Brandeis University

Undergraduate Thesis in Computer Science

Graphs: Isomorphism, Cycles, Reconstruction and Randomness

Grady Ward

supervised by
Prf. James Storer

March 27, 2016

Contents

Introduction

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 (and possible re-discovery) 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 niches of it exist which are promising and not yet fully 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 forced me to think about 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, the 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.

Chapter 1

Definitions and Syntax

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

The complete graph over N vertices is K_N. The cycle graph over N vertices is C_N. The star graph is N vertices is S_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:

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

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 problem is known to be computable in quasi-polynomial time CITE, 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 CITE. This group has at least one element (the identity element as the identity isomorphism), but may have as many as $N!$ elements CITE. 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, and comparable in polynomial time. 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:

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

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 four.

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 all further choices. For example, C_n is a perfectly similar graph, as is Peterson's graph CITE.

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 couple of proposed invariants 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.

Cycles are allowed to repeat vertices and edges, and 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 TODO 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 $O(pv^3)$ 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

$$Cycles(A, v_i) = [c_1, c_2, ...c_n], c_p = Cycles(A, v_i, p) = A^p[i, i]$$

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 purposes. 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.

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 NxP 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, primarily for advancements in computational limits, by using delayed evaluation and multi-threaded comparison.

1.6.4 Invertibility

Note that if two graphs agree on the paths invariant, their complements (or inverses) may also agree on the paths 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 guarantee as a general case.

1.6.5 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, generally, this is a quick computation. This is made asymptotically better by the fact that efficient algorithms for matrix multiplication are well studied and optimized for a variety of contexts. In a modern context, matrix multiplication can be done efficiently by GPU arrays, and there are quick ways to do multiplication on sparse matrices.

1.6.6 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 $N = 10$ CITE. Overflows occur when the summed value of the elements in the dot product add to a number greater than 2^32. 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:

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

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). If they later agree on a value for the cycles function, we know that CITE.

1.6.7 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 [CITE] 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:

$$O(Bits(N)) = \log_2(N-1)^{N-1} = (N-1)\log_2(N-1) = O(N \log N)$$

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, Representation

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 set value of the invariant allows only 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.

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 $M_{Reps}(G)$. 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}).

Chapter 2

Cycles as a Graph Invariant

Cycles is a powerful graph invariant. In this chapter we will discuss properties of cycles 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

2.1.1 Vertices, Edges, Degree Sequence

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.

2.1.2 Triangles, Higher Order Polygons

Within the context of a graph, 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.

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 CITE.

2.1.3 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 "cospectral" (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.

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

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

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,

$$\lambda_1 + \lambda_2 + \lambda_3 + \cdots + \lambda_V = \sum_{i=0}^{V} Cycles(1, i)$$

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

$$\sum_{i=0}^{V} \lambda_i^P = \sum_{i=0}^{V} Cycles(P, i)$$

This means that we can create V nonparallel (as not divisible in the ring $R[x]$) equations, each containing λ_1 to λ_V 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 non-distinct. 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 lambda'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 $Paths(p, v)$, we could calculate the eigenvalues of the matrix A using $Paths(p, v)$ restricted to $1 <= p <= V$. This is superb, as it shows that the $Paths$ function encodes at least as much information as the chromatic polynomial, and that any graphs which produce the same $Paths$ function up to (p = V) are cospectral. A quick verification of the base non-isomorphic cospectral graphs (the butterfly and the box) show that they produce *different* values of the paths function, showing that the $Paths$ function carries *more* information than graph spectrum does, even though both are computed in polynomial time. Also helpful: while cospectral tests might rule out isomorphisms instead of providing a mechanism for their potential generation, the Paths function suggests the isomorphism

that it believes to exist with a natural sorting and comparison of the vectors within the $Paths$ object.

2.1.4 Reconstructing a Valid Graph from Paths

If we are given full access to the $Paths$ function, could we create a graph that would produce that function? Note that this question is fundamentally distinct from the one which asks if we could reconstruct the *only* graph that could generate that Graph Function. That is a separate question, equivalent to the question of if GI = P, but the question posed is signifigantly more general, and does not decide or influence the question of computational complexity of GI.

The answer to this question, fortunately, is yes. 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, which by their definition will always hold as true. 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. [NOTE ON BRUTE FORCE]

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 $Paths$ function, is calculable in polynomial time. The fact that we can reconstruct a valid graph from the paths function is about exploring or debunking the invertible nature of the $Paths$ object, and the time it takes to perform an inverse of our 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 an *enormous* 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 paths function), but the reader should try to convince themselves that this approach would terminate, and would yield a valid solution, given a valid $Paths$ object.

We know that the diagonals of A^p yield the $Paths$ function. We also know that A is comprised of $(v*(v-1))/2$ 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 $Paths$ function for a graph, and we will call $Paths(p, v) = k_{p,v}$ to simplify our work. In each equation, we will set the paths 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 $[1, V]$ 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 positive, 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 $Paths$, 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

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

Where z is an integer, $z \geq 1$, and N and D can be any number of positive expressions that are comprised of terms of the addition and multiplication of variables in the set $\{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{(v(v-1))/2}\}$. 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 paths 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 (N) 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:

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

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

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

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

Expressing that N, 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 N has a positive, integer valued weight that corresponds to its coefficient. If

N 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 N 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 N 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

2.2.2 Deck Reconstructability

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 distinguishes between vertices in a way which fully distinguishes them. 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 constant amount of work that has to be done to calculate cycles for any number of vertices in the 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.

One way we can discuss this tradeoff is computationally. An invariant that is 'good' distinguishes between non-similar vertices in a small amount of time. An invariant is 'worse' the more time it takes to compute.

[COMPUTATIONAL RESULTS HERE]

2.4 Discrimination on Tough Graph Classes

2.4.1 Background

Any algorithm or methodology we create needs to be verified in the same terms that precursors have used to verify themselves. One way to do this is to examine the performance of tasks (such as canonical labeling and isomorphism testing) on the same classes of graphs that established renditions of those algorithms have used to push their performance. The most established canonical labeler comes from the NAUTY package CITE. The three sets of graphs that the authors of that package use to push the performance of their algorithm are the 1-Sparse, 2-Dense, and Miyizaki Graphs, each of which present their own set of challenges. In this section, we will briefly outline each of these sets of graphs, then show how a cannonical labeler based in cycles performs on each, comparing against NAUTY benchmarks.

2.4.2 1-Sparse Graphs

2.4.3 2-Dense Graphs

2.4.4 Miyizaki Graphs

2.5 Imperfection, Co-Cycles Graphs

2.5.1 Discovering Co-Cycles Graphs

2.5.2 Constructing Co-Cycles Graphs

2.5.3 As a Proposed Dataset for Invariant Analysis

2.6 Discriminatory Agreement By N and M

2.6.1 Expectations Borne out of Graph Counts

2.6.2 An Unexpected Dip

Chapter 3

Cycles as a Vertex Invariant

A powerful feature of the cycles invariant is that it is not only discriminatory between graphs, it suggests potential isomorphisms between their vertices. 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 Perfect Similar Vertex Sets

A vertex-similar sets 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. 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 to each set within the collection of similar vertex sets. 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.

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

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

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

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

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

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

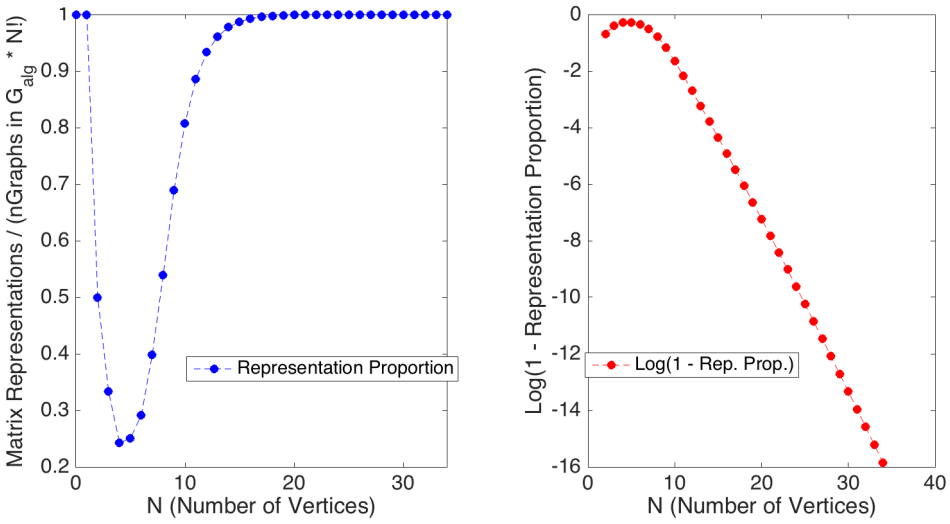
Though we don't have a closed form for this, a quick calculation pretty easy, and gives us a very clear trend. In Figure 3.2, there are two figures shown. The first plot shows the right hand side of the final equation, graphed for different values of N. The second plot explores the difference between the first plot and one, and takes the logarithm to see how small the values get.

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 reason that if a graph only has a single automorphism, then the only automorphism is the trivial automorphism. Thus, the SVSes are a set of N distinct, one element sets,

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.



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 are constructed with respect to a vertex invariant. 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 QSVS.

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 $P^* < N$: 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.

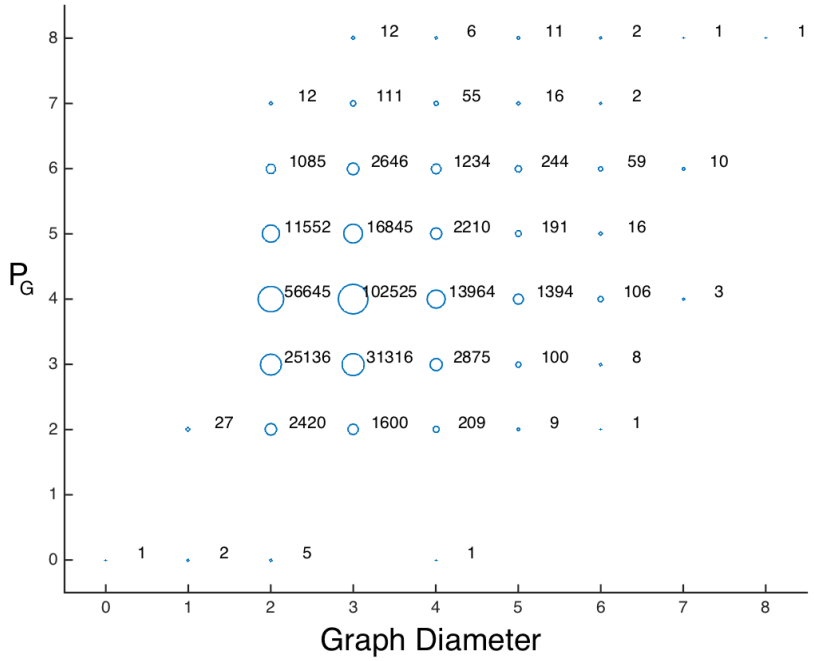
Specifically, within the context of cycles as a vertex invariant, I started out with the assumption that all vertices share a single Similar Vertex Set (i.e. all are similar unless we have proof that they are not). We then distinguish vertices by advancing P by one, we are in effect splitting this one class (or smaller classes) into 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 amount of P). Thus, if we take the maximum of P_G across all G, we will find a value P^* at which point computation beyond this point is redundant. This would enable us to more concretely talk about running time, and its tradeoffs. What we find (through both computational exploration and theoretical justification) is that P^* is contingent upon N, and specifically that:

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

3.4.2 Observational Data: Diameter vs. P_G

I first tried to explain P_G as a function of the individual graph G. 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 coorelation is very strong. It is also interesting (personally) just to see the diameter of a set of graphs laid out like this, just a side note.

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

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

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

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^* make sense?

This proof consists of two parts, the first is why $P_{P_N} =$ the proposedP^* 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

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

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

$$P = 2\lceil\frac{N-2}{2}\rceil = 2\lceil\frac{2K+1-2}{2}\rceil = 2\lceil K - 1/2\rceil = 2K = N - 1$$

Thus, we know that

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

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 with solid results.

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 deterministic way should preserve isomorphism. We use this principle in the pursuit of fully differentiating a set of QSVS into a refined state that more clearly mirrors 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. What a mouth-full!

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, we can predict the number of cycles for the flag vertex, as each one of the cycles that it is a part of must go to and from the flagged vertex, 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 are tangibly new.

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 energy reverberates around a structure. One conceptual tool I like is to think about electricity flowing (bidirectionally) around a circuit, and measuring the self-connectivity or resistance.

Flagging a node changes that dynamic. 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 $N <= 10$, cycles with flagging as a mechanism for QSVS generation correctly generated QSVSes which turned out to be the same as the 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? Maybe. 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 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

Cycles and the Reconstruction Conjecture

4.1 Reconstruction Conjecture

The *Reconstruction Conjecture* is an open conjecture in theoretical computer science which proposes a relationship between subgraph isomorphisms and graph isomorphism. Specifically, the conjecture revolves around the idea of a *deck* of a graph G: a multiset of graphs which are the vertex deleted subgraphs of the graph. Two decks are isomorphic if their graph constituents can be matched up on a one to one basis such that each pair of deck *cards* is isomorphic. The conjecture claims that if two graphs G and H have isomorphic decks, then they are isomorphic.

4.1.1 Manual Verification

In 2004,

Chapter 5

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.

5.1 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.

5.2 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.

5.3 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 6

Random Graph Generators and Automorphisms

6.1 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.

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.2 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.

6.3 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.

6.3.1 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:

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

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

$\overline{T}_{Ideal}$ = 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.

6.3.2 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.

6.3.3 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.

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 graph algo-

rithms to underweight what we consider 'hard' cases, and overweight what we would generally consider 'easy' cases.

6.4 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.

6.4.1 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?

6.4.2 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.

6.4.3 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 statisitcs 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.

6.4.4 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.

6.4.5 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-Paths graphs - Same as Co-cycles graphs, this only applies to graphs of size 10 and larger, counts the number of co-paths 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 $FFC(\dot{F}FV^2)/nGraphs$
- 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

6.4.6 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, by comparing 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:

$$\text{Score for Metric M} = \frac{T_M(I,N)}{T_M(I,G)}$$

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

where higher scores are worse, and a score of 1 corresponds to a generator that is approximately 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.

6.5 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.

6.5.1 Cloning Model

One of the thoughts that I first had was on the way that co-paths 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 7

Reflections

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.