discovered by an independent contemporaneous mathematician, Edgar Gilbert. For the sake of clarity, we will refer to the contemporaneously described model as the Gilbert Model, and the one that is about to be outlined as the Erdös-Rényi Model.

The model $G(N, M)$ is defined as choosing a graph G with uniform probability from the set of all graph instances with N edges and M vertices. The size of this set is $\binom{E_{max}}{M}$, where $E_{max} = \frac{1}{2}(N^2 - N)$ represents the maximum number of vertices possible in a graph over N vertices. Though the probability of getting any graph *instance* with N and M edges out of this model is uniform, the probability of getting any graph with those constraints is not. Since the number of representations of a graph fluctuates along with other properties of the graph, this random generator has the flaw that certain graphs are more heavily weighted than others. This is a flaw that we will discuss at length in discussion of the Gilbert Model.

In the study of random graphs, the Erdös-Rényi model has not been as popular as the Gilbert model because it is more cumbersome to deal with, and has fewer concrete applications CITE. Though some papers have utilized this model (namely CITE and CITE), the majority of theory is better suited to the combinatorial and probabilistic methods that are made useful by the Gilbert model. Though knowing the number of edges in a graph gives us some information about the graph, it turns out that the probability of a given edge, and the guarantee of its independence, is significantly more malleable to theoretic goals.

5.1.2 The Gilbert Model

The second model, which we will call the Gilbert model, comes from the mathematician Edgar Gilbert (as well as Erdös and Rényi) and is denoted $G(n, p)$. In the Gilbert model, n specifies N, the number of vertices in the graph, and every pair of vertices is connected by an edge with a fixed and independent probability p. The Gilbert model is both intuitively pleasing, justified by real world use, and has convenient properties for proof.

The Gilbert Model is an effective model for real world applications where graphs are thought of as occurring naturally without oversight or intervention. If we think about the configuration of the a network generated by actors acting randomly, the Gilbert model is appropriate. For example, consider a cocktail party among strangers, where the odds that any two people have a conversation in a given evening are likely independent and uniform. Or, consider the reproduction of coral, where fertilization of one coral by another is reasonably random through the fluid dynamics that carry, combine and disseminate their pollen. Gilbert's model gives us the language to describe graphs that pop up in wide-ranging uses, and a model to express the assumptions we frequently make about graphs in practical applications.

Additionally, the Gilbert Model allows us to make bold proof-based claims about random graphs through established combinatorial and probabilistic methods. For example, if we try to estimate the number of edges within the a Gilbert graph, we simply are asking the binomial question with n and p, and we have a readily available probability distribution to answer our questions. A more interesting example arises when we ask about the number of triangles expected in a large graph. If our graph is sufficiently large enough, the existence of one triangle does not impact the potential existence of another. We can express the number of triangles as a simple combinatorial problem: multiply the total number of triangles possible $\binom{N}{3}$ by the probability of all three edges existing (p^3). This shows how combinatorics gives us tools to deal with Gilbert random graphs, and to make theoretical statements about expectations of the properties of these graphs.

Finally, the Gilbert model has a revelatory connection to matrix representation. Consider the model with a fixed probability of $p = 0.5$ and some fixed N. We will show that this random generator has a uniform probability of selecting any matrix from the set of all valid graph instances of size N.

Consider the range of integers $[0, K^2 - 1]$. If we assume numbers are left-padded with infinite zeros, the bth bit of a randomly selected integer from this range has an equal probability of being a 1 or a 0, as exactly half of these numbers have each bit set. This is trivially true through the fact that there are K^2 integers in this range, and K^2 different bit strings. Since each bit string is only achievable with exact probability $(0.5)^K$, each integer is generated with the same, uniform probability. We will reshape the bit-string into representing each one of the edge variables, and we let $K =$

$E_{max} = \frac{1}{2}(V^2 - V)$. This establishes a connection between the Gilbert model and a randomly selected matrix from the set of matrices which represent our definition of valid graphs. This connection is intuitively pleasing, but further investigation should also show that its implies skewed results for some algorithms which rely on it.

5.1.3 Isomorphism Under the Gilbert Model

One of the first places that theoreticians turned their attention toward after the start of the study of random graphs was the problem of Graph Isomorphism. The dominant lens of that study was the discussion of naming a class of graphs, and having the following proof structure:

- The class of graphs is closed under isomorphism (i.e. all graph instances are in the class of graphs)
- The class of graphs includes a large proportion of all graph *instances*
- Any two graph instances in the class of graphs can be determined isomorphic or not in polynomial time

This approach was undertaken by a large number of theoreticians in the 1960s and 1970s. A secondary component of these papers became the discussion of alternative algorithms which could solve the cases that were not solved for by the large model of the paper. It seemed (to paraphrase from Erdos), that graph theoreticians were going to chip away at the problem of graph isomorphism until there was nothing left to chip away. Yet, as it went forward, the increasingly restricted set of graphs for which no fast algorithm was known did not vanish. Instead, theoreticians (pardon the projection) were surprised to find that Graph Isomorphism was a hydra that they couldn't vanquish through these means. No matter the number of large classes of graphs they covered, no matter the diminishing proportion of graphs that were left uncovered, they couldn't find a way to solve all cases.

Hindsight is clear, and we can see that the theoreticians' realization was really a self-reflexive commentary on the work that we are going to be doing in this section. What they had discovered is that the easy cases in isomorphism are exponentially (in fact, factorially) more common under a Gilbert model than they are if we treat them as algebraic objects. Thus, even as they shrunk the probability of seeing one of these highly automorphic graphs under a gilbert model, they did little to increase the overall number of graphs that their algorithms covered.

The work we are going to do in this report is taking the opposite approach. Rather than trying to create algorithms that only cover a set of easy to handle graphs, we are going to ask how we can explicitly look for the harder cases. There are a number of justifications for this kind of work, but one simple one is the evaluation and clarification of random graph models.

Initial attempts at modeling social networks, first through gilbert models, then through exponential models, failed to appropriately assess the true nature of the problem. Similarly, algorithm designers who use a gilbert model to test against the 'average' graph should have their algorithms checked against a truly 'average' graph.

Treating graph instances as graphs allows theoreticians to use easier math, and allows algorithm designers to inflate their claims. In the next section, we will show how a simple problem (like graph isomorphism) has dramatically different theoretical and experimental analysis when performed over all graphs than when it is performed over all graph instances.

5.2 Disconnect from Graphs as Algebraic Objects

Though it is the foundation for most probabilistic random graph theory, the Gilbert model is has a different meaning than we typically think when discussing 'random' generators of other kinds. When we consider most other discussions of 'uniform randomness', we state the assumption that the result element was selected from its set with a uniform probability. Moreover, we generally assume that each object within that pool was represented the same number of times. When I say 'a randomly generated integer from the range', we are all agreeing on assumptions of what integers fall within this range, as well as how many times each is in our pool for selection (namely, once). Whereas, when I say 'a randomly selected word from a book', there is the possibility that common words are more likely to occur, or it could mean that I found the unique set of words in the book, and I am selecting from that.

This is where random graph theory and colloquial understandings of randomness miss one another. Throughout this work I have gone to great lengths to distinguish between graphs (an entity that has a given structure), and graph instances (a given representation of that structure). Most graphs have many distinct graph instances; many different ways of representing themselves, but this number varies as a function of the properties of the graph.

The problem with the two models outlined above is that they select a random *graph instance* with a uniform probability, but this does not translate to our understanding of graphs as algebraic objects, which denote structure irrespective of representation. Thus, a model which chooses graph instances with uniform probability does not choose graphs with uniform probability, just as selecting a word at random off of the page of a book is not equivalent to selecting a word from all of the words in the book with uniform probability.

Consider an illustrative example with two graphs, G and H, on N vertices and M edges. Graph G has only the trivial automorphism, and Graph H

has an automorphism group with 20 elements. It was shown by XXX in CITE that the number of distinct matrix representations (and thus distinct labelings) of a graph is equal to $\frac{N!}{|Aut(G)|}$. Thus, the number of graph instances that represent graph G is $N!$, while the number of graph instances that represent graph H is $\frac{N!}{20}$. Since graph instances are really just a way about talking about the number of matrices which represent the graph, this means that in the set of all valid undirected, non-looped graph matrices, there are 20 times as many which represent G as represent H. This is critical because the two models of selecting random graphs select a matrix with a certain number of ones (some number of edges) with equal probability. Even when the probability is not $p = 0.5$ as it was in the illustrated case, it is clear that this is true. This means that the probability of selecting a matrix which represents graph G is twenty times more likely than selecting a matrix which represents graph H under either 'random' graph generator.

Though this seems like a semantic difference, as I will show over the next several sections, it has critical implications for the algorithms that use it to argue about computational complexity.

5.2.1 Comparing the Distribution of Graph Connectivity

In Chapter 1, we described the overall distribution of graphs as algebraic objects with respect to their connectivity. We can also model the connectivity if we are considering the set of graph instances. Since there is a one to one correspondence between graph instances and the set of all valid zero-diagonaled, binary symmetric matrices, we can discuss the connectivity of graph instances as a function of the binomial distribution. That is because we can view the process of random graph instance generation under the gilbert model with probability .5 as a sequence of disjoint (as equally probable) binary variables, one per edge. Thus the number of edges in a Gilbert random graph follows a binomial distribution with E_{max} trials and fixed probability of success $\frac{1}{2}$.

The variance of such a model is known to be

$$Var(Binom(n,p)) = (1-p) * p * n = E_{max} * .5 * .5 = \frac{E_{max}}{4}$$

Since we want to normalize our distribution so that the x values represent connectivity, we divide by the number of edges possible (after we convert to the standard deviation by taking the square root:

$$\sigma_{norm}(|E(G_{inst})|) = \frac{\sqrt{Var(Binom(E_{max}, .5))}}{E_{max}} = \frac{\sqrt{1/4 * E_{max}}}{E_{max}} = \frac{1}{2\sqrt{E_{max}}}$$

Using these values, we can plot the relative sigmas for the distribution of graphs relative to their connectivity for both of the sets: the set of all graph

instances, and the set of all graphs as algebraic objects. Doing this we get the graph shown below.

[GRAPH HERE]

Note that the values appear to shrink in their disparity, a remarkable observation, which gives us an idea of how the two sets/distributions differ with respect to the number of edges they produce as a function of N. Note that this doesn't tell us about other properties of the sets of graphs. Similarity in the distribution of the number of edges does not equate distributional symmetry. However, what we can see is that the Gilbert graph generator models the number of edges successfully for large values of N.

5.3 Specifically Disadvantaging Highly Automorphic Graphs

Dominant models of random graph generation specifically preference graphs with fewer non-trivial automorphisms over graphs that have many automorphisms. This flaw is most glaring when discussing average or worst case running time over the 'random' class of graphs. Many algorithms make exciting claims about their performance on 'random' graphs, but we will show how theoretical and practical analysis of performance changes dramatically under different random graph generators.

5.3.1 Two Quick Justifications

To get an initial sense of this problem, we will first examine the relative probabilities of selecting a Co-Cycles graph from the set of graphs of size 10. Since we know all 116 Co-Cycles graphs over 10 vertices, we can (without much computational effort) calculate the relative probabilities of generating a Co-Cycles graph under the standard (Gilbert) model and under the idealized model. The results are not shocking, but a notable difference is apparent:

$$P(CoCycles(G)|G \in Gilbert) = \frac{\sum_{g \in CoCycles}^{116} M_{Reps}(g)}{NMatrices} = \frac{260820000}{3.5184 \times 10^{13}} = 7.4130{\times}10^{-6}$$

$$P(CoCycles(G)|G \in Gilbert) = \frac{NCoCyclesGraphs}{NGraphs} = \frac{116}{12005168} = 9.6625{\times}10^{-6}$$

Though these are not exorbitent differences, it should be noted that there is a difference of 23%, meaning that by using a Gilbert random graph generator, you are 23% less likely to observe a co-cycles graph as a result of generating one random graph than you would be via an idealized generator. In this case, the difference between the generators is clearly appreciable, but not severely limiting.

As a second example, we will compare the most and least likely graphs to be observed under either random graph model. In an idealized random graph generator, all graphs have equal probability of being selected. Thus, the ratio of the most likely to least likely is exactly 1.

In a Gilbert random graph generator (abstracting over p, as discussed earlier), graphs that are fully automorphic (where the automorphism set contains all one-to-one mappings), are the least likely to occur. One such graph is the fully connected graph, or the complete graph (K_N). This is simply because the number of matrices which represent them is given by M_{Reps}, and the probability of selecting a the complete graph is thus:

$$P(G = K_N | G \in Gilbert) = \frac{M_{Reps}(K_N)}{NMatrices} = \frac{N!/|Aut(K_N)|}{2^{.5N(N-1)}} = \frac{N!/N!}{2^{.5N(N-1)}} = \frac{1}{2^{.5N(N-1)}}$$

However, if we examine a graph with only the trivial automorphism (G_{TA}):

$$P(G = G_{TA} | G \in Gilbert) = \frac{M_{Reps}(G_{TA})}{NMatrices} = \frac{N!/|Aut(G_{TA})|}{2^{.5N(N-1)}} = \frac{N!/1}{2^{.5N(N-1)}} = \frac{N!}{2^{.5N(N-1)}}$$

Thus, the ratio of these probabilities is always going to be $N!$.

These two examples give us a really important message: the difference between the Gilbert and Ideal models is frequently substantive, but its effects can be mild (as in the case of co-cycles graphs), or extreme (as in the case of selecting a perfectly automorphic graph). What largely determines the degree of this distortion is the *average number of automorphisms in the set that is being examined*. The larger this number, the more distortionary the Gilbert Model is when compared to the Idealized model.

5.3.2 Theoretical Average Case Comparison

Consider a standard canonical labeling algorithm. This algorithm has two abstract components, one which correctly places vertexes into Similar Vertex Sets (SVS), and another which finds a canonical labeling given an accurate SVS partition. If we take the first part of this algorithm as given, and assume that it can be computed in polynomial time (a reasonable assumption, as discussed in the section on SVS), then the running time of the algorithm is contingent upon the number of matrices we need to evaluate against some canonical property. One common property for canonization is selecting the adjacency matrix which is the lexicographically smallest (or largest) representation of the graph. It is reasonable to assume that the second half of the algorithm dominates the running time of the overall algorithm, as it is the piece that is inherently exponential based on the number of representations of the matrix. Thus, we can express the running time of the overall algorithm in terms of $O(|Aut(G)|)$, as this is the number of possible labelings we will have to examine to find our canonical one.

The selection of this simplified algorithm for analysis is not an accident: we have chosen it because the number of matrix representations ($M_{rep}(G)$) for a given graph G is $\frac{N!}{|Aut(G)|}$. Thus, if we are considering the average running time of this canonization algorithm ($\overline{T}_{Gilbert}$) over the set of all graph instances ($G_{Inst} = G(A)$ for $A \in \{\{0,1\}^{N\times N}\}$) (i.e. using the standard models for random graph generation), we come to different conclusions if we consider the set of possible graphs G_{Inst}, versus if we consider the set of all graphs as algebraic structural objects (G_{Alg}).

$$O(\overline{T}_{Gilbert}) = \frac{\sum_{g\in G_{Inst}} O(T(g))}{|G_{Inst}|}$$

$$O(\overline{T}_{Gilbert}) = \frac{\sum_{g\in G_{Alg}} O(T(g)) * M_{Reps}(g)}{|G_{Inst}|}$$

$$O(\overline{T}_{Gilbert}) = \frac{\sum_{g\in G_{Alg}} |Aut(g)| * \frac{N!}{|Aut(g)|}}{|G_{Inst}|}$$

$$O(\overline{T}_{Gilbert}) = \frac{\sum_{g\in G_{Alg}} N!}{2^{.5(N^2-N)}}$$

$$O(\overline{T}_{Gilbert}) = \frac{|G_{Alg}| * N!}{2^{.5(N^2-N)}}$$

This property might not look like it tells us much, but we can actually approximate what it looks like for different values of N, since we have the frist nineteen values of this sequence from OEIS sequence A000088 CITE. This data is shown below. CITE

However, if we attempted to select a graph from the set of all graphs of that size, we would find that:

$$\overline{T}_{Ideal} = \frac{\sum_{g\in G_{Alg}} O(T(g))}{|G_{Alg}|}$$

$$\overline{T}_{Ideal} = \frac{\sum_{g\in G_{Alg}} |Aut(g)|}{|G_{Alg}|}$$

$$\overline{T}_{Ideal} = \text{Average Number of Automorphisms over Graphs}$$

Some sample numbers to give you the scale of these disparities is given below [9].

The takeaway here is that specifically disadvantaging a class of graphs (namely highly automorphic graphs) warps our analysis of running time in a substantive way, not only for fringe algorithms, but for well studied algorithms too. We can show this through the theoretical proof as shown above, but we can also show it through data describing actual algorithm performance.

5.3.3 Practical Average Case Comparison

The NAUTY package, written in CITE by CITE, is one of the quickest algorithms for canonical labeling CITE. Though its performance is weaker on some graphs (like the Miyazaki, for example), in general, its canonical labeling algorithm is widely regarded and frequently used.

I ran the NAUTY canonization algorithm on batches of graphs with different numbers of vertices. To generate the first sets of batches, I randomly selected graphs using the Gilbert model with probability 1/2, and varying values of N. To generate the second batch, I first set N, then generated a random variable M to describe the number of edges (also with internal binomial probability 1/2), then selected with uniform probability from the set of all graphs with N vertices and M edges. I ran each batch set ten times, to get a scatterplot describing the time complexity growth, and understand its variance. The logarithmic graph of my results is shown below. [CITE].

We can see from these figures that not only does NAUTY have slower performance on 'Idealized' random graphs, but its running time increases at a higher constant value. If we set these numbers to a log-lin regression, we see that this growth rate is heavily supported by the data.

Both in theory and in practice, distinguishing between 'random graph' and 'random graph instance' has significant implications. It is very possible that misuse of the Gilbert model has understated the true average and worst case time complexity for a number of algorithms.

5.3.4 Should We Never Use the Gilbert Model?

The short answer to this question is no, the Gilbert model is remains relevant (even dominant) even when we embrace these flaws. The primary argument for the Gilbert model is not its application to questions of theory, *but questions of practice and natural occurrence.* Most applications of graph theory to world problems are based around structures that are naturally arising, without centralized planing or design. It is much more likely that some graph that is found in the world will be non-automorphic than have some kind of inherent and difficult to describe complexity. This is because if we look at graph building as a process [CITE GILBERT], or as a probabilistic construction [CITE EDRODS], then some outcomes are more likely than others. This is well in line with the pervasive assumptions of the Gilbert model.

Rather, what we have been criticizing is the *application* of the Gilbert model to questions about theoretical algorithmic complexity. In graph theory, we hold a meaningful distinction between graphs and graph instances, a distinction only made at the theoretical level. It is thus inappropriate to use a model which ignores this distinction, and ignores it in such a way as to specifically disadvantage certain classes of graphs. This allows theoretical

arguments about graph algorithms to underweight what we consider 'hard' cases, and overweight what we would generally consider 'easy' cases.

This distinction is important. In practice, graphs that arise in the world are likely to be selected randomly from the set of graph instances. However, in graph theory, where we consider isomorphic graph instances to be fundamentally equivalent, the assumptions of the Gilbert model are antithetical to other assumptions of graph theoretic research.

5.4 Measuring and Comparing Random Graph Models

Our mandate suddenly becomes: 'well, what can we do which is better?' Establishing an intuition to answer this question requires us first to understand what we would think of as 'ideal', and see if we can approach that model.

5.4.1 An Ideal Random Graph Model

To allow our comparisons to be explicit, we will use the same nomenclature to describe an idealized generator as we do with the Gilbert model, so that they are directly comparable. Thus, our ideal random graph model will take two input parameters, N and p, where p describes the probability of getting any given edge existing. Note that this explicitly breaks from the idea of edge independence that is assumed in the Gilbert model. Rather, we are assuming that the probability of any given edge existing is p if we have no knowledge about the rest of the graph. We will define our 'Ideal' generator as one which first selects a number of edges M from the binomial distribution with E_{Max} trials and trial success probability p. After selecting M as an observation from this random distribution, we will select a graph from the set of all graphs of N vertices and M edges with uniform probability. To accomplish this task, we will enumerate all graphs of that size, and select from that fully defined set.

The flaw in this scheme is obvious. Our 'Ideal' generator is only possible by this methodology so long as you can enumerate all graphs of a given size. There are 2463780925312500452438300749143276 8 non-isomorphic graphs over 19 vertices, so this is not remotely sustainable methodology for a random generator. If we cannot rely on an ideal graph generator, are we doomed to stick with the Gilbert model? Or, could we create a model that is better than the Gilbert model (closer to our ideal), while being computationally reasonable?

5.4.2 An Equivalent Ideal Random Graph Model

Another way to view our equivalent random graph model is as a overlaid construction on top of a standard, Gilbert, random graph model. Since we know that the Gilbert random graph model produces a graph with a probability proportional to its number of automorphisms, we can actually create a random graph generator which is equivalent to our idealized model by systematically throwing out graphs with probability inverse to their probability of showing up. The algorithm which generates to this model is not guaranteed to halt, but does simulate our idealized random graph generator, even over large values of N.

```
while true:
        g = gilbertRandomGraph(N, P)
        na = |Aut(g)|
        p = Math.rand(0, 1)
        if (p < (na / N!))
                return g
```

Here we find a gilbert random graph, and calculate the number of automorphisms that it has. That gives us knowledge of how many times we would expect it to occur in a sample of $2^{E_{max}}$ random graph instances, namely $N!/na$ times. Thus, we weight any randomly observed random graph instance by the inverse of this value. This equalizes the probability of getting any graph, regardless of its number of automorphisms.

This methodology is, unfortunately, equally unsustainable. This is an example of a geometric process, where we will have to go a certain number of trials before finding a success and stopping. The probability of a success on any given trial is $p = \frac{|G_{alg}|}{|G_{inst}|}$, which approximates the $p = fact^{-1}(N) = \frac{1}{N!}$ growth function. Thus, for even a small number of vertices, the expected number of trials that need to be generated before a success is found is very large, as it is given by $1/p = N!$. For example, if $N = 10$, we would expect to have to examine 3.6 million graphs before finding one which is returned successfully.

This too, is not a sustainable methodology for random graph generation.

5.4.3 Establishing a Methodology for Random Graph Generator Evaluation

So what can we do that is better? Answering that question required some really deliberate thought. I knew lots about what I didn't want to do:

- I didn't want to use subjective measures to support one random graph generator over another.
- I didn't want to write generators and then use collected data to support them.

- I didn't want to focus in on singular ways of looking at the quality of the graphs that I generated.
- I didn't want to allow my own intellect and lack of creativity be a limiting step on the way to a better generator. The process that I decided on attempted to address all of these concerns, and was designed to avoid the pitfalls of bad computer science: overfitting, statistic selection, and rigidity.

I decided on a wide portfolio of statistics each of which is calculated over a sets of graphs. These statistics are inherently heuristic, but each attempts to capture some property of the graphs that are generated, or some property of how the set looks as a whole. Each of the statistics I decided on is meant to describe some property of a set of graphs, but it does not establish an 'idealized' or theoretical value for that statistic. Though for several of these metrics we know how to express the ideal/expected probability, that is not always the case (or not always known to be the case0. These metrics serve as a way for us to identify and name ways that random graph generators fail us, and challenge us to do better, to compare a proposed solution to an ideal or standard solution in quantifiable and concrete terms.

The set metrics is described in the next section, and they vary in computational complexity and broad descriptiveness. I set up a (pardon the brag) BEAUTIFUL system for calculating these statistics over arbitrary datasets. I created an idealized random generator (as described above), and a Gilbert random generator, and verified the accuracy of the results produced through this computational system over the initial results from each.

Most importantly, I set up these metrics, this system, and the baseline Gilbert/Ideal metric values before I wrote a line of code which randomly generated graphs. This is really important to me, because it frees the results of this work from the publication and statistic selection bias that plagues research everywhere.

5.4.4 Graph Set Metrics

Below are the metrics that I compared across graph sets generated by different random graph generation procedures. Though I calculated some more statistics for personal use and discovery, the ones included below are the 'single result' statistics. Others were distributional properties that were not easily captured and compared (except through distributional goodness of fit tests, which only give us an idea of alikeness, not of directed difference). The reliance on singular metrics as a means of comparison allows us to evaluate systems of random graph generation automatically, and come up with similarity tests that are divorced from our intuition and hypotheses.

Described below are each of the metrics that are calculated over every set of random graphs that were generated. Each is labeled a statistic (implying

a singular value used to judge the generator) or a distribution (connoting that it was not used in the systematized evaluation of generators). Each is listed under its acronym, which can be found throughout my code and tests.

5.4.5 Graph Set Metric Reference, Coding and Description

- ADIAM - Set Statistic - Average Graph Diameter - Take the diameter of each graph, and average over all graphs in the set
- ADSM - Set Statistic - Average Degree Sequence Mean - take the mean of degree sequence of each graph, and average that across all graphs
- ADSMD - Set Statistic - Average Degree Sequence Median - take the median of the degree sequence of each graph, then average that across all graphs
- ADSMN - Set Statistic - Average Degree Sequence Minimum - take the degree of the least connected node in each graph, then average that across all graphs
- ADSMX - Set Statistic - Average Degree Sequence Maximum - take the degree of the most connected node in each graph, then average that across all graphs
- ADSR - Set Statistic - Average Degree Sequence Range (i.e. Find the difference in degree between the most and least connected node, and average over all graphs in the set
- ADSV - Set Statistic - Average Degree Sequence Variance - take the variance of the degree sequence of the graph, then average across all graphs
- ALNA - Set Statistic - Average of the Logarithm of the Number of Automorphisms - Find the number of automorphisms that each graph has within the set, take the logarithm of each, then average across the logged results
- ANA - Set Statistic - Average Number of Automorphisms - find the number of automorphisms for each graph, then average that across the set
- ANCC - Set Statistic - Average Number of Connected Components - take the number of connected components of each graph in the set, then average across all of them
- ANQUAD - Set Statistic - Average Number of Quadrilaterals - The Average number of non-trivial (non-edge repeating) quadrilaterals across all graph instances within the graph set.

- ANR - Set Statistic - Average Number of Repeats - The average graph, when selected from the set in question, will have this expected number of equivalent instances in the set
- ANTRI - Set Statistic - Average Number of Triangles - The Average number of triangles across all graph instances within the graph set.
- CP - Subset - Co-Cycles Graphs - The intersection between the set of all $V = 10$ co-cycles graphs and the given set
- DIAMV - Set Statistic - Variance in the Diameter of Graphs - calculate the diameter of each graph, and calculate the variance of the set as a whole
- DSMDV - Set Statistic - Degree Sequence Median Variance - Take the median degree of each graph, and calculate the variance over all graphs in the set
- DSMNV - Set Statistic - Degree Sequence Minimum Variance - Take the minimum degree of each graph, and calculate the variance over all graphs in the set
- DSMV - Set Statistic - Degree Sequence Mean Variance - Take the mean degree of each graph, and calculate the variance over all graphs in the set
- DSMXV - Set Statistic - Degree Sequence Maximum Variance - Take the maximum degree of each graph, and calculate the variance over all graphs in the set
- DSVV - Set Statistic - Degree Sequence Variance Variance - Take the variance of the degree sequence of each graph, and calculate the variance over all graphs in the set
- FC - Set Property - Frequency Count - The number of isomorphic graph instances associated with each one of the graphs in the set
- FFC - Set Distribution Counts - Frequency Count Counts - The counts that correspond to FFVs, the frequency with which graphs show up in our random set
- FFV - Set Distribution Bins - Frequency Count Bins - The discrete values for which there exist graphs in our set that poses that number of instances in the set, this is really just a bin count for FC
- MLNA - Set Statistic - Median of the Logarithm of the Number of Automorphisms - Find the number of automorphisms that each graph has within the set, take the logarithm of each, then find the median across the logged results

- MNR - Set Statistic - Maximum Number of Repeats - The graph in the set that was selected the most times was selected this many times
- NA - Graph Statistic - Number of Automorphisms - Counts the number of automorphisms that a graph has by finding the number of unique matrices that describe a graph
- NAB - Set Distribution Bins - Number of Automorphisms Bins - The unique values of NA, to provide histogram alongside NAC
- NAC - Set Distribution Counts - Number of Automorphisms Counts - The counts of the NAB, as in a histogram
- NCCV - Set Statistic - Variance in the Number of Connected Components - take the number of connected components of each graph in the set, then take the variance of the set
- NCP - Set Statistic - Number of Co-Cycles graphs - Same as Co-cycles graphs, this only applies to graphs of size 10 and larger, counts the number of Co-Cycles graphs (which tend to be highly automorphic) in the overall set of graphs
- NE - Set Statistic - Number of Edges - the average number of edges in the graph set. This should be the same across generators, as we are specifying p, and asking for a large sample size
- NR - Set Statistic - Number of Regular Graphs - the number of regular graphs in the graphset
- NUG - Set Statistic - Number of Unique Graphs - The proportion of graphs within the set that are only generated once
- ODSPL - Set Statistic - Overall Degree Sequence Poisson Distribution Lambda - Find the set of all of the degrees of all of the graph, then fit a poisson distribution to this distribution. Report the lambda that defines this poisson distribution
- PCONN - Set Statistic - Probability of Connectivity - take the number of fully connected graphs, and divide by the total number of graphs in the set
- PDL - Set Statistic - Poisson Distribution Lambda - Take the distribution of the number of times that each graph is represented by an instance within a graph set, and set this distribution fit to a poisson distribution, reporting its lambda
- PQRA - Set Statistic - Percentage Quazi-Regular A - Percentage of graphs where the degree sequence range is less than or equal to 1

- PQRB - Set Statistic - Percentage Quazi-Regular B - Percentage of graphs where the degree sequence range is less than or equal to 2
- PQRC - Set Statistic - Percentage Quazi-Regular C - Percentage of graphs where the degree sequence range is less than or equal to $sqrt(N)$
- PTRIL - Set Statistic - Probability Triangle-less - the Probability that a graph instance selected randomly from the graph set is triangle-less.
- SQNR - Set Statistic - Sum of Squared Number of Repeats - A measure of how dispersed the distribution is, calculated as $FFC\dot{(}FFV^2)/nGraphs$
- UG - Set Property - Unique Graphs - The canonical form of all of the graph set's graphs, with duplicates removed (if they existed)
- VLNA - Set Statistic - Variance of the Logarithm of the Number of Automorphisms - Find the number of automorphisms that each graph has within the set, take the logarithm of each, then find the variance of the logged results

5.4.6 Establishing Baselines for Graph Set Metrics

Once we set up these metrics, we need to find a way to systematically compare them within our desired context: building a random graph generator which models an idealized random graph generator. To do this, I established baseline understandings for each one of the metrics, over a problem 'domain':

- Number of Graphs - I decided to operate over sets of 1000 graphs. It allowed us to get reasonable numbers for standard deviations and not use an excess of CPU time
- Number of Vertices - I baselined the metrics over graphs of size 4 to size 10. This allowed us to see what the metrics looked like on an overrepresented graph set, and on a sample set.
- Probability of Edges - I used p from the set $[.1, .2, .3, .4, .5, .6, .7, .8, .9]$. I figured that focusing on sparse, dense or intermediate graphs might introduce biases that I hadn't accounted for, so I decided for the most generic set possible.
- Number of Trials - For each one of these baseline scenarios, I performed 30 trials

Thus, in the end, there were $2 \times 9 \times 7 \times 30 = 2780$ sets of 1000 graphs that were used to create our baseline metrics.

For each one of the statistics we calculated the mean and standard deviation of the sampled metrics over both algorithms. That resulted in two

sample means and sample standard deviations, for the ideal and gilbert graph models: $\overline{x_I}, \overline{x_G}, s_I$ and s_G. Using these values, I came up with a simple and intuitive way to measure a new random graph generator, given its value for $\overline{x_N}$ and s_N and the number of samples that were used to compute each, n_N. This methodology compares the t values in two tests of significance for two unknown means and unknown standard deviations. In a test of unequal unknown means and standard deviations, the slightly modified t-statistic is given by:

$$T(A,B) = T(\overline{x_A}, \overline{x_B}, s_A, s_B, n_A, n_B) = \frac{|\overline{x_A} - \overline{x_B}|}{\sqrt{(s_A^2/n_A + s_B^2/n_B}}$$

And our scoring mechanism for a metric and generator is given by:

$$\text{Score for Metric M} = -1 * \min\left[\ln\left(\frac{T_M(I,N)}{T_M(I,G)} + e^{-10}\right), 10\right]$$

Lets break down this scoring mechanism:

- First off, $T_M(A,B)$ is strictly greater than or equal to zero (and all of our metrics have non-zero $T_M(I,G)$ values), thus the fraction in the logarithm is defined and in the range $[0,\infty)$. When we add e^{-10}, we shift that range up to $[e^{-10},\infty)$, meaning that the logarithm's value will be bounded between $[-10,\infty)$
- Next, we take the minimum of this score and 10, to bound the individual score between $[-10,10]$.
- Finally, we invert the values with a -1, so that a score of -10 corresponds to a metric where the new random graph generator performs incredibly poorly, while 10 corresponds to a metric where the new random graph generator is indistinguishable from ideal (with respect to the differences setup by the baselines).
- The resulting metric has the helpful property that the baseline metric is at (or ever so slightly below) 0, so that a score of 0 marks an equivalence in quality to the Gilbert random generator.

Once we calculate individual metrics, we can calculate the score for a generator as the sum over the number of metrics that are calculated for it, which includes multiple values of p and n:

$$\text{Score for Generator} = \frac{1}{N_{Metrics}} \times \sum_{m}^{Metrics} Score_m$$

in this metric, higher scores are better and a score of 0 corresponds to a generator that is exactly as bad as the Gilbert Generator.

This is a good mechanism because it scores a value on a metric based on how close it is to the ideal mean, but also compares any deviancy to the deviancy observed between the ideal and standard (Gilbert) generators.

5.4.7 Establishing Baselines for Graph Set Metrics

The baselines for the graph set metrics are all stored in one file for convenience (Thesis/Matlab/Alternative Generator/Data/baselineMetrics.mat). As mentioned above, they are for the tenth-probabilities, over graphs of size 4 to 10. The baselined metrics were constructed using thirty random trials, each containing 1000 random graphs. In the baseline file, the mean, standard deviation and number of samples is given.

An appendix to this thesis lists the graph set baselined metrics, along with their associated uncertainties, for all values of N and P. Since there were 7 tested values of N, and 9 tested values of P, and 35 metrics, over two different generators for random graphs, there are a total of approximately 4,000 component statistics in this baseline file.

5.5 Alternative Ideas for Random Graph Modeling and Creation

Now that we have a mechanism for quantifiably adjudicating the quality of a proposed random graph generator, we are tasked with creating different ideas for random graph generation and evaluating them on this basis. Discussed below are each of the algorithms that I built for Random Graph Generation and Processing. They are not presented in any particular order. Each is prefixed by the string that it is represented as/by throughout the code.

5.5.1 MinusOne - A Weighted Automorphic Subgraph Generator

MinusOne has a simple methodology. If you are generating a graph with N vertices and with probability of edge creation p, start by creating a graph H of size $N+1$ via the gilbert model. Then, generate all $N+1$ of H's single vertex deleted subgraphs, store each in a set we will call $VD(H)$. Lets define an automorphic metric AM which is defined by the following formula, finding the QSVSes, and take the product of the factorial of their sizes.

$$AM(g) = \prod_{i \in QSVSes(g)} |QSVS_i|$$

Finally, over each member of the set $g \in VD(H)$ calculate the automorphic metric, and select that graph with the probability:

$$P(g|H) = \frac{AM(g)}{\sum_{h_i}^{VD(H)} AM(h_i)}$$

5.5.2 MinusOneA - Less Automorphic Version of MinusOne

MinusOneA has a methodology with closely matches that of MinusOne. The only difference is the automorphic metric function, which is the logarithm of the one proposed above (plus a constant factor).

$$AM(g) = 1 + \log_2 \Big[\prod_{i \in QSVSes(g)} |QSVS_i| \Big]$$

5.5.3 MinusOneB - Interspersing Some Gilbert Random Graphs

MinusOneC has an identical methodology with closely matches that of MinusOne, with one caveat. With a probability selected based on p, the algorithm chooses between picking a graph from a MinusOne distribution and a Gilbert distribution:

$$MinusOneB(n,p) = \begin{cases} Gilbert(n,p) & \text{if } rand < .5 * (p) * (1-p) \\ MinusOne(n,p) & \text{otherwise} \end{cases}$$

5.5.4 MinusOneC - Less Automorphic Version of MinusOne

MinusOneC has a methodology with closely matches that of MinusOne. The only difference is the automorphic metric function, which is the square root of the one proposed above.

$$AM(g) = \sqrt{\prod_{i \in QSVSes(g)} |QSVS_i|}$$

5.5.5 BuildingA - An Iterative Graph Generator

Building Algorithms are processes by which edges are added individually to a matrix (given a 'probability matrix'), and then a new probability matrix is generated from the current iteration of the formative graph.

```
nEdges = binomalRandom(N, P)
A = zeros(N)
P = (ones(N) - eye(N))/(N*(N-1))
while nEdges > 0:
        A = addEdgeFromProbabilityMatrix(A, P)
        P = generateProbabilityMatrixFromPartialGraph(A)
        nEdges = nEdges - 1
```

From this, it is clear that the only place that one can really modify this algorithm is in the generation of the P matrix from the partially generated graph A. BuildingA used a simple algorithm to do this. It created an edge metric $EM(v_i, v_j)$ to describe the odds of creating an edge between vertices

v_i and v_j (here, DS refers to the degree sequence of A as it exists at each iteration):

$$EM(v_i, v_j, A) = (i - j \neq 0) * (1 - A[i, j]) * \Big[1 + 2max(DS) - DS[i] - DS[j]\Big]$$

And with the probability of selecting a given edge for creation being:

$$P(\,(v_i, v_j)\,|A) = EM(v_i, v_j, A) / \sum_{x=1}^{N} \sum_{y=1}^{N} EM(v_x, v_y, A)$$

5.5.6 BuildingB - An Inverse Differential Graph Generator

The BuildingB random graph generator operates identically to BuildingA, with a change in the individual edge metric EM_A:

$$EM(v_i, v_j, A) = (i - j \neq 0) * (1 - A[i, j]) * \Big[1 + 2max(DS) + DS[i] + DS[j]\Big]$$

5.5.7 Cloning Model

One of the thoughts that I first had was on the way that Co-Cycles graphs tended to be highly automorphic, and the way that internal structure of each set within the SVSes was almost always fully automorphic. Remember that a fully automorphic graph (or in this case, subset of vertices) means that within its self contained structure, there exists an automorphism which maps any one vertex to any other (not implying that all proposed mappings are automorphisms, but only that a mapping exists that pairs any pair of vertices within the set. To try to create a model that is likely to produce this kind of internally reflective behavior, I broke the task of generating a graph down into two subtasks. First, it creates K disjoint graphs, each of which is fully automorphic/similar, and with the total number of vertices over all of these smaller graphs equalling the target number of vertices the generator is looking to produce. Then, it constructs a set of connections between each pair of the K subgraphs. With fixed probabilities, it constructs these connections to be either fully symmetric (i.e. preserving symmetry for both the right and left hand side of the connection), partially symmetric (just preserving one side's symmetry), or asymmetric (differentiating both sides).

This model was not successful at producing graphs which are broadly in line with the aims of the chapter (i.e. idealized random graphs), but it was very successful at producing co-cyclic graphs, with regularity far exceeding that of either generator. This initial attempt's results are detailed below, alongside some different results given different parameter values (for the fixed probability of edge symmetry creation.

Chapter 6

Further Questions

6.1 Reconstruction Hypothesis

The Reconstruction Hypothesis is a claim in theoretical computer science that is very likely true, but has not been proven to be. The hypothesis revolves around the notion of a 'deck' of a graph G with N vertices. The Deck of G ($Deck(G)$) is a (possibly multi-)set of N graphs, each of size $N-1$, all of which are created by deleting one vertex from the graph G.

Two Decks D_1 and D_2 are said to be isomorphic if there exists a one to one mapping of their elements such that every graph $g_i \in D_1$ is mapped to an isomorphic graph $g_j \in D_2$. The reconstruction hypothesis is the claim that two decks are isomorphic if and only if their graphs are isomorphic, or in formal terms:

$$Deck(G) \cong Deck(H) \iff G \cong H$$

Though this hypothesis seems pretty iron clad, there is one known violation, two non-isomorphic graphs over two vertices have congruent decks. Can you find them? (hint: there are only two graphs over two vertices). Thus the actual hypothesis only makes its claim for $N \geq 3$. As we get to larger and larger values of N, it would seem that the hypothesis becomes harder to refute via a counter example, both because the amount of information that goes into the physical representation of the deck increases cubically, and because the unknown pieces of the graph shrink relative to the consistent amount provided by each card in the deck.

However, the nature of the hypothesis is so hard to verify, (and the number of graphs grows so large so quickly) that its verification has only been done up to graphs of 11 vertices, by McKay in 1999.

One would think, that with today's technology and resources, we ought be able to improve upon that bound, and increase N to 12 (or find a counterexample therein). That is hopefully the site of some future work.

6.1.1 Parallelizing a Reconstruction Check to 12 vertices

One thought that I had in pursuing this question was: how can we use the massively available scalability of today's cloud based world to extend this problem past 11 vertices? If McKay can do 11 vertices in 1999, one would think that in 2016 we ought be able to do 12 (despite the 160 fold increase in the number of graphs). Here is the algorithm as I would design it.

First, lets define a simple algorithm for checking for equivalent decks. Given a graph G, create all of its vertex deleted subgraphs, canonically label and encode each, then append them in lexicographical order. This string is relatively short. For an original graph with 12 vertices, there would be 12 graphs of size 11, if we strip off the first character, which is the same across all graphs of the same size, the encoding is only 10 characters long per graph, meaning that a deck could easily be encoded in only 110 characters (or, 440 bytes). The running time for this algorithm (using NAUTY as a sub-call from a C function (with hardcoded values of 11 and 12 vertices)) takes approximately XXX milliseconds, on average.

However, we need to be smart about when we compare these decks. We cannot use a database to store them (as 440 bytes * 160 billion graphs = 71TB) so we need to generate all graphs which could share a deck at the same time and check at a local level. This is how we would go about doing that:

If two graphs G and H over 12 vertices are going to have isomorphic decks, then there exist 12 graphs of size 11 such that each is a vertex deleted subgraph of both. Given a graph G, lets call the seeds of G to be the vertex deleted subgraphs such that the vertex that was deleted is the one with the maximum degree. There can be multiple seeds of G if the graph's maximal degree is repeated multiple times in its degree sequence. It has been shown that across all graph instances, the number of graphs with a maximal degree incidence greater than one is a negligible proportion of all graphs. Thus, our algorithm leverages this fact to do seeded checking for deck isomorphism.

As a side note, it should be obvious that no graphs could violate the reconstruction hypothesis if they have a fully connected node or a fully disconnected node.

Here is the algorithm: a computational node is delivered a graph of size 11, called S. If the maximum degree of S is K, then the range to search is deemed $[K, 10]$ (as remember, no violating graph can have a fully connected node with degree 11). For each integer $k \in [K, 10]$, we generate $choose(11, k)$ augmented graphs of S, for each of the possible connection patterns. Before we move on to the next value of k, we check each of these graphs for deck isomorphism using the technique provided above.

From the small amount of work (unoptimized) that I have done on implementing this checking system so far, it appears that the running time of this algorithm would be about a millisecond per graph of size 11 checked.

This would imply that computation of this algorithm over all graphs of size 11 (which would check all graphs of size 12) would take about 12 days to complete, running on the single workstation I was using. If we distribute this workload across many servers, as we could easily do by generating different sets of graphs to check on each (i.e. using NAUTY geng with different values of e), I imagine that this running time could be significantly decreased.

I hope to be able to pursue this verification once my thesis is submitted and I have nothing to do until graduation.

6.1.2 Potential Reconstruction Procedure: The Triangle Inequality

One of the first questions I asked when I came across the reconstruction hypothesis was: can Cycles be a part of a solution to this problem? Can we create an algorithm which leverages properties of cycles in order to attempt to reconstruct a graph, or at least place limitations on the possibilities for reconstruction? I puzzled away at this for a while, and came up with an interesting way of incorporating cycles into a reconstruction procedure. I will apologize in advance: this section is particularly notation heavy, and I don't think there is anything that can be done to avoid that.

Lets start with a thought experiment. In a graph G, with vertices x and y, the number of cycles which pass through a given vertex (lets say vertex x) is equal to the number of cycles which pass through the vertex x and the vertex y, plus the number of paths that pass through the vertex x but not the vertex y. This is true for all lengths of cycles we are considering. Thus, we can make a claim of this nature using vectors of length P as our operating unit. We will refer to these vectors by the notation $[x]_G$, denoting the vector of the number of cycles that pass through vertex x within graph G, and we will use $[x\bar{y}]_G$ to refer to the number of cycles which pass through vertex x but not through vertex y in graph G.

In this notation, the above claim translates to:

$$[x]_G = [xy]_G + [x\bar{y}]_G$$

though this is not revolutionary, when we start to mess around with our graph and its subgraphs, this simple claim can be manipulated to produce much more interesting ones. For example, in the vertex deleted subgraph $G - y$:

$$[x]_{G-y} = [x\bar{y}]_G = [x]_G - [xy]_G$$

We can state the same thing about vertices y and x:

$$[y]_{G-x} = [y]_G - [xy]_G$$

If we subtract these two equations together, we get:

$$[y]_{G-x} - [x]_{G-y} = [y]_G - [x]_G$$

This is something we will call the 'tuple' identity. If we have some proposed mapping between the deck cards ($G-y$and$G-x$ in this case) and their proposed associated vertices in their deck cards, we can verify this via the difference between their cycles vector values over the full graph G. A careful reader will now interject: but we don't know the full graph G! If we did, we wouldn't be mucking around with all of this!

We can get around this fact by employing three such 'tuple' identities, to construct a much stronger (and not too information needy) test for possible deck-vertex mappings:

$$[y]_{G-x} - [x]_{G-y} = [y]_G - [x]_G$$

$$[z]_{G-x} - [x]_{G-z} = [z]_G - [x]_G$$

$$[z]_{G-y} - [y]_{G-z} = [z]_G - [y]_G$$

Adding the first and third, while subtracting the second yields us:

$$[z]_{G-y} - [y]_{G-z} + [y]_{G-x} - [x]_{G-y} - [z]_{G-x} - [x]_{G-z} = 0$$

$$([z] - [x])_{G-y} + ([x] - [y])_{G-z} + ([y] - [z])_{G-x} = 0$$

This is the proposed 'triangle equality', which needs to hold for every triplet of vertices within a proposed mapping of vertex to card.

If I had more time (if only, if only), I think I could transform this into a set of boolean predicates which could begin to describe a highly discriminatory set of constraints on the possible reconstructions. The key thing here is that the overall number of triplets for which this can hold is very small, but also sortable. In other words, if you have an idea for a mapping between to vertices and cards, the third could be found quickly by using the remaining constraint of the vector equality dictated by the triangle inequality. Alas, this is as far as I got with this promising use of the Cycles invariant.

Bibliography

[1] Jin Akiyama. The graphs with all induced subgraphs isomorphic. *THE BULLETIN OF THE MALAYSIAN MATHEMATICAL SOCIETY*, 2, 1979.

[2] Laslo Babai, Paul Erdos, and Stanley M Selkow. Random graph isomorphism. *SIAM Computing*, 9(3), 1980.

[3] B. Bollobas. Mathematical results on scale-free random graphs. *Handbook of Graphs and Networks*, 2003.

[4] Keith M. Briggs. Number of unlabeled undirected graphs up to 75 vertices.

[5] Peter J Cameron. Automorphisms of graphs. Queen Mary, University of London, apr 2001.

[6] Sourav Chatterjee and Persi Diaconis. Estimating and understanding exponential random graph models. *The Annals of Statistics*, 41(5):2428?2461, 2013.

[7] Dragos Cvetkovic, Peter Rowlinson, and Slobodan Simic. A study of the eigenspaces of graphs. *Linear Algebra and its Applications*, 1992.

[8] Tomek Czajka and Gopal Pandurangan. Improved random graph isomorphism. *Department of Computer Science, Perdue*, 2015.

[9] P. Erdos and A. Reyni. On random graphs i. *Publicationes Mathematicae*, 6, 1959.

[10] Marcelo Fiori and Guillermo Sapiro. On spectral properties for graph matching and graph isomorphism problems. *ARXIV*, 2014.

[11] D. J. Watts M. E. J. Newman and S. H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences of the United States of America*, feb 2002. http://www.jstor.org/stable/3057594.

[12] Brendan D McKay. Small graphs are reconstructible. *Australasian Journal of Combinatorics*, 15, 1997.

[13] Vclav Ndl. Graph reconstruction from subgraphs. *DISCRETE MATHEMATICS*, may 2001.

[14] Online Encyclopedia of Integer Sequences. Number of graphs on n unlabeled nodes.

[15] Sonja Petrovic, Alessandro Rinaldo, and Stephen E. Fienberg. Algebraic statistics for a directed random graph model with reciprocation. *Contemporary Mathematics*, 516, 2010.

[16] Tom A. B. Snijders, Philippa E. Pattison, Garry L. Robins, and Mark S. Handcock. New specifications for exponential random graph models. *Center for Statistics and the Social Sciences*, apr 2004.

[17] Tom AB Snijders. Markov chain monte carlo estimation of exponential random graph models, 2002.

[18] Remco van der Hofstad. *Random Graphs and Complex Networks*, volume 1. Eindhoven University of Technology, feb 2016.

[19] Marijte A.J. van Duijn, Krista J. Gile, and Mark S. Handcock. A framework for the comparison of maximum pseudo likelihood and maximum likelihood estimation of exponential family random graph models. *Social Networks*, 2009.

[20] A whirlwind tour of random graphs. Fan chung. *UCSD*, 2008.

[21] Virginia Williams. Algorithms for fixed subgraph isomorphism. CS267 Lecture 1, jan 2015.

[22] Katona Zsolt. *Random Graph Models.* PhD thesis, Eotvos Lorand University, Hungary, 2006.

Chapter 7

Appendices

7.1 Appendix A: Explanation of Code Structure

All of the code used in this project is available and version controlled on my GitHub repository, TODOHREF here.

Here is the basic structure of the project:

7.1.1 Thesis/Cardinality Analysis

Used to generate and analyze the co-cycles graphs, where they exist. The C code is largely what I did my computation in, while the Java code was usually there just for corroborating unclear results, and unit testing that my work (particularly around paths calculations and graph encodings) was correct. Within the C code, there are three projects. The first (C/deprecated) did not incorporate the trie structure or delayed evaluation, and evaluated against all matrices, (reinforcing the distinction between graphs and their instances!) and was unable to calculate up to the values that I needed because of really high run times. The second (C/updated) did incorporate NAUTY to do random graph generation, and attempted to use a Tree, but was too slow because of memory allocation and non-delayed processing (eager evaluation). The third (C/tries) is the most up to date, and was originally used solely for cardinality calculation. It uses delayed evaluation, tries, and minimal memory (less than a sixth of its predecessor). When I examined the power curves as described in chapter 3, I modified some of this code to relay depth information appropriately, which did not significantly modify its specs or the way that it runs. To see the cardinality calculations as they were when originally completed, simply visit that SHA in GitHub.

7.1.2 Thesis/data

Shared graph data amongst many different elements of the project. Two main components: geng (the NAUTY graph generator), and two bash scripts

which generate small and regular graphs. These graphs are shared among all different kinds of applications, and are created as read-only files.

7.1.3 Thesis/Documentation+Reports

Documentation for all of my work has been accompanied alongside the project itself. This includes multiple write ups that I did for my advisor throughout the semester (labeled by date) alongside an original thesis proposal, and the final result. All of these files are written in latex, so the source code is provided alongside the PDF generated by it. Some ideas and planning documents are included as well, mainly simply to maintain my thoughts and plans in centralized places. These documents are not very reflective of what ended up happening, but mark my vector trajectories at different points in time.

7.1.4 Thesis/Excel

Power curve data from chapter two, describing the interrelationship of trie-depth in co-paths calculations and the connectivity of the graph.

7.1.5 Thesis/Mupad

Algebraic manipulations, encoded and formalized using Matlab's MuPad symbolic math toolkit. Makes explicit the ideas presented in chapter two about the reconstruction of a valid graph (given the valid cycles invariant), and the reconstruction/determination of the chromatic polynomial given the cycles invariant. Please note that this code was done last summer, and is thus... messy. Some of the terminology is outdated, it is poorly commented, and was really only used to see if I could take my proof ideas about procedures of conversion and formalize them into code that performs that conversion. The software works, but is slow (symbolic manipulation of abstract types, amirite?) so don't expect to use it to perform any real computations, only proofs of concept.

7.1.6 Thesis/Matlab

This is where the majority of my second semester work was done. Though Matlab is incredibly slow (compared to the other languages I worked in over the course of the year), it provides incredibly simple and quick support for things like visualization of graphs, bulk processing and multi-threading optimization, and code reuse.

I ended up writing a LOT of Matlab code, and they are separated into nested directories which dig deep. The first level of these directories is included here.

Thesis/Matlab/alternative generator

All of the code from chapter 5, discussing the random graph generation procedures and their improvements. There are many archeological layers to this code. The deprecated folder abounds with earlier versions that did not store data in an efficient enough manner. The results of the calculations are stored in small data, which is NOT included in the git repository, but instead is zipped up and processed that way.

Thesis/Matlab/augmentation

Testing augmentation hypotheses about what happens to the Cycles invariant when we append additional vertices to it. This led to the discovery and discussion of flagging as a methodology of distinguishing between vertices.

Thesis/Matlab/automorphism vertex sets

This is the starting work that informed chapter three, on the way in which automorphism vertex sets (later called similar vertex sets) is a mechanism to describe vertex invariants, particularly Cycles within the context of increasing p.

Thesis/Matlab/cannonical

The canonical labeler as described in chapter four. The labeler is separated into generations, as the first generation I created did not employ the sorting methodology that is discussed at length in chapter four, and each iteration of the labeler has seen some incremental improvements. Note that the labelers are NOT stable across versions, so a V1 canonical labeling is not guaranteed to equal a V4 canonical labeling. Starting with V3, the labeler is assisted by a memoization function which is stored as a part of the repository in the data section. The memoization is turned off when we are timing (for obvious reasons).

Thesis/Matlab/comparison and processing

Deprecated. This is a folder that I used to use to create one-off processing pieces. I decided to make that singular use explicit through the use of the oneoffs folder.

Thesis/Matlab/constituentpaths

This section tries to grapple with the idea of the number of cycles that pass through given vertices. Like the piece of chapter six that describes an idea of how to incorporate the cycles invariant into solving the reconstruction

hypothesis, this folder deals with appending and deleting vertices from a graph and seeing what happens/what we can predict.

Thesis/Matlab/copaths search

This section was set up to test how likely co-cycles graph generation is within different random graph generators. I was not able to calculate it successfully because co-cycles graphs are so rare that generating enough graphs to have np ¿ 30 is in the billions of graphs (for a non-ideal random graph generator. This piece was abandoned and replaced by absolute probability calculations via known probabilities.

Thesis/Matlab/data

Data used by multiple subfolders. Memoization and small graph sets abound here.

Thesis/Matlab/nuralnetexplorer

A failed project to try to teach a neural net to modify a graph to be more automorphic. The question of representation deeply complicated this approach, and it wasn't clear how to resolve it, so I abandoned it.

Thesis/Matlab/ngraphs

Calculations about the number of graphs of a given size/connectivity. Produces the estimations for the distributional sigmas that are shown in chapter 1.

Thesis/Matlab/oneoffs

All of the code that actually generated results, called functions, etc. Oneoffs are my idea of how to separate out functionality from procedure, while documenting both. They are listed in chronological order, and each shows how calls are used and processed.

Thesis/Matlab/paths

Calculation of the Cycles invariant, used across the project. Note that the files in here have been modified so that all of them use the safe exponentiating procedure described in chapter 1. Additionally, the sortrows command is frequently useful for those interested in it in this context.

Thesis/Matlab/random graphs

Deprecated, before I started work on rigorous examination of random graph generators, this was the way I was looking into Random Graph Properties.

Thesis/Matlab/test graphs

Generation procedures for Miyizaki Graphs, 1-sparse and 2-dense graph generators.

Thesis/Matlab/utils

Graph utilities and timing utilities shared across all of the pieces of the project

Thesis/Matlab/visualization

Visualizations of graphs, their paths, and everything in between. Lots of results are found in this folder, small, large, intermediate and complete.

7.1.7 Thesis/Number of Graphs

A piece corresponding to Thesis/Matlab/ngraphs

7.2 Appendix B: Co-Cycles Data Set Over 10 Vertices

Below are the co-cycles graphs that are over 10 vertices. They are grouped in pairs to make them easy to distinguish, alongside their number of vertices. The NAUTY canonization (in graph6 form) alongside the Cycles canonization (also in graph6 form) is shown for each.

Co-Cycles Dataset				
N	M	CyclesCannonization	NautyCannonization	Co-Cycles Group
10	16	IIa?hONKo	IQQ@HcMEW	1
10	16	Il?GOKrN?	I‘GXOiBOw	1
10	17	IIa?X_VIw	IQ‘@OgfEw	2
10	17	Il?GOKrNG	I‘G[ACjDw	2
10	18	IIok?[RLg	IB_iCcNZ_	3
10	18	I_[haGR‘w	IOOgidwbo	3
10	18	I@Pjk?XeW	I_Ogpgrr_	4
10	18	I@QKj?]]O	IBOcSKN$\hat{?}$	4

Co-Cycles Dataset				
N	M	CyclesCannonization	NautyCannonization	Co-Cycles Group
10	19	IIq?XgjDw	IBaOXLJYW	5
10	19	I_[qCKjDw	I_\@cKVJW	5
10	19	IICkKSN?	IIaGXSV[W	6
10	19	I_ChhtKrG	I_OpmOlEw	6
10	20	I@PjcKXfg	I?SsbTefW	7
10	20	I@QKbK]^_	IAGZCmM]W	7
10	20	IBO_[uFG	I@X?kMZO	8
10	20	ICO_zpdfg	ICOXJDxfo	8
10	20	IKr@hoNKo	I?dVF?}NO	9
10	20	IlQGhS{Bo	I‘KCYmk\G	9
10	20	IKCeMO—N_	I?YVE_—N_	10
10	20	IlQGhS]JO	I‘GRSlkfG	10
10	20	IXeI?LrFo	I?hRM_}mO	11
10	20	IpKgWNPSw	I_Cg—Xq{G	11
10	20	IJaJcWVIo	I?iYbE{[o	12
10	20	I_CXXô{G	I‘KxGrBow	12
10	20	IcXTHpeeO	IukH@LFT_	13
10	20	IceahlgYG	ItsI@KfT_	13
10	20	IJPKkmgiG	Itp?W[r\?	14
10	20	ILQIstchG	I{cBIkkFG	14
10	21	I@TjcKXfg	I?djQgrdw	15
10	21	I@UKbK]^_	I?Y[qgrXw	15
10	21	IBA[YWj[w	IB?kQ]]\o	16
10	21	ICDrIWjew	ICCjA\ufo	16
10	21	IKr@hoNKw	IQr@‘suBw	17
10	21	IlQGhS{Bw	I‘dh_]RWw	17
10	21	IKCmMO—N_	IQr@hofEw	18
10	21	IlQGhS]JW	I‘YXOlbcw	18
10	21	I[d@bK]Bw	IQV@Okzho	19
10	21	IpKXeDFBw	I‘MohTJow	19
10	21	IJaJcWVIw	ITPHGurRo	20
10	21	I‘KwXfBow	I‘KxGvBow	20
10	21	I‘T\DCN{G	IvO_W]RZG	21
10	21	I‘T\DDdeg	I}CHG]RZG	21
10	22	IYGUUGÑo	I@o]LHr]o	22
10	22	IoGRrh[bw	ID‘BXx[ww	22
10	22	IiGL_}FG	I‘Pcc[z^_	23
10	22	Iq?JpxMlW	IOF?zpm{o	23
10	22	IiGLG}RG	I‘PcSkz^_	24

Co-Cycles Dataset				
N	M	CyclesCannonization	NautyCannonization	Co-Cycles Group
10	22	Iq?JXxYlW	I_IOzpm{o	24
10	22	IJCeMQNNo	IAK]LJb]o	25
10	22	I‘CbjrKbw	ICOnaw{ww	25
10	22	I?S}fEbĜ	IGG}EeZ^_	26
10	22	ICOi _m{W	I_GW ‘m{o	26
10	22	I@PjgqFfw	I?SsbTffw	27
10	22	I@QKg—eĝ	IAGZCmN]w	27
10	22	IEYakthhg	IoLUL_Ŷo	28
10	22	IEYbLTXdg	IoLTeK\Yg	28
10	22	IIeahrJho	IoLTM_Ŷo	29
10	22	IIedUGvUo	IoSteK\Yg	29
10	23	IYr?xSVNW	IRHKshNZo	30
10	23	Io\q‘SVbw	IoCzbpMbw	30
10	23	IhiIO{V\g	Ib_iKsn^_	31
10	23	Io\YPcVhw	IoKag—{ro	31
10	23	IhiGyKZ\W	Ib‘HKsn^_	32
10	23	Io\XaKZhw	IoKQh\{ro	32
10	23	I@TjgqFfw	I?dPjXjfw	33
10	23	I@UKg—eĝ	I?SzCuVZw	33
10	23	IJCmMQNNo	IJeQPMNZo	34
10	23	I‘CjjrKbw	I‘ogzhqbw	34
10	23	IJokOmN\o	IJ_iKuN^_	35
10	23	I‘[honDpw	I‘CuP\{ro	35
10	23	IKNLQyRXg	I‘qkiteMW	36
10	23	IKoytXbdg	I‘qkqdfNO	36
10	23	IKNIlLheg	I‘q\QdfNO	37
10	23	IKNItTddg	I‘q\IteMW	37
10	24	I?vVF? No	I?mrbbNro	38
10	24	IlhYtKNKo	I_hZSlxlg	38
10	24	IgCydMN]w	I@oZeMfŴ	39
10	24	IgC{JdZjw	IEIBXx\xw	39
10	24	I@Sy AFxw	IANcIszxw	40
10	24	I@UeNK]ĝ	IAKÊMfŴ	40
10	24	IJX[suehW	ITiZJLYUW	41
10	24	IJiYrMYhW	ITiZJdLUg	41
10	24	I?zecpvNo	I?]rcflro	42
10	24	I_Yko p\g	ICLP]q̂zG	42
10	24	IK]oxZTy_	I?hRtj\z_	43
10	24	IlC‘hvLrg	I_YXW p{g	43
10	24	I@Q}vVsF_	I?iRjrlv_	44

Co-Cycles Dataset				
N	M	CyclesCannonization	NautyCannonization	Co-Cycles Group
10	24	ILs?w dxg	I_hgw q{W	44
10	25	IaL‘syN\w	IQLas]VŴ	45
10	25	IaMHrhffw	IcLHZdtfw	45
10	25	I]r?Xc No	I@zee_ No	46
10	25	IlhYtKNKw	I‘hZSlxlg	46
10	25	IHTjkqFfw	IKdHZhrfw	47
10	25	IHUKk—eĝ	IILcsmNŴ	47
10	25	I]q‘_\N̂o	I@VVFO n_	48
10	25	Ilgx_F̂Uw	I‘HZS}u}G	48
10	25	IK]unONNG	I@dVNQ—ˆ_	49
10	25	IlKpmVLBw	I‘Kcy szG	49
10	25	IK]unO\Jg	I@hUvH—n_	50
10	25	IlChhvLrg	I‘GZ[s}G	50
10	25	IrXkskN{G	I}aJyWxLW	51
10	25	IrXksktqg	I—qAzg\Jg	51
10	25	IhP\tYtu_	I}iAzg\Jg	52
10	25	IodrZq]{O	I}aJhxMMW	52
10	26	IDoy{Lxxw	ICqbyw—{w	53
10	26	IDqeG—}ô	IDW\JNZvo	53
10	26	IEKÛMfŴ	IC[ZLNrvo	54
10	26	IEMFYw—xw	IChNhx\{w	54
10	27	IpTjkLxfw	I‘iRzX\{w	55
10	27	IpUKjL}ô	IqLck\zô	55
10	27	IiWtkuNŴ	IHtKlNrô	56
10	27	Ii_tzplfw	I‘cnixl{w	56
10	28	IJmwhVVyw	I@h[zx̂—g	57
10	28	ILQHxz— _	I?mrrjnvo	57
10	29	IjmwhVVyw	IKpX—x̂—g	58
10	29	IlQHxz— _	I_mrrjnvo	58

7.3 Appendix C: Machine Specifications

To give a historically situated reader an idea of the difficulty (or triviality) of the computations that I performed, I will provide some basic specs on the two machines that I used:

- 13” Mid 2012 MacBook Air - 4GB RAM, 1.8GHz Intel Core i5, with a quick ping SSD

- 27” Retina 5K iMac - 24GB RAM, 4.0GHz quad-core Intel Core i7, with a very slow ping Network based File System