Brandeis University

Undergraduate Thesis in Computer Science

Graph Isomorphism, Reconstruction and the Cycles Invariant

Grady Ward

supervised by
Prf. James Storer

March 10, 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 rediscovery) 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 finding niches of it that are promising and not yet fully explored is difficult. 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 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.

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 descision problem (GI) can be computed in polynomial time is an open question in theoretical computer science. The problem is known to be solveable in quasi-polynomial time, though no convincing arguments have placed it in NP-complete nor in P.

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

1.2.1 Graph Invariants

A *graph invariant* is an ordered property of a graph which remains the same irrespective of the representation or labeling of the graph. More specifically, an algorithm or property is a graph invariant only if it produces output which is stable across all instances of the same graph. A graph invariant $I(G)$ can allow us to conclude that two graphs (G_1, G_2) are *not* the same graph if $I(G_1) \neq I(G_2)$. However, it is distinctly limited, in that the converse does not 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 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 I discriminates at a level α for N vertices and M edges if selecting two graphs G and H at random from the set of all graphs with N vertices and M edges:

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

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 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 pair of 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 can calculate over a specific vertex within a graph which attempts to identify potentially similar vertex pairs. Similar vertices within a graph will 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.

1.3.1 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?!?!

1.3.2 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.

1.3.3 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.

1.3.4 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 vertices or edges.

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

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

2.4 Discrimination on Tough Graph Classes

2.4.1 Background

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

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 kicked into motion with two independently generated papers in 1959, defining two models of generating and analyzing random graphs. A emphrandom graph model is any system of generating graphs through chance, with various constraints that describe the desired properties of the output. These two papers began a movement in graph theory that allowed theoreticians and algorithm designers new tools to examine notions of efficiency, 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. 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.

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 discovered 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. Though many 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.

6.1.3 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 model random graphs.

6.2 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]

Chapter 7

Reflections

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`