Brandeis University

Undergraduate Thesis in Computer Science

Graphs: Isomorphism, Cycles, Reconstruction and Randomness

Grady Ward

supervised by
Prf. James Storer

March 11, 2016

Contents

Introduction

We can't choose what interests us, but we do get to choose what we pursue.

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.

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. 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 research and documentation techniques. My skill set has been broadened by a project which has deeply challenged me and always kept me on my toes.

I would like to thank my advisor, 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, and it is frequently used without clarification that $M \in [0, \frac{1}{2}(N)(N-1)]$. 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 $(v_1, v_2) \in E(H) \leftrightarrow (v_1, v_2) \notin E(G)$.

1.1.1 Representations, Labeling, Matrices

An important distinction in this report is on the *representation* of graphs. Most graphs can be represented by distinct adjacency matrices, but changes to representation do 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 factor into our algorithms. Throughout this report that when we discuss a set of graphs, or an algorithm over graphs, we will treat graphs as objects which denote structure, and will in every way be blind to their representation. Thus 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. This is not a semantic choice, it has important implications (particularly around ideas of random graph models).

1.2 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.

1.2.1 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).

1.2.2 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 ($\beta = 0$).

1.2.3 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.

1.2.4 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 two graphs. 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 ideas that might suggest idealized invariants.

1.3 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 the other denotations.

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. 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.

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

$$Paths(G, v_i, p) = A^p(i, i)$$

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 paths 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.

Extending this vertex invariant to a graph invariant requires no imagination. We simply calculate the paths 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.

DEPRECATED

The $Paths(p, v)$ function counts the number of closed paths 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:

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

Where v is being used interchangeably here with its represented position within the adjacency matrix. Thus, calculating a specific value of $Paths(p, v)$ 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 nave assumption that matrix multiplication runs in $O(n^3)$ in order to make the computational complexity calculations more accessible. Under that assumption, it is clear that calculating $Paths(p, v)$ will occur in $O(pv^3)$ time. Thus, as long as we request a polynomial number of values from $Paths(p, v)$, any algorithm derived from $Paths$ is in P. Thus we can consider this invariant one in the traditional vein of looking for polynomial invariants with the broad aim of solving and approximating GI in polynomial time, though we have already established that it does not uniquely determine isomorphism.

END DEPRECATED

1.3.1 Expansion to a Paths Object

Since $Paths(p, v)$ is a function that operates over a vertex, we will be examining a slightly modified version of the function which operates over a graph: $Paths(P, G)$. $Paths(P, G)$ produces a graph invariant that is related to its vertex invariant function by a simple translation. $Paths(p, v)$ is calculated for all $v \in G, p \leq P$, resulting in v vectors, each of which has P elements, representing the paths function when calculated at that node for each successive value of p. Since vectors are comparable in linear time, we can sort them in log-quadratic time, and return back a list of these vectors as a graph invariant, one that is computable in polynomial time and 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 $Paths$ 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 naive estimates, primarily for advancements in computational limits, not its theoretical pursuit.

1.3.2 Invertibility

Note that if two graphs agree on the paths invariant, their complements (or inverses) also agree on the paths function. This is easily observed through

the knowledge that two graphs are isomorphic if and only if their complement graphs are isomorphic.

WAIT COPATHS GRAPHS THIS COULD BE FUCKING HUGE TODO !!! PATHS + INVERSE(PATHS) != PATHS(K) IS THIS FUCKING TRUE?!?!

1.3.3 Running Time

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, but for the sake of simplicity and comprehensibility within this paper we will assuem 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.

1.3.4 Dealing with 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. In the worst case, the largest value in the Paths invariant will occur when we have a fully connected graph of size N. The largest value will be TODO as can be quickly shown via TODO.

We can avoid most of the problems associated with this by simply performing regular modulo operations by a large value of 2^K. This preserves thee properties of addition and composition that we are looking for, but does us th favor of maintaining reasonable sized bit arrays. If we break every number into two parts, one divisible by a large power, we can see that any means TODO.

1.3.5 Asymptotic Bit Growth

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, it is not acceptable to ignore the possibility of collisions. Thus, when we discuss the cycles invariant within the context of theory, we need to prove that computation of it can be done in linear space. Otherwise, the polynomial aspect of this computation could be misleading away from a gross inefficency in space that masks the the true tosts and limiting factors of the computation.

Proving that the number of bits that the paths function takes is actually not too difficult. TODO Waiting on invertability

1.4 Reconstructability, Determined, 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 calculate the property if we are given the Cycles invariant for the graph. Similarly, a property is determined by Cycles if a set value of cycles allows only zero or one value for the property in question. Reconstructability and determinability differ only in that reconstruct-ability describes a procedure for the conversion, while determinability only claims a valid mapping, and doesn't suggest a mechansim for its computation.

Finally, we will talk about the number of distinct matrices that describe isomorphic graphs as being the 'number of representations' of the graph, or $Reps(G)$.

TODO: What if we have Paths and Cycles (Paths being not closed?) Or like sorted rows/columns of AI???

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 prove the example of Co-Cycles graphs, and the establishment of small datasets of co-cycles graphs as matrix by which to measure 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 paths 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 adjacenct to a specific node. We can formalize this notion as follows. TODO 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. Similarly, figuring out higher order polygons can be done, but it requires an awareness of the adjacent values of cycles. More 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 a conception of a 'neighbors aware mechanism'. A formal discussion of such mechanism sis discussed in TODO.

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 graphs 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 $Paths$ 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 $Paths$ function to determine the entries of the diagonal, we know that for any adjacency matrix A,

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

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

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

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. 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

3.1 Quantifying how Discrimination Varies with P

3.1.1 Limitations of Cycles' Discriminatory Power

3.1.2 Observational Data

3.1.3 Theoretical Explanation: Path vs Cycle Graphs

3.2 Automorphism 'Quazi-Equivalence Classes'

3.2.1 Background

3.2.2 Vertex Similarity is Transitive

3.2.3 Internal Structure of QEC's

3.3 Improving upon QECs

3.3.1 Appending a Flag, Somewhat Predictable

3.3.2 Theoretical Justification for Flagging

3.3.3 Analytical Support for Flagging

3.4 Limitations to Augmentation

3.4.1 A Second Augmentation Hypothesis

Chapter 4

Cycles and the Reconstruction Conjecture

Chapter 5

Canonical Labeling Using Cycles

5.1 Background

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 check to make sure that alike-labeled vertices preserve all adjacencies and non-adjacencies. Thus, any canonical labeling algorithm is in 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.

5.2 A Consistent Algorithm

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

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 traverse through these permutations and select the matrix which is lexicographically the 'smallest'. A clever optimization allows us to not generate any permutations which are automatically 'larger' than the thus far 'smallest' matrix.

I wrote and optimized the algorithm for Matlab (as that is the slowest of my use cases), but then worked on implementing it in C and Javascript. 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.

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 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 through 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 or NP-Complete graph problems.

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 an alternative mechanisms by which graphs can be generated for more theoretical applications and algorithms.

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 disemminate 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 sufficently 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.

6.2 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 without 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. 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 an incredibly smart canonical labeling algorithm. This algorithm has two abstract components, one which correctly places vertexes into Similar Vertex Sets, 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 test for consistency against some canonical property. One common property for canonization is selecting the adjacency matrix which is the lexicographically smallest representation of the graph. If we assume that the second half of the algorithm dominates the running time of the overall algorithm (which is a reasonable assumption), we can express the running time of the overall algorithm in terms of $O(|Aut(G)|)$, as this is the number of matrices we will need to evaluate on a canonical property.

This is important because the number of matrix representations for a given graph G is $\frac{N!}{|Aut(G)|}$. Thus, if we are considering the average running time of this canonization algorithm over the set of all graph instances (i.e. using the standard models for random graph generation), we come to the key conclusion:

$$T^{Gilbert}_{Average} = \frac{\sum_{g}^{Graphs} O(T(g)) * N_{Representations}(g)}{|AllMatrixRepresentations|}$$

$$T^{Gilbert}_{Average} = \frac{\sum_{g}^{Graphs} |Aut(g)| * \frac{N!}{|Aut(g)|}}{|AllMatrixRepresentations|}$$

$$T^{Gilbert}_{Average} = \frac{\sum_{g}^{Graphs} N!}{2^{.5(N^2-N)}}$$

$$T^{Gilbert}_{Average} = \frac{|Graphs(N)| * N!}{2^{.5(N^2-N)}} \sim 1$$

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

$$T^{Ideal}_{Average} = \frac{\sum_{g}^{Graphs} O(T(g))}{|AllPossibleGraphs|}$$

$$T^{Ideal}_{Average} = \frac{\sum_{g}^{Graphs} |Aut(g)|}{|Graphs(N)|}$$

$T^{Ideal}_{Average}$ = Average Number of Automorphisms over Graphs >> 1

Some sample numbers to give you the scale of these disparities is given below CITE.

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. 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

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

6.4 Measuring Flaws of Random Graph Models

Our mandate suddenly becomes: '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. 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, we will be generating a random graph given two input parameters, N and p, where p describes the probability of getting any given edge (not independent of other edges, however). 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 probability of a success 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 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 a 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?

Answering these questions required some really deliberate thought. I knew lots about what I didn't want to do: I didn't want to write generators and then use data to support them. I didn't want to focus in on a single way 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 twenty seven statistics of sets of randomly generated graphs. Each of the statistics I decided on was because I subjectively valued it as a property that I would hope a random graph generator would have. Each of

the set metrics is described below, and they vary in computational complexity and broad descriptiveness. I set up a (pardon if I say it) BEAUTIFUL (and quasi-distributed) system for calculating these statistics over arbitrary datasets. I created an idealized random generator and a Gilbert random generator, and tested 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.

6.4.1 Graph Set Metrics

- ADIAM - Statistic - Average Graph Diameter (i.e. Take the diameter of each graph, and average over all graphs in the set)
- ADSM - Statistic - Average Degree Sequence Mean (i.e. take the mean of degree sequence of each graph, and average that across all graphs)
- ADSMD - Statistic - Average Degree Sequence Median (i.e. take the median of the degree sequence of each graph, then average that across all graphs)
- ADSMN - Statistic - Average Degree Sequence Minimum (i.e. take the degree of the least connected node in each graph, then average that across all graphs)
- ADSMX - Statistic - Average Degree Sequence Maximum (i.e. take the degree of the most connected node in each graph, then average that across all graphs)
- ADSV - Statistic - Average Degree Sequence Variance (i.e. take the variance of the degree sequence of the graph, then average across all graphs)
- ANA - Statistic - Average Number of Automorphisms (i.e. find the number of automorphisms for each graph, then average that across the set)
- ANCC - Statistic - Average Number of Connected Components (i.e. take the number of connected components of each graph in the set, then average across all of them)
- DIAMV - Statistic - Variance in the Diameter of Graphs (i.e. calculate the diameter of each graph, and calculate the variance of the set as a whole)

- PCONN - Statistic - Probability of Connectivity (i.e. take the number of fully connected graphs, and divide by the total number of graphs in the set)
- NCP - 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 - Statistic - Number of Edges (i.e. 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 - Statistic - Number of Regular Graphs (i.e. the number of regular graphs in the graphset)

6.5 Flaws in Proofs of Average Case Random Graphs

6.5.1 Selection of Certain Classes of Graphs

6.5.2 Examples Uses in Proofs

6.5.3 Average Case Runtime Under Erdos-Reyni and Worst Case

6.6 Alternative Ideas for Random Graph Modeling and Creation

6.6.1 Distributional Goals

6.6.2 Cloning Model

6.7 Inherent Limitations on Models by Computational Theory

6.7.1 Uncertainty in Set Size

6.7.2 Computational Verification Limits

6.8 On the Number of Graphs of a given Size

The number of non-isomorphic graphs of a given size is an open question. The first twenty two terms of this sequence have been calculated, and are available in the Online Encyclopedia for Integer Sequences[4].

6.8.1 Proposed Closed Forms

Chapter 7

Reflections

7.1 Broad Project, Unclear Aims

7.2 Modes of Discovery

7.3 Freedom to Pursue Interest

7.4 Acknowledgments

Bibliography

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891?921, 1905.

[3] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`

[4] Online Encyclopedia for Integer Sequences, A000088
`https://oeis.org/A000088`