COMPUTING STRATEGIES FOR GRAPHICAL NIM

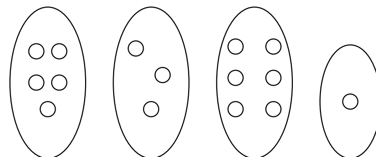
SARAH LEGGETT, BRYCE RICHARDS, NATHAN SITARAMAN,
STEPHANIE THOMAS

Abstract. In this paper, we use the Sprague-Grundy theorem to analyze modified versions of Nim played on various graphs. We also describe the periodic behavior of the Sprague-Grundy numbers for games played on paths, triangle paths, and caterpillars. A brief hueristic analysis of the distribution of Sprauge-Grundy numbers for Nim played on trees and graphs of order n are discussed. This research was completed during the Clemson University Math REU which was funded by the NSF[1].

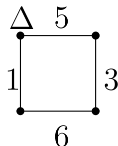
1. Introduction

A game in which both players have the same set of possible movements in game play is called impartial. On the other hand, a partizan game is whose players are limited to movements that their opponents cannot take. The game of checkers is partizan because the players are restricted to moving a single color piece. Nim is a simple two player game in which players take turns removing rocks from disjoint piles until there are no rocks remaining. The player to pick up the last rock (or group of rocks) is the winner. The game of Nim is impartial.

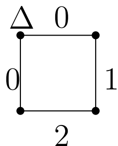
Nim played on graphs has previously been analyzed by Fukuyama [5], who adapted a graphical representation of the traditional Nim game played with piles of rocks. The piles of rocks in Fukuyama's *Nim on Graphs* are represented with vertices and the edges of the graphs are labelled with the number of rocks in each pile. An example of Nim on piles is shown below.



The corresponding graph would be:



A player's move begins at the vertex labelled Δ. The players alternate travelling vertex to vertex, decreasing the number on the label of each edge they pass along the way. The significance of decreasing the label on the edge is equivalent to removing rocks in a pile. A player loses when he is at a vertex that is connected to all edges labeled with zeroes (as demonstrated below).



We will discuss the new version created by Michael Albert (and communicated to us by Neil Calkin) where players take turns removing edges from vertices in graphs instead of rocks from piles.

The object of Graph Nim is to be the person to remove the last set of edges from a given graph. Instead of removing rocks from disjoint piles, a player can remove edges that are incident to a given vertex. The number of edges incident to a given vertex is said to be the degree of the vertex; hence, the maximum number of edges that can be removed in a player's turn is equal to the degree of a specific vertex. For example, if a vertex has degree 4, a player can remove 1, 2, 3, or all 4 edges from that vertex.

When analyzing Graph Nim, it is not long before we notice that certain graphs are winning (or losing) graphs for Player 1. An impartial game consists of N-positions and P-positions. An N-position is defined to be a position where the Next player to move will win the game. Similarly, a P-position is a winning position for the Previous player. When a game has reached a point where no player is able to move, the game is said to be in terminal position. In the case with Nim, the terminal position is the position in which there are no rocks left to pick up. A terminal position is a P-position.

One method for determining who wins an impartial game is analyzing N-positions and P-positions, but another method involves utilizing the Sprague-Grundy Theorem. A winning strategy for traditional Nim has

already been discovered [1] and we hope to expand the concept with the new versions of the game we introduced.

2. Graph Theory Background

Here we provide a few definitions of some of the basic concepts of graph theory. In the sections that discuss Graph-Nim played on specific types of graphs, we will provide some more specific definitions.

Definition. *A graph G is an order pair (V, E) comprising of a set V of vertices and a set E of edges between the vertices, with $E \subseteq V \times V$. A simple graph is a graph in which an edge may only connect two distinct vertices and in which two vertices may only be connected by a single edge. A multi-graph is a graph which allows multiple edges between (not necessarily distinct) vertices. Unless otherwise stated, we will be working with simple graphs.*

Definition. *We call the number of vertices of a graph G, the order of G. We typically use the letter n to denote the order of a graph.*

Definition. *We call the number of edges of a graph G, the size of G. We typically use the letter m to denote the size of a graph. We also write $e(G)$ to denote the size of G.*

Definition. *A labeled graph is a graph in which each vertex is assigned with a label, typically the integers 1 through n. An unlabeled graph is a graph whose vertices have no identifications except through their interconnectivity. Unless otherwise stated, we will be working with unlabeled graphs.*

Definition. *A graph isomorphism between two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ is a function $\phi : V_G \to V_H$ s.t. $(\phi(v_i), \phi(v_j)) \in E_H \iff (v_i, v_j) \in E_G$. More intuitively, two labeled graphs G and H are isomorphic if they have the same "graph structure."*

Definition. *An isomorphism class of of a labeled graph G is the collection of all labeled graphs which are isomorphic to G.*

3. Sprague-Grundy

3.1. Sprague-Grundy Function.

Definition. *A follower is a position a player can obtain in one move in a game.*

Definition. *Given a finite set of integers S, x is the minimum excluded value if it is the smallest non-negative integer such that $x \notin S$.*

For example, for the set $K = \{0, 1, 3, 5, 6, 7\}$ the mex is 2.

Let $F(x)$ denote the followers of a given position x. The Sprague-Grundy function, $g(x)$, is defined as

$$g(x) = \text{mex } \{g(y) : y \in F(x)\} \text{ [4].}$$

For the traditional game of Nim, the winning strategy is to finish every move leaving the game's Sprague-Grundy value at zero because of the following theorem.

Theorem. *A position in Nim is a P-position if and only if the nim-sum of its components is zero.*

It follows from the definition of the Sprague-Grundy function that once a player is given a position in a game with a Sprague-Grundy number equal to zero, then any move that player makes will change the value of the game to some non-zero Sprague-Grundy number. It is also known that a player can force a Sprague-Grundy value of zero onto the next player only if the player was not handed a Sprague-Grundy value of zero at the beginning of their turn.

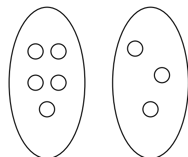
Sprague-Grundy function values are helpful when analyzing Nim played on graphs. The Sprague-Grundy theorem explains the reason why we take interest in computing Sprague-Grundy numbers.

3.2. **Sprague-Grundy Theorem.**

Theorem. *The Sprague-Grundy value of a game consisting of many disjoint games is the nim-sum of the Sprague-Grundy values of those components.*

Note that Sprague-Grundy will now be denoted by S-G. In the case of traditional Nim, the S-G number of the entire game is the addition of each pile's S-G number. In other words, we can consider each pile as a distinct game with its own S-G number. To calculate a nim-sum with traditional Nim, we first must realize that the S-G number for a pile of rocks is simply the number of rocks in the pile (this is trivial). We then take the S-G numbers from all piles and convert them to binary. The nim-sum is then found by the addition of all converted S-G numbers mod 2.

For example, in a game with two piles below,



we take the number of rocks in each pile, 5 and 3, and convert to binary. We then have the nim-sum below.

$$\begin{array}{rccc l} & 1 & 0 & 1 & = 5 \\ + & & 1 & 1 & = 3 \\ \hline & 1 & 1 & 0 & = 6 \end{array}$$

Thus, since the S-G number for this game is greater than zero, player one has a winning strategy.

With the nim sum calculated for a given game, it is possible for a player to determine whether a game win is achievable. We know that a game with a nim-sum of zero is in P-position. Otherwise, if the nim-sum is non-zero, the game is in N-position. So in order for a player to win a game of Nim from an N-position, he should remove enough rocks to force the nim-sum to zero.

The strategy for forcing a nim-sum to zero is as follows:

(1) Find the left most 1 in the calculated sum. Let's call the column you find this 1 in k.
(2) From that 1, trace up the column until you find a 1. Note that the row you found this 1 in will be the only row to be modified, let's call it r.
(3) Change the 1 is row r column k to a 0.
(4) We then manipulate the 1's and 0's in the same row we changed in the last step so that each column's nim-sum is zero.
(5) We continue this process until all 1's in the original nim-sum are 0's.
(6) Subtract the value of the altered form of row r from the orginial value of row r to determine the winning move. The difference in the rows will be the number of rocks you need to remove from the pile represented by row r.

Let's denomstrate how this strategy works. Take into consideration the game played on two piles mentioned earlier. The nim-sum we calculated was:

$$\begin{array}{rccc} & 1 & 0 & 1 \\ + & & 1 & 1 \\ \hline & 1 & 1 & 0 \end{array}$$

Step 1:

$$\begin{array}{rccc} & 1 & 0 & 1 \\ + & & 1 & 1 \\ \hline & \mathbf{1} & 1 & 0 \end{array}$$

Step 2:

$$\begin{array}{rccc} & \mathbf{1} & 0 & 1 \\ + & & 1 & 1 \\ \hline & 1 & 1 & 0 \end{array}$$

Step 3:

$$\begin{array}{rccc} & 0 & 0 & 1 \\ + & & 1 & 1 \\ \hline & 0 & 1 & 0 \end{array}$$

Step 4:

$$\begin{array}{rccc} & 0 & 1 & 1 \\ + & & 1 & 1 \\ \hline & 0 & 0 & 0 \end{array}$$

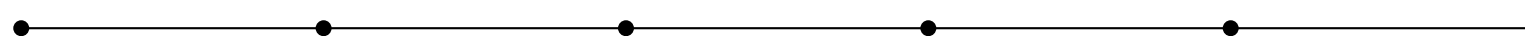
Since $5 - 3 = 2$, we remove 2 rocks from the pile represented in the first row. Since the next player will have no choice but to change the nim-sum to a non-zero value, we know that we are able to force the nim-sum to zero again. We continue this pattern until the game is won.

4. Graph Nim on Paths

4.1. **Simple Paths.**

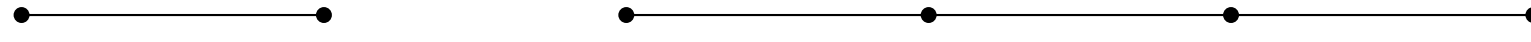
Definition. *A path with n edges, denoted P_n, is a tree with two vertices of degree one and all other vertices of degree two.*

An example of P_5 is shown below.



When playing Path Nim, we consider disjoint paths instead of disjoint piles of rocks. A move is made by by removing either one edge, or two edges connected to the same vertex. The illustrations below should clarify the movements allowable on a path.

From P_5, we take away one edge to create two disjoint paths P_1 and P_3.



Also from P_5, we can take away two edges creating disjoint paths P_1 and P_2.

We can also take away one edge, therefore creating two paths of equal length. Creating two paths of equal length turns out to be the winning strategy for Player 1.

By creating two paths of equal length, we force each path to have the same Sprauge-Grundy number. It is easy to see that when we nim-sum two equal numbers, we get zero. Therefore, if we are faced with P_{2c}, where $c \in \mathbb{Z}^+$, we know we need to remove the two inner edges incident with the center vertex. Similarly, if we are faced with P_{2c+1}, we only need to remove the center edge. Therefore, since we know that Player 1 can always force the nim-sum to zero at the end of their turn, paths can always be won by Player 1. Since graph circuits are simply paths where the begining vertex is also the terminating vertex, the first movement would result in a path; hence, Player 2 wins Nim played on graph circuits.

With paths, the Sprague-Grundy numbers are as follows:

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	3	1	4	3	2	1	4	2	6
12	4	1	2	7	1	4	3	2	1	4	6	7
24	4	1	2	8	5	4	7	2	1	8	6	7
36	4	1	2	3	1	4	7	2	1	8	2	7
48	4	1	2	8	1	4	7	2	1	4	2	7
60	4	1	2	8	1	4	7	2	1	8	6	7
72	4	1	2	8	1	4	7	2	1	8	2	7

The column labels $0-11$ represent the least residues of the congruence $l(\text{mod}12)$. The row labels represent the length of path in intervals of 12.

We notice that, due to the periodic behavior that occurs, a path of length greater than 72 has an easily computable S-G number. Say that we have a path of length l, where $l \geq 72$, we can use modular arithmetic to calculate its S-G number. For example, for $l = 87$, we have

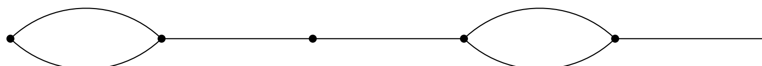
$$87 \equiv 3 \mod 12 \Rightarrow g(87) = 8.$$

The difficulty with computing S-G numbers for paths of length less than 72 is that there are multiple exceptions before the S-G numbers become settle down and become periodic. Luckily, we were able to verify our S-G numbers because our game on path is similar to the game of Kayles [4]. Kayles is a game in which expert bowlers are capable of removing one or two adjacent pins in a line of pins. It does not require much convincing to see that Kayles requires the same S-G function as Path Nim and therefore their S-G values are the same.

4.2. **Multi-edge Paths.**

Definition. *A multi-edge path is a path in which there can be multiple edges between two adjacent vertices. More specifically, the multi-edge that we refer to, the maximum number of edges between those vertices is two.*

An illustration of the definition is below.



Specifically, we examined multi-edge paths where the greatest degree of a given vertex is three. Understandably, calculating S-G numbers became very complex and as we added edges the computation times grew at a tremendous rate. For the sake of our sanity and progress, this part of our research was put on hold indefinitely. However, from the data that we collected, it appears that the S-G numbers for simple cases were periodic.

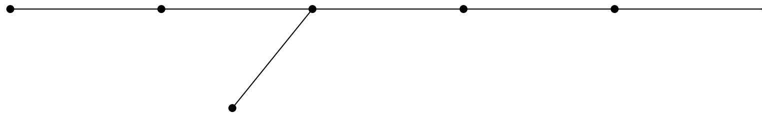
5. Caterpillar Nim

Definition. *A caterpillar, C_n, is defined as a path consisting of n edges with one or more edges appended to k vertices of the path, $k \geq 1$. Any vertex not in the path has degree one and distance one from the main path.*

Definition. *A caterpillar, $C_{n,k}$ is defined as a caterpillar of length n, where n is the number of edges, and consisting of one extra edge (or a leg) on index k, where the leftmost vertex is index zero.*

Note that $C_{n,0}$ and $C_{n,n}$ are equivalent to P_{n+1}.

The gameplay of nim on caterpillars is similar to that on paths. Since there are legs attached to the main path, there are more possible moves available to a player. For example, consider $C_{5,2}$ shown below:



All moves available to a player on a path of length five are also available on this caterpillar (though with different results). In addition, the following four moves are also available for each extra edge:

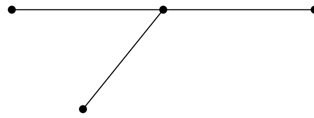
Removing all three edges connected at index 2, we get P_2 and P_1:

Removing the left edge and the appended edge connected to index 2, we get P_1 and P_3:

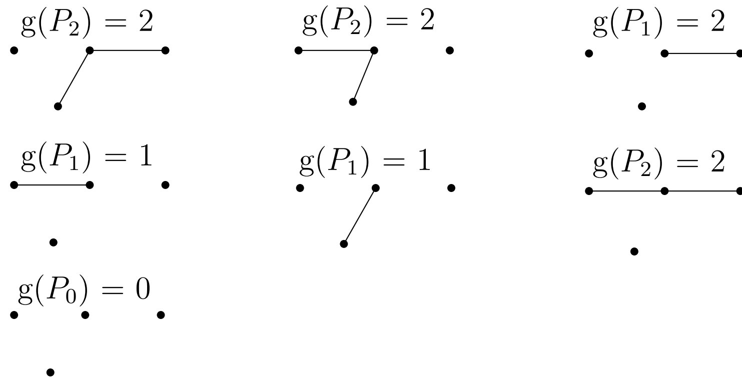
Removing the right edge and the appended edge connected to index 2, we get two copies of P_2:

Removing only the appended edge from index 2, we get P_5:

We want to compute the S-G number for a caterpillar with one appended edge. Consider the simplest caterpillar $C_{2,1}$:



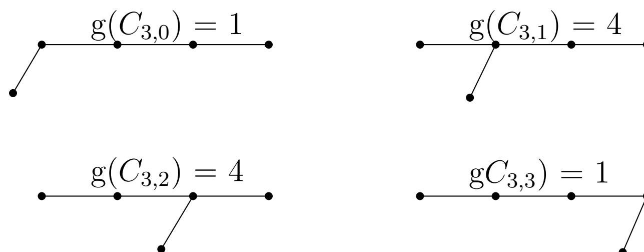
We can obtain the following graphs in one move:



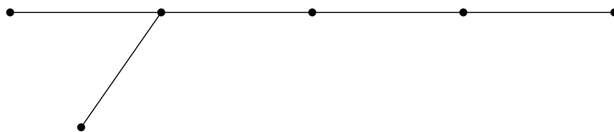
We see that the set of S-G values is $\{0, 1, 2\}$ because the resulting moves consists only of the empty graph and paths of lengths one and

two. Taking the mex of this set, we conclude that the S-G number of $C_{2,1}$ is 3.

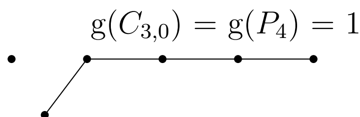
We can continue to analyze caterpillars in this way, obtaining the following S-G numbers for caterpillars of length three:



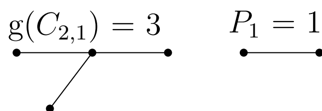
To further understand this process, we can look at a few followers of $C_{4,1}$:



By removing the left most edge from this caterpillar, we obtain the following graph and corresponding S-G number:



Another interesting follower results when we remove the 3rd edge from the caterpillar:



To get the S-G number of this, we must nim-sum 3 and 1. From this nim-sum we obtain 2, which is the S-G number of this follower.

It will be left to the reader to show that the set of the followers' S-G numbers for $C_{4,1}$ is $\{0, 1, 2, 3, 4\}$. With this set, the reader can see that the minimal excluded value is 5, thus the S-G number of $C_{4,1}$ is 5.

We can see that a natural algorithm forms for computing the S-G numbers of caterpillars.

We first developed an algorithm specifically for caterpillars with only one extra edge.

Given a caterpillar of length n, we begin with the extra edge appended to index zero. We first compile a list of the S-G numbers of all possible graphs obtainable in one move, as in the example above. From this list, we find the minimal excluded value (mex), which is the S-G number of $C_{n,0}$. We then move the extra edge to the next index and compute the S-G number of $C_{n,1}$ using the same algorithm. This process is repeated to find the S-G numbers for all caterpillars $C_{n,k}$ up to $C_{n,n}$. See Appendix B, 14.1 for more details.

This algorithm allowed the S-G numbers to be computed for much longer caterpillars. S-G numbers were computed for these caterpillars up to length 4000.

As with the case of paths, if we fix the index of the extra edge, as we increase the length of the main path the S-G numbers of caterpillars become perodic. Thus far, the S-G numbers of caterpillars with one leg eventually have a period of 12 or 60.

The following charts show the periods for two caterpillars that are mod 12. Each table is for a caterpillar $C_{n,k}$ where k remains constant as n changes. For each given position, n is the sum of the corresponding numbers in the first row and first column. The *'s denote a length n, such that the index k is out of range.

Table for $C_{n,1}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	$*$	2	3	4	5	6	2	1	0	8	6	0
12	1	2	3	8	5	12	7	1	0	8	9	14
24	1	2	3	11	4	7	12	14	0	16	2	4
36	12	2	3	10	4	7	15	1	16	9	18	16
48	12	2	3	10	16	7	12	1	16	18	11	16
60	12	2	22	11	16	7	12	1	20	24	16	26
72	12	13	22	11	16	24	15	14	16	22	19	16
84	12	13	19	11	16	24	15	14	16	25	11	16
96	12	13	22	11	16	7	15	1	20	25	19	11
108	12	13	22	11	32	19	22	14	20	22	19	11
120	12	13	22	11	25	19	22	14	16	22	19	11
132	21	13	22	11	25	19	22	14	21	25	19	11
144	21	13	22	11	25	19	22	14	20	22	19	11
156	21	13	22	11	25	19	22	14	21	22	19	11

Table for $C_{n,15}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
12	$*$	$*$	$*$	1	5	7	6	8	15	13	8	4
24	1	8	7	13	1	7	2	1	4	14	7	4
36	1	11	8	13	4	21	2	8	18	19	7	4
48	1	8	21	1	4	7	6	8	1	2	7	4
60	5	8	2	1	4	7	8	16	1	14	7	4
72	1	16	2	13	4	7	22	8	1	2	7	4
84	1	24	2	1	4	7	8	19	1	2	7	4
96	1	24	2	1	4	7	8	24	1	2	7	4
108	1	16	2	1	4	7	2	8	1	2	7	4

While both of these periods start at lengths less than 200, it is important to note that this is not always the case. For example, when the edge is appended at index 8 the S-G numbers of the caterpillars also have a period of 12, but this period does not begin until length 444.

Caterpillars with periods of 60 include:

Index 7 with period: 5, 12, 18, 27, 17, 24, 6, 15, 10, 37, 41, 29, 5, 12, 18, 27, 17, 29, 6, 15, 10, 27, 41, 30, 5, 12, 18, 27, 17, 29, 6, 15, 10, 37, 41, 30, 5, 12, 18, 27, 17, 29, 6, 15, 10, 37, 41, 29, 5, 12, 18, 27, 17, 24, 6, 15, 10, 37, 41, 29

Index 12 with period: 22, 14, 28, 1, 4, 11, 25, 12, 28, 21, 7, 8, 22, 14, 28, 1, 4, 11, 25, 12, 28, 21, 7, 8, 22, 14, 25, 1, 4, 11, 25, 12, 28, 21, 7, 8, 22, 14, 28, 1, 4, 11, 25, 12, 28, 21, 7, 8, 22, 14, 28, 1, 4, 11, 25, 12, 31, 21, 7, 8

See Appendix C for more periods of $C_{n,k}$.

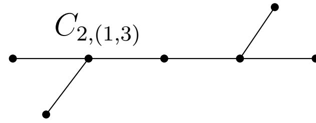
Another interesting period of 12 appears in caterpillars $C_{2k,k}$. By increasing k, we see the following period:

Table for $C_{2k,k}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	3	5	2	3	3	4	3	4	2	4	3
12	4	2	4	2	5	3	4	3	4	2	4	4
24	5	2	4	2	4	7	4	7	4	2	18	3
36	5	2	10	16	4	3	5	7	5	13	13	3
48	5	2	4	7	13	7	5	13	4	2	11	3
60	5	2	11	7	13	3	16	19	4	8	25	3
72	4	2	11	7	18	11	5	7	19	13	14	7
84	29	13	10	7	8	11	5	9	18	13	11	7
96	16	2	11	8	8	11	5	18	19	13	5	7
108	19	2	11	7	19	7	11	13	19	13	5	7
120	16	2	11	7	19	7	5	13	29	13	5	7
132	31	2	11	7	8	7	5	13	19	13	5	7
144	28	2	11	7	21	7	5	13	22	13	5	7
156	21	2	11	7	21	7	5	13	35	13	5	7
168	22	2	11	7	21	7	5	13	29	13	5	7
180	28	2	11	7	21	7	5	13	22	13	5	7
192	28	2	11	7	21	7	5	13	22	13	5	7
204	22	2	11	7	21	7	5	13	29	13	5	7
216	22	2	11	7	21	7	5	13	22	13	5	7

This period is particularly interesting due to the fact that the extra leg is not remaing at the same index for each caterpillar, rather it changes as the length of the caterpillar changes.

This code for caterpillars with one extra leg was expanded to caterpillars with two extra legs. The following is an example of a two-legged caterpillar, denoted $C_{n,(x,y)}$ where n is the length of the caterpillar and x and y are the indices where the extra edges are appended:

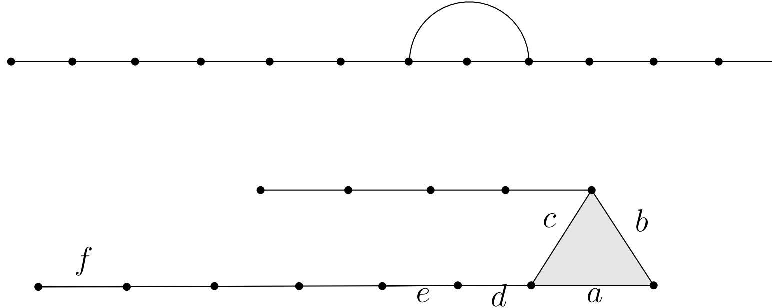


The algorithm for finding these S-G numbers can be generalized to any caterpillar of length n with extra edges appended to indices k_0, k_2, ..., k_m, $0 \leq$ m $\leq$ n.

6. Tri-Paths

Definition. *A triangle-path (which we will refer to as "tri-path") can be defined as a path with two verticies exactly one vertex apart connected to one another by an edge. Alternatively, we can define a tri-path as a path in which two adjacent vertices on a path are connected to a single point off of the path.*

For example on P_{11} with verticies labeled 0-12 a tri-path could be made by connecting vertices 6 and 8 with an additional line.



The above graphs both represent the same tri-path. The name "tri-path" stems from the second drawing, where all edges are the same length.

On a tri-path, a player can use one move to create

1. A single path, by removing edge 'c'
2. Two paths, by removing 'b' and 'c'
3. A path and a one-legged caterpillar, by removing 'c' and 'd'
4. A path and a tri-path, by removing 'e'
5. A one-legged caterpillar by removing 'a'
6. a tri-path by removing 'f'

Other moves can be made resulting in different one-legged caterpillars, paths, and tri-paths, but no other types of follower graphs may be

produced. A tri-path will be denoted as $T_{(m,n)}$. m is the total length of the upper and lower paths when the tri-path is drawn as in the second graph above (this includes all edges except those that form the triangle), and n is the length of the upper path. We fix the upper path to be shorter than the lower path, to avoid isomorphism. For example, a simple triangle would be $T_{(0,0)}$,the graph above would be $T_{(10,4)}$, and $T_{(10,6)}$ would represent the same tri-path, so it would not be considered.

If a move is made on a tri-path and another tri-path is formed, m must decrease, because for the triangle to remain we must remove edges from either the upper or the lower path. Because of this, our program (with data for paths and one-legged caterpillars already present) begins at $m = 0$ and calculates S-G values for any m based on data for tri-paths with smaller m.

A limitation encountered in calculating these values is the amount of data that can be stored and processed. In the calculation of S-G values for a set of tri-paths length m, every tri-path, one-legged caterpillar, and path of length m and lower is accessed at least once. Because of memory limitations, a maximum of 12 megabytes was accessable resulting in a limit of roughly length 1000. Fortunately, this is long enough to reach some interesting conclusions

For the data we have analyzed, tri-paths of a fixed n tend to have patterns of period 12, 36, or 60, keeping consistent to the periods of $12k$ we saw in one-legged caterpillars. A pattern was also noted in "A graphs", graphs where $n = m/2$ (i.e. graphs that look like a capital A).

Unlike one-legged caterpillars, some of these patterns include losing graphs (a S-G value of 0). This ensures an infinite amount of losing graphs, which represent 3.285% of the first 250,500 graphs (all graphs from $m = 0$ to $m = 999$). Larger S-G values are also found in tri-paths compared to one-legged caterpillars, the largest being 80. Because the S-G value 80 is first detected at $m = 397$ and no larger value is found through 1000, it is possible that 80 is the upper bound on S-G values for tri-paths.

7. Distribution of S-G numbers for one-legged caterpillars

Regarding one-legged caterpillars, not all S-G numbers are distributed equally. In fact, while the overall trend is that higher numbers are less common, the chart in Appendix D, 16.1 seems to indicate a very unpredictable distribution.

However, there is a remarkably easy way to classify S-G numbers. We can place all S-G numbers into two categories.

Definition. *If a number has an odd amount of 1's in its binary expansion, we call the number odious. Conversely, if a number has an even amount of 1's in its expansion, we call the number evil.*

While odious numbers are no more common than evil numbers, the odious S-G numbers represent slightly more than 93 percent of all one-legged caterpillars (see Appendix D, 16.2). We hypothesize that the dominance of odious numbers is due to two key phenomena:

Firstly, in early graphs, moves that split the graph into two paths make up a significant proportion of all moves (up to 8). Because paths (with only 10 exceptions, because the mod-12 pattern contains only odious numbers) are odious, and two odious numbers always nim-sum to an evil number (this can be proven easily), more evil follower graphs are produced in this way than odious. Assume for the time being that dividing the one-legged caterpillar into a shorter one-legged caterpillar and a path produces no more odious follower graphs than evil follower graphs (this can be proven true given that evil one-legged caterpillars are no more common than odious ones). If the set from which we select the mex thus has a majority of evil numbers in (most) one-legged caterpillars, the mex is likely to be odious. Because of this, even if odious numbers have no immediate majority, one is sure to develop. Note that because the largest nim-sum of two paths is 15, this argument is only valid for lower S-G numbers.

Secondly, the vast majority of moves, especially on large one-legged caterpillar, result in two separate graphs, where one is a path and the other is another one-legged caterpillar. Note that when the S-G values of these graphs are added, two odious numbers nim-sum to an evil number, while odious and evil nim-sums to odious. If we note an early majority in odious one-legged caterpillars as the former argument would suggest, then we know that we should have more cases of odious (path) plus odious (one-legged caterpillar) than odious (path) plus evil (one-legged caterpillar) in our follower graphs. Because odious+odious=evil, while odious+evil=odious this further increases the majority of evil S-G numbers in our follower set, and thus further increases the proportion of odious one-legged caterpillars. Note that this argument is only valid given an initial majority of odious one-legged caterpillars.

An easy way to visualize argument 2 is to take an odious number we know to be common, 14 for instance. We can nim-sum 14 with the odious path S-G numbers to reach 6, 9, 10, 12, and 15. Because 14 is

common in one-legged caterpillars, the odious path numbers are very common, a graph consisting of one of each is a common (evil) follower graph. Because these follower graphs have evil S-G numbers, we expect these evil S-G numbers to be uncommon in one-legged caterpillars (if they appear in the follower set often, they will rarely be the mex). The numbers that are not steamrolled by this argument are the odious numbers, which logically "pick up the slack" for their odious bretheren. Their ubiquity is reinforced by the fact that follower graphs with their S-G number are almost always evil one-legged caterpillar+odious path, and we know that evil one-legged caterpillars are rare.

The 5 odious path S-G numbers can nim-sum with exactly 5 other numbers to reach any given value n. We hypothesize that the rarity or ubiquity of an S-G number is inversely proportional to the rarity or ubiquity of these 5 numbers. Because for odious n, these 5 numbers are always evil and vice versa, we expect that a majority in a single odious number (or rarity in an evil number) will translate to a rarity in those 5 evil number (or ubiquity in those 5 odious numbers). In many cases, however, these 5 evil numbers nim-sum (with the 5 paths) to fewer odious numbers than the $5 \times 5 = 25$ one might expect. For example, nim-summing the evil numbers of 4 digits and fewer with odious paths can only produce the odious numbers of 4 digits and fewer. For this reason a proof of the ubiquity of odious S-G numbers 1-14 does not translate convincingly to odious numbers 16-31. Looking back to our two previous arguments, the former clearly only applies to 1-15 and the latter is dependant on the majority generated by the former! Prooving that odious numbers 16-31 too are more common than their evil brethren is a subject of active research.

Because our second argument requires only that one odious number be proven common, we can show that a single odious number is common and thereby explain a trend in many following numbers. We can prove 32 is common by the argument that because it is the lowest 6 digit number, no combination of lesser S-G numbers can nim-sum to it. Furthermore, the 5 evil numbers that nim-sum to 32 can be proven to be rare because 32 is reached before them (it must, because for a higher S-G number to be present it must have a follower graph of S-G 32 which must involve 32 itself or yet another > 32 S-G one-legged caterpillar and a path). If 32 is common (at least early on, because it is reached first) and can nim-sum with a common odious path to form these evil S-G numbers, then they must be rare. From the rarity of these evil numbers, we can show that other odious numbers below 48 should be common (they are) and from these that the other evil numbers below 48 should be rare (they are).

Unusually, but perhaps predictably, we can show that 48 is common in the same way. The two 1's in 48 are unreachable by paths (paths can have 1's in up to the '8' position, these are in the 16 and 32 positions), and our one-legged caterpillars so far have only had a 1 in one of these two positions. Like 32, 48 is unreachable by nim-summing a path and a lower S-G one-legged caterpillar. Unlike 32, 48 is an evil number! Using the same arguments we used with 32 we can show that the next 5 odious numbers above 48 are rare, and other evil numbers in this range are more common. We would predict that this trend reverses again at S-G 64, but the highest number in our one-legged caterpillar data is 62, so this cannot be confirmed.

Tri-path data includes S-G data up to 80, and it seems to confirm the same trends we would expect based on this reasoning. Further analysis of this data is a subject of active research.

8. Graph Nim on G-paths

After witnessing the eventual periodicity of the S-G numbers for caterpillars, certain multi-edge paths, and tri-paths, a natural question to pose is: Will the S-G numbers of any similarly "shaped" graph eventually become periodic? We do not yet not know the answer to this question. In this section, we present the progress we have made in this area. We start by defining a couple terms.

Definition. *We call path appended to one vertex of a graph G, a G-path. We say that a G-path has length L if the appended path has L edges, and write G_L.*

If we make a move on a G-path that deletes any of the edges of the original graph G, we say that we have made a move on G. Conversely, if we make a move that deletes only edges from the appended path P_L, we say that we have made a move on P_L.

We note that caterpillars, loop-paths, and triangle-paths are all unified under the definition of a G-path. And so we now provide sufficient conditions for the S-G numbers of a G-path of growing length to stay periodic once an initial period has developed.

Theorem 1. *Suppose we have a G-path whose S-G numbers have started exhibiting a period of p some time before the G-path has reached length L. Suppose also that all G'-paths have become periodic for every follower graph G' by the time they reach length L. Let $d := lcd\{p' | p'$ is the period of the S-G numbers of a G'-path$\}$, and let $E := lcd\{p, d\}$. Then if the G-path's S-G numbers stay periodic past length $L + 72 + E$, they will stay periodic forever.*

Proof. We seek to show that $g(G_{L+72+k}) = g(G_{L+72+k+nE})\ \forall\, n \in \mathbb{N},\ 0 \leq k < E$. To do this, we will show that any S-G number obtainable by a move on G_{L+72+k} is obtainable by a move on $G_{L+72+k+nE}$, and vice versa. Consider three positions of where the moves can be made:

Case 1: Moves made on G yield identical S-G numbers for G_{L+72+k} and $G_{L+72+k+nE}$, since $L+72+k \equiv L+72+k+nE \pmod{d}$, and all G'-paths have already become periodic by the time they reach length L.

Case 2: If we make a move on P_{L+72+k}, leaving G_s and P_t where $s+t = L+72+k-1$ or $s+t = L+72+k-2$ and $0 \leq s < L$, make a move on $P_{L+72+k+nE}$ that leaves G_s and P_{t+nE}. Since $g(P_t) = g(P_{t+nE})$, the two disjoint components resulting from each move nimsum to the same number.

Conversely, if we make a move on $P_{L+72+k+nE}$, leaving G_s and P_t, where $s+t = L+72+k+nE-1$ or $s+t = L+72+k+nE-2$ and $0 \leq s < L$, make a move on P_{L+72+k} that leaves G_s and P_{t-nE}. Since $g(P_t) = g(P_{t-nE})$, the two disjoint components resulting from each move nimsum to the same number.

Case 3: If we make a move on P_{L+72+k}, leaving G_s and P_t where $s+t = L+72+k-1$ or $s+t = L+72+k-2$ and $L \leq s$, make a move on $P_{L+72+k+nE}$ that leaves G_{s+nE} and P_t. Since $g(G_s) = g(G_{s+nE})$, the two disjoint components resulting from each move nimsum to the same number.

Conversely, if we make a move on $P_{L+72+k+nE}$, leaving G_s and P_t, where $s+t = L+72+k+nE-1$ or $s+t = L+72+k+nE-2$ and $L \leq s$, make a move on P_{L+72+k} that leaves G_{s-nE} and P_t. Since $g(G_s) = g(G_{s-nE})$, the two disjoint components resulting from each move nimsum to the same number.

Since these three cases cover all the possible moves we can make on the two G-paths, we conclude that $g(G_{L+72+k}) = g(G_{L+72+k+nE})\ \forall\, n \in \mathbb{N},\ 0 \leq k < E$. And since $p|E$, the G-path retains its period p.

□

We still hope to be able to show that any G-path's S-G numbers will become periodic. However, so far we have only been able to establish a condition that will ensure eventual periodicity.

Theorem 2. *The S-G numbers of a G-path become periodic if and only if they are bounded.*

Proof. ($\Rightarrow$) Immediate.

($\Leftarrow$) Suppose the S-G numbers for a G-path are bounded above by N. Suppose also that $e(G) = m$, and that all G'-paths with $G' \leq G$ and $e(\text{G}') < m$, are eventually periodic. Since we know that the empty graph G$_o$ (with $e(G_o) = 0$) is a subgraph of G, and that a G_o-path is simply a path, we have a valid base case for our inductive assumption on $e(\text{G})$. Lastly, let $d = lcd\{p : p \text{ is the period of a } G'\text{-path}\}$. Note that since $G_o \leq G$, and a G_o-path has eventual period 12, 12|d. And so $g(P_m) = g(P_{m+dt})\ \forall t \in \mathbb{N}$.

Consider any 72 consecutive G-paths, $G_q, G_{q+1}, ..., G_{q+71}$; call these G-paths a 72-block. Since there are at most N^{72} ways for the 72 S-G numbers to be distributed over the 72-block, and since there are infinitely many disjoint 72-blocks, there must exist a 72-long sequence of consecutive S-G numbers that is repeated infinitely many times. Call this sequence $a_1, a_2, ..., a_{72}$.

Now consider G-paths (both on the same graph G) of length r and s, with $r + 72 \leq s$ and $s \equiv r \pmod{d}$, such that $g(G_r) = a_{72} = g(G_s), g(G_{r-1}) = a_{71} = f(G_{s-1}), ..., g(G_{r-71}) = a_1 = g(G_{s-71})$. In other words, we have two sequences of S-G numbers, one of length r and one of length s, that both terminate with "tails" of 72 identical numbers. Choose these G-paths such that all G$'$-paths have already become periodic by the time they reach length r. Also choose these two G-paths such that for every G-path G_k, with $r < k < s - 71$, $\exists$ G_j with $0 \leq j < r - 71$ and $j \equiv k \pmod{d}$, such that $g(G_k) = g(G_j)$. Note that we are guaranteed to have two G-paths that meet these requirements.

We argue that the S-G number computed for the G-paths of length $r + 1$ and $s + 1$ must be identical to each other, by showing that any S-G number obtained by a move on G_{r+1} can be obtained by a move on G_{s+1}, and vice versa. Consider three cases of where the moves can be made:

Case 1: Moves made on G yield identical S-G numbers for G_{r+1} and G_{s+1}, since $r+1 \equiv s+1 \pmod{d}$, and all G'-paths have already become periodic by the time they reach length r.

Case 2: If we make a move on P_{r+1}, leaving G_k and P_v where $k + v = r - 1$ or $k + v = r$ and $0 \leq k < r - 71$, make a move on P_{s+1} that leaves G_k and P_{v+dt} for some $t \in \mathbb{N}$. Since $g(P_v) = g(P_{v+dt})$, the two disjoint components resulting from each move nimsum to the same number.

Conversely, if we make a move on P_{s+1}, leaving G_k and P_v, where $k + v = s - 1$ or $k + v = s$ and $r < k < s - 71$, by our choice of G_{r+1} and G_{s+1} $\exists$ G_j with $0 \leq j \leq r$ and $j \equiv k \pmod{p}$, such that $g(G_j) = g(G_k)$. So, consider the move on G_{r+1} that leaves G_j and P_{v+dt} for some $t \in \mathbb{N}$. Since $g(P_v) = g(P_{v+dt})$, the two disjoint components resulting from each move nimsum to the same number.

Case 3: If we make a move on P_{r+1}, leaving G_{r-i} and P_v with $0 \leq i \leq 71$, make a move on G_{s+1} that leaves G_{s-i} and P_v. Since $g(G_{s-i}) = g(G_{r-i})$, the two disjoint components resulting from each move nimsum to the same number.

Conversely, if we make a move on P_{s+1}, leaving G_{s-i} and P_v with $0 \leq i \leq 71$, make a move on G_{r+1} that leaves G_{r-i} and P_v. Since $g(G_{r-i}) = g(G_{s-i})$, the two disjoint components resulting from each move nimsum to the same number.

Since these three cases cover all of the available moves that we can make on G_{r+1} and G_{s+1}, we conclude that $g(G_{r+1}) = g(G_{s+1})$. By inductively extending this argument, we see that $g(G_{r+t}) = g(G_{s+t})$ $\forall t \in \mathbb{N}$. And so the G-path's S-G numbers become periodic. $\square$

We now present a result about the periodicity of S-G numbers of what we call a GH-path. We start, of course, by defining a GH-path.

Definition. *Let a GH-path be two graphs, G and H, joined together by a path that meets each graph at exactly one vertex. We write GH_L to be a GH-path where the path has length L.*

Theorem 3. *If the S-G numbers of a G-path and an H-path become periodic, then the SG-numbers of the GH-path (where the path is appended to the same vertices of G and H is it is for the G-path and H-path) will also become periodic.*

Proof. We proceed by induction on the total number of edges in G and H, m_G and m_H. For $m_G + m_H = 0$, the statement is trivial, as the

GH-path is just a path.

So assume for all G-paths and H-paths with periodic S-G numbers and with $m_G + m_H < M$, the S-G numbers of the GH-path become periodic, and consider a GH-path with $m_G + m_H = M$. Let $d = lcd\{p' : p'$ is the period of the S-G numbers of a HG'-path or a GH'-path$\}$. Let $E = d \cdot p_H \cdot p_G$, where p_H is the periodicity of the S-G numbers of the H-path and likewise for p_G. Lastly, let H_s and G_r be the shortest H-path and G-path whose S-G numbers have become periodic. Without loss of generality assume $s > r$.

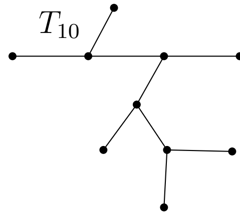
Now consider GH_{2s+E+x} and $GH_{2s+2E+k}$ for any $k \in \mathbb{N}$. By a similar argument as was made in proving the preceding two theorems, we can show that any S-G number obtained by a move on GH_{2s+E+k} can be obtained by a move on $GH_{2s+2E+k}$, and vice versa. Thus, $g(GH_{2s+E+k}) = g(GH_{2s+2E+k})\ \forall\, k \in \mathbb{N}$. And so the GH-path's S-G numbers become periodic. $\square$

9. Nim on Trees

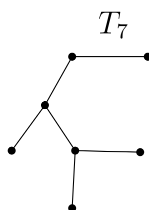
Definition. *A tree is a connected graph that contains no cycles. A tree with n vertices can be denoted* T_n

We can define a caterpillar as a tree such that when all leaves and incident edges are removed the remaining graph is a path. Thus the game of nim played on a tree is similar to that played on a caterpillar. However, the number of trees expands greatly, as does the number of possible moves.

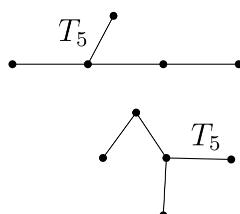
Consider the following tree:



One follower of this tree simply results in another tree:



However, sometimes the follwers of tree are actually forests (i.e. a set of trees):



When this situation occurs, we must nim sum the S-G numbers of the trees in the forest to obtain the actual S-G number of this follower.

Because our trees can become increasingly complex as the number of vertices grow, our algorithm for finding the S-G numbers of caterpillars must evolve so that it can calculate the S-G numbers of these more complex trees. To do this we implemented Sage-specific tools.

A built-in Sage function was used to generate a list of all trees with a given number of vertices.

This function generates unlabled trees. Suppose for a given number of vertices n, we are given all isomorphism classes of trees of n vertices (i.e. all labeled trees that are isomorphic to each other). Our set of unlabled trees consists of only one tree from each isomorphism class. This has a significant effect on decreasing the runtime of our program for calculating the S-G numbers of trees. With this set of trees, we can calculate the S-G number of each non-isomorphic tree.

Iterating through each tree on n vertices, we obtain all possible moves for each tree. This is done by analyzing each vertex of the tree individually and obtaining the set of moves for the specific vertex. We iterate through this set of moves to obtain all followers. Again, note that these followers may actually be forests containing multiple trees. The S-G numbers are then calculated for these followers, nim-summing when appropriate, and added to a set of all S-G numbers for the tree being analyzed. We find the minimal excluded value of this set, which

is the S-G number of the original tree. We calculate this value for all trees with k edges and then we then repeat the process for trees with k+1 edges.

Because the number of trees for a given number of vertices increases at an expoential rate as the number of vertices increases, this code was parallelized. To run the parallelized code, we used the Condor System (or Palmetto Cluster) which is a High-Throughput Computing envirnonment. If certain segments of code do not depend on other segments involved, Condor can be used to significantly decrease the runtime of the code. The Palmetto Cluster was vital in this part of the research. To parallelize the code we divided the set of trees for a given number of vertices into smaller subsets. This was possible because while the S-G number for a tree of n vertices depends on S-G numbers of trees with vertices less than n, it does not depend on the S-G numbers of other trees with n vertices. Thus, we were able to run these subsets of trees on multiple machines. Separate code was then written to combine these subsets and add the complete set to the original set of S-G numbers. See Appendix A, 13.2 for more details.

In our code, we generated each set of trees with n vertices for each separate subset of trees. Upon reaching trees with 19 vertices, we began generating 317,955 trees per subset. This caused the memory of each machine in the Palmetto Cluster to overload. Due to this, we have not been able to completely analyze the S-G numbers of trees.

Thus far, we have noticed that the first player 1 loss occurs at 10 vertices. Of the 106 trees of 10 vertices, 16 are player 1 losses. We have also noticed that there are player 1 losses after 10 vertices, but more computation will be required to extend our analysis.

10. Graph Nim on Graphs

In this section, we discuss the most general form of Graph Nim, the game played on any n-vertex graph. We first discuss the question: For which $n \in \mathbb{N}$ is the complete graph on n vertices, K_n, losing. We next discuss the distribution of the S-G numbers of all unlabeled graphs of order n. Finally, we present a heuristic analysis of the S-G number distributions for labeled graphs of order n.

10.1. Winning and Losing Complete Graphs.

Definition. *A complete graph on n vertices, denoted K_n, is the simple graph in which every pair of distinct vertices is connected by an edge.*

Note that

$$e(K_n) = \binom{n}{2}$$

At the outset of this project, it was known that K_1 (trivially), K_3, and K_5 were losing graphs; one of our goals was to determine if this losing pattern continued for all K_{2n+1}. Note that since K_1, K_3, and K_5 are losing graphs, K_2, K_4, and K_6 are winning. For instance, if it is Player 1's turn to move on K_6, he can delete the 5 edges incident to a single vertex. Then it will be Player 2's turn to move on K_5, a losing graph. And so Player 1 wins. We can extend this argument to show that there are infinitely many winning complete graphs. Remarkably, we know of no way to prove that there are infinitely many losing complete graphs.

Proposition 1. *There are infinitely many winning complete graphs.*

Proof. If K_n is losing then K_{n+1} will be winning, since on K_{n+1} Player 1 can delete the n edges incident to one vertex, leaving the Player 2 with K_n. Thus at least half of the complete graphs must be winning. □

We give the following definitions to allow us to talk about the relationship between two graphs.

Definition. *We call an addition of edges that are all incident to the same vertex an anti-move. This name comes from the fact that these edges may be deleted in a single move.*

Definition. *We say that a graph G' is a child of the graph G if G' may be obtained from G in a single move (equivalently, if G may be obtained from G' in a single anti-move). Likewise, we call G the parent of G'.*

Since the number of unlabeled graphs on n vertices grows very quickly, it soon becomes impractical to investigate by hand whether K_n is winning or losing. For instance, in order to compute whether K_7 is a win or loss, we need to know the status of all 1043 subgraphs of K_7. This task is best left to a computer. Below we describe the program we wrote to find all n-vertex losing graphs, which we were able to run for $1 \leq n \leq 9$. For a more detailed description of the code, see Appendix A, 13.4.

Program 1. *Our program operates on the fact that every parent of a losing graph is a winning graph. We start by creating a list of losing*

graphs; initially this list consists only of the empty graph on n vertices. We then iterate through the possible edge-numbers of n-vertex graphs, generating lists of all nonisomorphic graphs with $1, 2, 3, ..., \binom{n}{2}$ edges, which we will call $L_1, L_2, L_3, ..., L_{\binom{n}{2}}$.

After L_m has been generated, we make anti-moves on each of the losing graphs (all of which have fewer than m edges) in every way possible that leaves an m-edge graph. Since all m-edge graphs obtained in this manner are the parents of losing graphs, they must be winning; we delete them from L_m. After performing all possible anti-moves on the losing graphs and deleting the resulting graphs from L_m, we will have eliminated all of the winning graphs from L_m. Thus, we add any graphs remaining in L_m to our list of losing graphs, generate L_{m+1}, and repeat the above process. In this fashion, we determine whether every graph on n vertices is winning or losing.

This program computed that K_7 and K_9 are losing graphs (and, by implication, that K_8 and K_{10} are winning graphs). Thus, we have extended the pattern of complete graphs alternating between winning and losing from $n = 6$ to $n = 10$. The results from this program also suggested a pattern to Player 2's strategy when the game is started on K_{2n+1}. However, we were unable to prove that Player 2 always has a winning strategy. We hope to look into this matter further in the future.

10.2. **The Sprague-Grundy Approach.** Rather than directly computing a list of the losing graphs on n vertices, we now discuss the more refined approach of computing the S-G numbers of all n-vertex graphs. Since those graphs with S-G number 0 are the losing graphs, this approach encompasses the approach of the previous section. However, what is gained in information is lost in efficiency; computing the S-G numbers of every n-vertex graph is a more demanding computational task than determining which graphs are losing. The program that finds the S-G numbers of n-vertex graphs operates similarly to the program that finds the S-G numbers for trees, and so we will provide just a brief description of how it works.

Program 2. *Assuming we have calculated the S-G numbers for all graphs with fewer than m edges, we describe how we will calculate the S-G numbers of the m-edge graphs. For each graph G with m edges, we generate all the children of G. We then find the minimal excluded number of the set of the S-G numbers of these child graphs; this will be*

the S-G number of G. After computing the S-G numbers of all m-edge graphs in this manner, we move up to $(m+1)$-edge graphs, and continue likewise until the S-G number of every graph on n vertices has been computed.

See Figure 1 below for a chart of the distribution of the S-G numbers for 7-vertex graphs (for the complete data, see Appendix E). The number in column m and row s indicates the percentage of 7-vertex graphs of size m whose S-G number is s. For instance, 50% of size-2 graphs have S-G number 2, 0% have S-G number 1, and 50% have S-G number 0.

Figure 1.

*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	100	0	50.0	20.0	20.0	19.0	12.2	12.3	8.2	9.2	4.1	5.4	8.4	6.2	0	4.9	4.8	10.0	20.0	0	0	100
1	*	100	0	20.0	20.0	23.8	9.8	4.6	17.5	6.1	2.0	3.4	12.2	4.1	0	7.3	0	20.0	0	0	0	0
2	*	*	50.0	0	30.0	9.5	4.9	13.8	5.2	5.3	4.1	8.1	3.1	5.2	6.2	2.4	0	0	0	0	0	0
3	*	*	*	60.0	0	0	12.2	16.9	4.1	4.6	7.4	4.1	3.1	12.4	6.2	4.9	0	0	0	0	0	0
4	*	*	*	*	30.0	4.8	4.9	9.2	15.5	5.3	2.7	2.0	6.9	4.1	6.2	0	23.8	10.0	0	0	0	0
5	*	*	*	*	*	42.9	0	0	5.2	18.3	8.1	1.4	1.5	2.1	9.2	12.2	4.8	0	0	50.0	0	0
6	*	*	*	*	*	*	56.1	0	1.0	0.8	8.1	19.6	3.8	0	0	9.8	0	0	0	50.0	0	0
7	*	*	*	*	*	*	*	43.1	0	0.8	0.7	9.5	9.9	5.2	0	0	0	10.0	40.0	0	0	0
8	*	*	*	*	*	*	*	*	43.3	0.8	0	0	3.1	12.4	6.2	2.4	4.8	0	0	0	100	0
9	*	*	*	*	*	*	*	*	*	48.9	0	0	0	3.1	10.8	17.1	4.8	0	0	0	0	0
10	*	*	*	*	*	*	*	*	*	*	62.8	0	0	0	0	7.3	9.5	10.0	0	0	0	0
11	*	*	*	*	*	*	*	*	*	*	*	46.6	0	0	0	0	14.3	0	0	0	0	0
12	*	*	*	*	*	*	*	*	*	*	*	*	48.1	1.0	0	0	0	20.0	0	0	0	0
13	*	*	*	*	*	*	*	*	*	*	*	*	*	44.3	1.5	0	0	0	0	0	0	0
14	*	*	*	*	*	*	*	*	*	*	*	*	*	*	53.8	0	0	0	0	0	0	0
15	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	31.7	0	0	0	0	0	0
16	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	33.3	0	0	0	0	0
17	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	20.0	0	0	0	0
18	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	40.0	0	0	0
19	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	0	0	0
20	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	0	0
21	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	0

A few observations regarding the S-G number distributions can be made immediately. For instance, as was true with trees, graphs have the maximum possible S-G number (i.e., $e(G) = g(G)$) roughly half of the time. Also, for graphs of size m, the percentage of S-G numbers tends to peak at $s \approx 0.6m$, before bottoming out to nearly zero for $0.6m < s < m$. In order to understand these and other patterns in the S-G number distribution, we conducted a heuristic analysis of the data.

10.3. **Heuristic Analysis of S-G Number Distribution.** Given the distribution of the S-G numbers for n-vertex graphs of size $0, 1, \ldots, m-1$, it would be ideal if we could predict what the distribution should be for graphs of size m. This would mean that we understand why the S-G numbers are distributed as they are. Here we present a heuristic method for making this prediction.

Throughout this section, we will be trying trying to predict the percentage distribution of the S-G numbers for labeled graphs of order n and size m, given that we know the distributions for graphs of order n and size $< m$. Working with labeled graphs has the advantage of allowing us to consider the deletion of any two distinct subsets of edges to be distinct moves, regardless of graph isomorphism. This is not the case when working with unlabeled graphs.

In order to predict the S-G number distribution for labeled graphs, we ask ourselves the question: For the "typical" n-vertex, m-edge labeled graph, $G_{n,m}$, what is the probability that the S-G number will be $0, 1, \ldots, m$? Let us make the notion of a "typical" labeled graph somewhat more precise.

Definition. *We define a typical labeled graph on n vertices with m edges, $G_{n,m}$, to be a graph whose degree sequence is the average of the degree sequences of all n-vertex, m-edge labeled graphs when the degrees are ordered from least to greatest. We write the degree sequence of $G_{n,m}$ as $(d_1, d_2, \ldots, d_n)$, where $d_1 \le d_2 \le \ldots \le d_n$.*

Once we have calculated the degree sequence of $G_{n,m}$ (for details about how this is done, see Appendix A, 13.5), we will know how many children of size $(m-1), (m-2), \ldots, (m-d_n)$ the typical m-edge graph has. We make the (admittedly flawed) assumption that each of these children of size $(m-k)$ for $1 \le k \le d_n$ will be a random labeled graph of size $(m-k)$. I.e., a child graph of size $(m-k)$ will have the S-G number $0, 1, \ldots, (m-k)$ with probability equal to the percentage of labeled $(m-k)$-edge graphs whose S-G numbers are

$0, 1, \ldots, m - k$. Since we know the number of children, we can can calculate the probability that $G_{n,m}$ will have S-G number $0, 1, \ldots, m$.

In order to run this heuristic, we need the distribution of S-G numbers for labeled graphs of size $m - k$. We find this by counting each graph's S-G number N times, where N is the number of unique relabelings of the graph (i.e., the size of the graph's isomorphism class). See Figure 2 below for a chart of the distribution of S-G numbers for labeled 7-vertex graphs:

Figure 2.

*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	100	0	50.0	2.6	12.3	14.1	9.0	6.7	6.9	4.2	2.1	4.2	6.1	3.9	0	5.2	6.2	10.5	2.6	0	0	100
1	*	100	0	7.9	24.6	13.9	4.7	4.3	16.8	3.8	0.3	0.8	9.4	3.0	0	2.0	0	10.5	0	0	0	0
2	*	*	50.0	0	33.3	8.3	0.6	18.4	3.1	1.7	2.2	7.3	3.6	3.0	2.3	1.2	0	0	0	0	0	0
3	*	*	*	89.5	0	0	9.5	15.1	2.8	3.7	6.1	3.1	1.0	9.1	3.5	0.8	0	0	0	0	0	0
4	*	*	*	*	29.8	12.4	3.5	6.7	12.2	2.6	2.0	1.5	6.6	1.0	1.7	0	11.2	1.8	0	0	0	0
5	*	*	*	*	*	51.3	0	0	3.4	22.4	5.1	0.5	0.2	1.9	5.1	14.5	0.5	0	0	50.0	0	0
6	*	*	*	*	*	*	72.6	0	0.6	0.0	5.3	16.0	0.9	0	0	9.4	0	0	0	50.0	0	0
7	*	*	*	*	*	*	*	48.8	0	0.1	0.7	12.7	10.1	1.8	0	0	0	1.8	39.5	0	0	0
8	*	*	*	*	*	*	*	*	54.1	0.9	0	0	2.3	15.1	3.3	0.1	6.2	0	0	0	100	0
9	*	*	*	*	*	*	*	*	*	60.7	0	0	0	4.3	9.6	14.3	1.5	0	0	0	0	0
10	*	*	*	*	*	*	*	*	*	*	76.1	0	0	0	0	3.4	7.2	21.1	0	0	0	0
11	*	*	*	*	*	*	*	*	*	*	*	53.9	0	0	0	0	10.3	0	0	0	0	0
12	*	*	*	*	*	*	*	*	*	*	*	*	59.9	0.2	0	0	0	12.3	0	0	0	0
13	*	*	*	*	*	*	*	*	*	*	*	*	*	56.8	2.2	0	0	0	0	0	0	0
14	*	*	*	*	*	*	*	*	*	*	*	*	*	*	72.4	0	0	0	0	0	0	0
15	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	49.1	0	0	0	0	0	0
16	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	56.8	0	0	0	0	0
17	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	42.1	0	0	0	0
18	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	57.9	0	0	0
19	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	0	0	0
20	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	0	0
21	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	0

Now we will define a few new terms, and describe in more detail our heuristic method. For a more complete description of the code, see Appendix A, 13.6.

Definition. *We write $P_e(s)$ to be the probability that a random labeled graph with e edges will have S-G number s. Similarly, we write $P_{G_{n,m}}(s)$ to be the probability that our typical graph $G_{n,m}$ has S-G number s.*

Definition. *Let c_k to be the number of children of $G_{n,m}$ with $m-k$ edges.*

Definition. *We write $N_e(s,t_e)$ to be the probability that given t_e random labeled graphs with e edges, none of them will have S-G number s. Note that $N_e(s,t_e) = (1-P_e(s))^t$. Also note that $N_e(s,t_e) = 0$ if $s > e$, since $g(H) \leq e(H)$ for all graphs H.*

Definition. *Lastly, we write $N_{G_{n,m}}(s)$ to be the probability that $G_{n,m}$ has no children with S-G number s.*

Program 3. *For the purposes of this heuristic, we assume that the probability that a child of $G_{n,m}$ with $(m-k)$ edges has S-G number s, is equal to $P_{(m-k)}(s)$. Take $1 \leq j \leq n$ to be the maximum index s.t. $m - d_j \geq s$. Then note that*

$$N_{G_{n,m}}(s) = N_{m-1}(s, c_1) \times \ldots \times N_{m-d_j}(s, c_{d_j})$$

We know that $P_{G_{n,m}}(s)$ is equal to the probability that $G_{n,m}$ has children with S-G numbers $0, 1, \ldots, (s-1)$ and no children with S-G number s. Thus, we have

$$P_{G_{n,m}}(s) = \{1 - N_{G_{n,m}}(0)\} \times \ldots \times \{1 - N_{G_{n,m}}(s-1)\} \times N_{G_{n,m}}(s)$$

From the above equations, we see that if we can find the number of children of $G_{n,m}$, $c_1, \ldots, c_{d_n}$, we will know the probability that $G_{n,m}$ has S-G number 1 through m. But since $G_{n,m}$ is labeled, computing $c_1, \ldots, c_{d_n}$ is a simple task.

We admit that some of the assumptions on which we base our heuristic are naive. We are currently working to refine our heuristic method. And yet, this method yields reasonably accurate predictions of the S-G number distributions. See Figure 3 below for a table of our heuristic predictions of the S-G number distributions compared to the actual distributions for $n = 7$ and $m = 4, 5, \ldots, 18$. Because of our method's ineffectiveness for values of m near 0 and $\binom{7}{2}$, we omit these data. For the complete results of our heuristic analysis for graphs of order 5, 6, 7, and 8, see Appendix E.

Figure 3.

edges	*type*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
4	*heuristic*	22.5	55.8	5.4	0	16.3	*	*	*	*	*	*	*	*	*	*	*	*	*	*
4	*computed*	12.3	24.6	33.3	0	29.8	*	*	*	*	*	*	*	*	*	*	*	*	*	*
5	*heuristic*	22.7	12.5	4.3	0	10.3	50.2	*	*	*	*	*	*	*	*	*	*	*	*	*
5	*computed*	14.1	13.9	8.3	0	12.4	51.3	*	*	*	*	*	*	*	*	*	*	*	*	*
6	*heuristic*	17.8	5.7	4.0	7.6	3.5	0.8	60.6	*	*	*	*	*	*	*	*	*	*	*	*
6	*computed*	9.0	4.7	0.6	9.5	3.5	0	72.6	*	*	*	*	*	*	*	*	*	*	*	*
7	*heuristic*	10.1	9.5	15.7	32.2	3.8	0	0	28.7	*	*	*	*	*	*	*	*	*	*	*
7	*computed*	6.7	4.3	18.4	15.1	6.7	0	0	48.8	*	*	*	*	*	*	*	*	*	*	*
8	*heuristic*	5.4	10.4	6.0	5.2	8.1	0.9	0	0.3	63.7	*	*	*	*	*	*	*	*	*	*
8	*computed*	6.9	16.8	3.1	2.8	12.2	3.4	0.6	0	54.1	*	*	*	*	*	*	*	*	*	*
9	*heuristic*	7.2	5.2	1.8	2.0	5.5	27.9	0	0	0	50.3	*	*	*	*	*	*	*	*	*
9	*computed*	4.2	3.8	1.7	3.7	2.6	22.4	0.0	0.1	0.9	60.7	*	*	*	*	*	*	*	*	*
10	*heuristic*	6.4	0.9	5.4	5.4	2.2	3.2	5.1	0.1	0	0	71.3	*	*	*	*	*	*	*	*
10	*computed*	2.1	0.3	2.2	6.1	2.0	5.1	5.3	0.7	0	0	76.1	*	*	*	*	*	*	*	*
11	*heuristic*	2.9	0.5	5.8	2.4	1.4	0	11.6	0.6	0	0	0	74.7	*	*	*	*	*	*	*
11	*computed*	4.2	0.8	7.3	3.1	1.5	0.5	16.0	12.7	0	0	0	53.9	*	*	*	*	*	*	*
12	*heuristic*	7.0	9.0	7.6	2.2	7.0	0	1.4	5.2	0.2	0	0	0	60.4	*	*	*	*	*	*
12	*computed*	6.1	9.4	3.6	1.0	6.6	0.2	0.9	10.1	2.3	0	0	0	59.9	*	*	*	*	*	*
13	*heuristic*	2.3	7.9	1.1	2.1	6.1	0.6	0	0.1	11.2	0	0	0	0	68.7	*	*	*	*	*
13	*computed*	3.9	3.0	3.0	9.1	1.0	1.9	0	1.8	15.1	4.3	0	0	0.2	56.8	*	*	*	*	*
14	*heuristic*	0.6	0.6	0.7	2.3	1.8	16.9	0.1	0	2.7	6.2	0	0	0	0	68.1	*	*	*	*
14	*computed*	0	0	2.3	3.5	1.7	5.1	0	0	3.3	9.6	0	0	0	2.2	72.4	*	*	*	*
15	*heuristic*	0.1	0.1	0.2	0	0.6	6.2	0.3	0	0	0.7	0	0	0	0	0	91.9	*	*	*
15	*computed*	5.2	2.0	1.2	0.8	0	14.5	9.4	0	0.1	14.3	3.4	0	0	0	0	49.1	*	*	*
16	*heuristic*	0.4	0.5	0.5	0	1.9	0.1	2.9	0.3	0	0	12.8	0.1	0	0	0	0	80.5	*	*
16	*computed*	6.2	0	0	0	11.2	0.5	0	0	6.2	1.5	7.2	10.3	0	0	0	0	56.8	*	*
17	*heuristic*	0	1.2	0.8	0	0.7	0	0	6.4	0	0	2.1	3.0	0	0	0	0	0	85.6	*
17	*computed*	10.5	10.5	0	0	1.8	0	0	1.8	0	0	21.1	0	12.3	0	0	0	0	42.1	*
18	*heuristic*	0	1.3	6.2	1.6	0	0	0	39.6	0	0	0	0	0.8	0	0	0	0	0	50.5
18	*computed*	2.6	0	0	0	0	0	0	39.5	0	0	0	0	0	0	0	0	0	0	57.9

11. Conclusion

There are a number of areas where we would like to continue work. First, we feel that many additional results are readily available through computation. We would like to continue computing the S-G numbers of trees and graphs. We would also like to know whether K_{11} is winning or losing. Lastly, in order to support or disprove our conjecture about their eventual periodicity, we plan on computing the S-G numbers of more complicated types of G-paths.

Second, we feel that we can extend several of the results we present in this paper. For instance, we would like to conduct a more formal analysis of the frequency of evil and odious numbers of one-legged caterpillars. In addition, we would like to see if this evil/odious pattern continues for tri-paths, multi-legged caterpillars, and other types of G-paths. And we would like to improve our heuristic analysis of the S-G numbers of graphs and extend this analysis to trees.

Despite all of our hard work this summer, there are a few questions left unanswered. Do the S-G numbers of all G-paths eventually become periodic? What about graphs with multiple appended paths? If so, will the periods always be of the form $12k$? Will K_{2n+1} always be losing for Player 1? Can much be said about Graph Nim played on complete bipartite graphs? On multigraphs?

This game has spawned a number interesting observations, conjectures, and theorems; hopefully more are soon to come.

12. Acknowledgements

Our group would like to give special thanks to the following:
Neil Calkin and Kevin James for the opportunity to do this research over the summer.
Janine Janoski for being inspirational and having so much patience.
Edward Duffy for help with Sage on the Cluster.
Benoit Larose for advice and helpful suggestions.
All of the other REU participants for support.

13. Appendix A: Program Descriptions

See Appendix B for corresponding code.

13.1. One-Legged Caterpillar Program.

One-Legged Caterpillar Program. Below are the important functions of our Python program that computes the S-G numbers for caterpillars with one extra edge.

- f(x) - Given an upper bound, computes the S-G numbers of all paths up to the upper bound + 3. This function returns the list of S-G numbers. (This is the program used in our initial research to calculate the S-G numbers of paths.
- c(x, gArray, pArray) - Where x is the length of the $C_{n,k}$ to be computed, gArray is the list of S-G numbers for the caterpillars from the from 0 up to the lower bound, and pArray is the S-G numbers of all paths from 0 to the upper bound + 3. This function returns an array consisting of the S-G numbers for all caterpillars of length x.
- h(x) - Takes in an upper bound, computes the path S-G numbers, and gets the array of caterpillar S-G number from a file. Computes S-G numbers for caterpillars up to the upper bound and writes the entire list of S-G numbers for all caterpillars of length 0 to the upper bound to the file that was originally read in.

13.2. **Tree Program.** Below are the important functions of our Sage program that computes the S-G numbers of all trees with a given number of vertices.

- getmoves(lis, k) - Takes in a list of 1's and 0's and a number of vertices k. Returns all possible representations of moves in the form of 0's and 1's.
- getverts(o2, lis) - Given the vertex from which edges are to be deleted and the a list of 1's and 0's from the getmoves function, converts the list of moves to a vertex specific list of moves.
- g(myTree) - Calculates the S-G number of a given tree.

The remainder of the code forms a list of trees for a given number of vertices and generates the list of S-G numbers for that list of trees. The output separated similar degree sequences and then specific trees.

13.3. **Tri-Path Program.** Below are the important functions of our C++ program that computes the S-G numbers tri-paths.

- The main function reads in S-G numbers for one-legged caterpillars and paths. It uses nested loops to check every case and every length up to 1000. It uses a switch statement to find the follower graphs, then locates S-G numbers for these stored in arrays and uses the sum function to nim-sum these when applicable. The resulting S-G numbers are sorted and the mex can then be easily calculated. This is then printed to an output file.
- The sum function reads in two S-G values and nim-sums them, outputting to S-G value of the total graph.

13.4. **Losing Graphs Program.** Below are the important functions of our Sage program that computes a list of losing graphs of order n.

- getGraphs(path) - As input, this function takes the location of a text file of graphs that are stored in graph6 format. It returns a list of graphs as Sage objects. We use this function to retrieve a list of graphs with a specific number of edges and to retrieve the list of losing graphs with $< m$ edges.
- getAntiMoves(list L, int k) - As input, this function takes a list of 1's, L, which represent the number of vertices not adjacent to a vertex, and a number, k, which is the number of edges we want to add to our vertex in an anti-move. This function returns a list of all the possible ways to add these k edges. For instance, getAntiMoves([1,1,1,1], 2) would return [[1,1,0,0]. [1,0,1,0], [1,0,0,1], [0,1,1,0], [0,1,0,1], [0,0,1,1]].
- getVerts(list vertices, list getM) - As input, this function takes a list of vertices that are not adjacent to a vertex and the list that was returned by getAntiMoves. This function returns a list of all the possible anti-moves we can make on the vertex. For instance, getVerts([2,4,6], [[1,1,0], [1,0,1], [0,1,1]]) would return [[2,4], [2,6], [4,6]].
- findLosses(int vertices, int edges, int loss) - This is the main function of the program. Assuming we have already computed all losing graphs of order 'vertices' and size $\leq$ 'loss', this program will compute the losing graphs up to size 'edges'. If no losing graphs have already been computed, one should enter 0 for the value of 'loss'. This function stores the list of losing

graphs of size $(1 + \text{'loss'}) \leq m \leq \text{'edges'}$ in text files.

13.5. **Graph S-G Number Program.** Below are the important functions of our Sage program that computes the S-G numbers of graphs of order n.

- getMoves(list L, int k) - As input, this function takes a list of 1's, L, which represent the number of vertices adjacent to a vertex, and a number, k, which is the number of edges we want to remove from our vertex in a move. This function returns a list of all the possible ways to remove these k edges. For instance, getMoves([1,1,1,1], 2) would return [[1,1,0,0]. [1,0,1,0], [1,0,0,1], [0,1,1,0], [0,1,0,1], [0,0,1,1]].

- getVerts(list vertices, list getM) - As input, this function takes a list of vertices that are adjacent to a vertex and the list that was returned by getMoves. This function returns a list of all the possible moves we can make on the vertex. For instance, getVerts([2,4,6], [[1,1,0], [1,0,1], [0,1,1]]) would return [[2,4], [2,6], [4,6]].

- makeNeat(list graphs, list grunds, int vertices) - As input, this function takes a list of graphs, a list of S-G numbers of those graphs, and the number of vertices. It then chops up the list of graphs by the degrees of three of each graph's vertices, storing the graphs in a list of lists (and the S-G numbers in a corresponding list of lists). It then returns these two chopped up lists of graphs and their S-G numbers. This function's purpose is to make the program run quicker by allowing us to quickly find a graph (with m edges) and its S-G number simply by accessing the degrees of three of its vertices (instead of having to search through the entire list of m-edge graphs).

- findMex(list gruns) - As input, this function takes a list of S-G numbers. It computes and returns the minimal excluded element of this list.

- getSG(vertices): This is the main function of the program. As input, it takes a number of vertices. It returns a list of the graphs of order 'vertices' (separated into sublists by the number

of edges) and the corresponding S-G numbers of these graphs.

13.6. **Graph S-G Heuristic Program.** Below are the important functions of our Sage program that predicts the S-G number distribution for labeled n-vertex, m-edge graphs.

- typicalDegree(int m, list grafs) - As input, this function takes a list of unlabeled graphs with n vertices and a number, m, of edges. It returns the "typical" degree sequence (see Section 10.3) of all labeled n-vertex, m-edge graphs.

- adjust(list graphs, list gruns) - As input, this function takes a list of unlabeled graphs of order n (separated into $\binom{n}{2}+1$ sublists by the number of edges) and their S-G numbers. It returns a list of $\binom{n}{2}+1$ lists, each which indicates the of the number of labeled graphs of size $(0 \leq m \leq \binom{n}{2})$ which have S-G number s $(0 \leq s \leq m)$.

- percentage(list distr) - As input, this function takes the list of the S-G number frequencies that the adjust(graphs, gruns) function returns. It returns a percentage distribution of these S-G numbers. For instance percentage([[1], [3,5], [2, 4, 10]]) would return [[100], [37.5, 62.5], [12.5, 25, 62.5]].

- probCant(list perc, list deg, int m, int s) - As input, this function takes the calculated list of the percentage distributions of the S-G numbers for labeled graphs of order n, a list which is the degree sequence of $G_{n,m}$, the size of $G_{n,m}$, and an S-G number s. It returns the probability that no child of $G_{n,m}$ will have S-G number s, assuming that all children of $G_{n,m}$ are random labeled graphs.

- predictG(list perc, list deg, int m) - As input, this function takes the calculated list of the percentage distributions of the S-G numbers for labeled graphs of order n, a list which is the degree sequence of $G_{n,m}$, and the size of $G_{n,m}$. It returns the predicted percentage distribution of the S-G numbers of labeled graphs of order n and size m.

- generateH(int n): This function is the main function of the program. Using all of the above functions, it generates a list of the

expected S-G number distributions for graphs of order n and size $1, 2, \ldots, \binom{n}{2}$.

14. Appendix B: Code

14.1. One-Legged Caterpillar Code. Before running the caterpillar code, we must run the following:

```
startArray = [[0], [2, 2], [3, 3, 3], [1, 4, 4, 1]]
pickle.dump(startArray, open("catOutput.txt", "w"))
```

This code creates a the first pickled file of S-G numbers for paths up to length 3. Once this file is created, one can begin using the following code with lowerBound equal to 4.

```
import pickle
import sys

lowerBound = int(sys.argv[1])
upperBound = int(sys.argv[2])

def f(x):
    grund=[0,1]
    #grund is the ordered list of all the grundy numbers for paths
    #of lengths shorter than x. so 0 is f(0), 1 is f(1), and the program starts
    #computing grundy numbers for paths of length 2 and higher'
    prev=[0]
    #prev is the list of grundy numbers for all positions attainable in one move
    #from a path of length x
    while x<upperBound+3:
        prev.append(grund[x-1])
        prev.append(grund[x-2])
        #clearly the path of length x-1 and length x-2 are attainable in one move
        #from path of length x, so right away we add the grundy numbers of these
        #two shorter paths into prev
        y=1
        while x-1-y>=0:
            #Adds the nimsum of the f-g values after removing 1 vertex to prev
            binsum=(grund[y])^(grund[x-1-y])
            prev.append(binsum)
            y+=1
        y=1
        while x-2-y>=0:
            #Adds the nimsum of the f-g values after removing 2 vertices to prev
            s=(grund[y])^(grund[x-2-y])
            prev.append(s)
```

```
            y=y+2
        y=0
        while y<len(prev):
            #this while loop finds the minimal excluded element of prev

            if prev.count(y)>0:
                y+=1
            else:
                grund.append(y)
                y=y+len(prev)
        prev=[0]
        x+=1
    return grund

def c(x, gArray, pArray):
#takes in a given number of vertices x, the list of all grundy numbers of
#caterpillars of length small than x, and the grundy numbers of paths.
    grundyNumbers = gArray
    pathNumbers = pArray
    grundyNumbers.append([])
    grundyNumbers[x].append(pathNumbers[x+1])
    #iterates through each vertex in the caterpillar as to examine
    #each possible case of an extra edge
    for i in range(1, x):
        temp = []
        temp.append((pathNumbers[x-1]))
        #for loops that simulate each possible move to be made on the
        #given caterpillar with an extra edge on index i
        for j in range(1, i+1):
            temp.append((pathNumbers[j-1])^(grundyNumbers[x-j][i-j]))
        for j in range(1, x-i+1):
            temp.append((pathNumbers[j-1])^(grundyNumbers[x-j][i]))
        for j in range(2, i+1):
            temp.append((pathNumbers[j-2])^(grundyNumbers[x-j][i-j]))
        for j in range(i+2, x+1):
            temp.append((pathNumbers[x-j])^(grundyNumbers[j-2][i]))
        for j in range(i, i+1):
            temp.append((pathNumbers[i-1])^(pathNumbers[x-i-1])^1)
        for j in range(i, i+1):
            temp.append((pathNumbers[j-1])^(pathNumbers[x-j+1]))
            temp.append((pathNumbers[j+1])^(pathNumbers[x-j-1]))
            temp.append((pathNumbers[j-1])^(pathNumbers[x-j-1]))
            temp.append((pathNumbers[j-1])^(pathNumbers[x-j]))
            temp.append((pathNumbers[j])^(pathNumbers[x-j-1]))
        mex=0
        z=0
        #finds the minimal excluded value of the set of all grundy numbers
        while z<len(temp):
```

```
            if temp.count(mex)>0:
                mex+=1
                z+=1
            else:
                z=len(temp)+1
        grundyNumbers[x].append(mex)
        temp = []
    grundyNumbers[x].append(pathNumbers[x+1])
    #returns all grundy numbers for a given x
    return grundyNumbers[x]


def h(x):
    #takes in a given x to calculate the sprague-grundy numbers
    #up to that x (inclusive)
    pathNums = f(2)
    #loads grundy numbers that have been computed
    #up to the lower bound (inclusive)
    gnums = pickle.load(open("catOutput.txt"))
    q = lowerBound+1
    sortedGr = []
    #appends next array for grundy numbers
    for i in range(x+1):
        sortedGr.append([])
    #iterates through the possible lengths of caterpillars
    #starting with lowerBound+1 and ending at upperBound
    while q < (x+1):
        temp = c(q, gnums, pathNums)
        q+=1
        #appends new grundy numbers to list
        for i in range((len(temp))):
            sortedGr[i].append(temp[i])
    #writes new list of grundy numbers to file
    pickle.dump(gnums, open("catOutput.txt", "w"))


h(upperBound)
```

14.2. **Tree Code.**

```
def getmoves(lis, k):
#returns a list representing the unique ways to delete k
#edges from an inputed number of edges
    remain=0
    temp = []
    shorter = []
    ret = []
    if k == 0:
        for i in range(len(lis)):
            temp.append(0)
```

```
        return [temp]
    current = 0
    if len(lis) == 0:
        return [[]]
    if len(lis) == 1:
        return[[k]]
    if lis[0]<=k:
        mini=lis[0]
    else:
        mini=k
    shorter = lis[:]
    del shorter[0]
    for d in shorter:
        remain+=d
    for j in range(mini+1):
        if remain>=k-mini+j:
            temp = getmoves(shorter,k-mini+j)
            for t in temp:
                ret.append([mini-j] + t)
    return ret

def getverts(o2, lis):
#given the list from getmoves, returns the specific moves
#for a given graph
    ret = []
    for l in lis:
        temp = []
        for ll in range(len(l)):
            if l[ll]==1:
                temp.append(o2[ll])
        ret.append(temp)
    return ret

def g(myTree):
#given a tree, return the Sprague-Grundy number
    myTreeDeg=myTree.degree()
    myTreeDeg.sort()
    GrundyList = []
    #returns 0 if the tree is empty
    if myTree.degree() == [0]:
        return 0
    #returns 1 if the tree is a simple path of length 1
    elif myTree.degree() == [1, 1]:
        return 1
    #returns 2 if the tree is a simple path of length 2
    elif myTreeDeg == [1, 1, 2]:
        return 2
    else:
```

```
#iterates through the vertices of the tree
#to determine all possible moves on each vertex
   for v in myTree:
       moves=[]
       #gets the neighbors (or connected vertices) of v
       t = myTree.neighbors(v)
       lengthNeigh = len(t)
       tempOnes = []
       for i in range(lengthNeigh):
           tempOnes.append(1)
       y=0
       #creates list of possible moves for v
       for i in range(1, myTree.degree(v)+1):
           temp = getmoves(tempOnes, i)
           possMoves = getverts(t, temp)
           for m in possMoves:
               movesTemp=[]
               for num in m:
                   movesTemp.append([v,num])
               moves.append(movesTemp)
       #for each move in moves, finds the resultant
       #followers and grundy number
       for y in range(len(moves)):
           treeM=myTree.copy()
           for M in moves[y]:
               treeM.delete_edge(M)
           graphMoves = treeM.connected_components_subgraphs()
           tempGrund = []
           #creates the list of all sprague-grundy numbers
           #obtainable from one move
           for g in graphMoves:
               g=g.canonical_label()
               tempDegree = g.degree()
               vLen = len(tempDegree)
               tempDegree.sort()
               for i in range(0, len(treeSG[vLen])):
                   if (vLen==0):
                       tempGrund.append(0)
                   else:
                       if tempDegree == (treeSG[vLen][i][0]):
                           for j in range(1, len(treeSG[vLen][i])):
                               if treeSG[vLen][i][j][0] == g:
                                   tempGrund.append(treeSG[vLen][i][j][1])
           grundSum=0
           for gr in tempGrund:
               grundSum=grundSum^^gr
           GrundyList.append(grundSum)
   mex=0
```

```
        z=0
        #from the list of obtainable grundy numbers,
        #finds the minimum excluded value (the grundy
        #number of the original tree)
        while z<len(GrundyList):
            if GrundyList.count(mex)>0:
                mex+=1
                z+=1
            else:
                z=len(GrundyList)+1
        GrundyList=[]
        return mex

#Given some number of vertices k and an list of sprague-grundy
#number for all trees of vertices smaller than k, breaks
#the job into several smaller jobs depdending on the number of
#trees for that number of vertices. Then runs each job on a separate
#machine in the Condor cluster.
jobNum=os.environ['CONDOR_PROCESS']
vertices =18
trees = []
u='treeSG17.sobj'
treeSG=load(u)
subTreeSG=[]
start=int(jobNum)*1000
stop=(int(jobNum)+1)*1000
if stop > 123867:
    stop=123867
treeDegrees = []
for v in range(vertices,vertices+1):
#gets the list of all trees for a given number
#of vertices, then calculates the grundy number
#for the given range.
    T=list(graphs.trees(v))
    for q in range(start, stop):
        t=T[q]
        t=t.canonical_label()
        x = t.degree()
        x.sort()
        grundyNum =g(t)
        #appends tree and grundy number if SG list is empty
        if len(subTreeSG)<1:
            subTreeSG.append([x])
            subTreeSG[0].append([t, grundyNum])
        else:
            aCounter=0
            #determines if the degree sequence is already
            #in the SG list
```

```
            for d in range(len(subTreeSG)):
                if subTreeSG[d][0] == x:
                    aCounter=1
                    appendV=d
            #if degree sequence is in list, appends tree and
            #SG number to array with that degree sequence
            if aCounter==1:
                subTreeSG[appendV].append([t,grundyNum])
            #appends degree sequence, tree, and SG number
            #if degree sequence does not already appear
            #in list
            else:
                subTreeSG.append([x])
                s = len(subTreeSG)
                subTreeSG[s-1].append([t, grundyNum])

#creates filename depending on job number and saves
#the SG list to that file
filename='subTreeSG'+str(jobNum)
save(subTreeSG, filename)
```

The following gives an example of a class ad used to run this program on Condor:

```
universe                = vanilla
executable              = condor-env.sh
requirements            = Arch=="X86_64" && OpSys=="LINUX"
should_transfer_files   = YES
transfer_executable     = TRUE
arguments               = /opt/sage/4.0.1/sage
input   = treesBroken.sage
environment = CONDOR_PROCESS=$(Process)

transfer_input_files = treeSG18.sobj, testOutput.txt

when_to_transfer_output = ALWAYS
notification            = NEVER

queue 287
```

Because the code is parrallelized, resulting in multiple output files, the following code is required to combine all output (this specific example is for combining the output after the program has been run for 18 vertices):

```
x = 'treeSG17.sobj'
trees = load(x)

treeSG=load('subTreeSG0.sobj')
#loads original S-G numbers
file='subTreeSG1.sobj'
```

```
subTreeSG=load(file)
for i in range (2,125):
        if (i==54 or i==97):
            for k in range (20):
                file='subTreeSG'+str(i)+'.'+str(k)+'.sobj'
                temp=load(file)
                subTreeSG.append(temp)
        else:
            file='subTreeSG'+str(i)+'.sobj'
    temp=load(file)
    subTreeSG.append(temp)

#for loops to append to base array of n vertices
for i in range(len(subTreeSG)):
     counter = 0
     for j in range(len(treeSG)):
         if subTreeSG[i][0] == treeSG[j][0]:
             for k in range(1, len(subTreeSG[i])):
                 treeSG[j].append(subTreeSG[i][k])
                 counter = 1
     if counter == 0:
         treeSG.append(subTreeSG[i])

#appending the completed base array to all SG numbers
trees.append(treeSG)

print trees
save (trees, 'treeSG18')
```

14.3. **Tri-Path Code.**

```
#include <vcl.h>
#include <math.h>
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
using std::ifstream;
#include <string>
#pragma hdrstop
using namespace std;

//---------------------------------------------------------------------------

#pragma argsused

//---------------------------------------------------------------------------
#include <iostream.h>
 int cat[999][499]= {
```

```
{3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
{4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}};
int tri[1000][501];
int sum (int a,int b);
int wait;
void main()
{
//
cout<<"Loading...\n\n";
cat[0][0]=3;
/*int cat[999][500]= {
{3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
{4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}};*/
int path[1001]={1,2,3,1};
int countcat=0;
int row=-1;
int col=0;
int incount=0;
int align=0;
int countcount;
int realign=0;
int yetanotherarray[101];
for (wait=0; wait <= 100; wait++) {yetanotherarray[wait]=0;}
int anothercounta;
int anothercountb;
int anothercountc;
int finalcount;
int justanothertemparray[100];
ofstream grundytriangle;
grundytriangle.open("grundy.txt");
ifstream dudley;
dudley.open("caterpillarold.txt", ios::in);
if (!dudley) {cout<<"Error: Cannot open file"; exit(1);}
string data;
//The following set of loops reads in S-G data for one-legged caterpillars.
while(( !dudley.eof())&&(incount<=2000)){
incount+=1;
getline(dudley,data);
if ((incount%2==1)) {
/*cout<<data.substr(1)<<endl;
cout<<data<<endl;*/
/*cout<<data[0]<<endl;
cout<<data[1]<<endl;
cout<<path[row+3]<<endl;*/
row=0;
if (col==0) {
while (row<1001)
{path[row+4]=(data[(4*(row))]-48);
```

```
row+=1;}
col+=1;}
else
{//cout<<(col-1)<<"\n\n";
countcat=0;
if ((col>=2)&&(col<=4)) align=2;
if (col>4) align=(col-2);
if (col!=1) realign=1;
for (countcount=0; countcount<=(incount/2); countcount+=1)
if (data[4*countcount+countcat+1]!=' ') countcat+=1;
for (row=(2*col);(row<(999+(2*realign)));row+=1)
{if (data[4*row+countcat-7-(4*align)]!=' ') {
cat[row-(2*realign)][col-1]=((10*(data[4*row+countcat-8-(4*align)]-48))+
...(data[4*row+countcat-7-(4*align)]-48));
//cout<<cat[row-(2*realign)][col-1]<<" ";
countcat+=1; yetanotherarray[cat[row-(2*realign)][col-1]]+=1;
}
else {
cat[row-(2*realign)][col-1]=(data[4*row+countcat-8-(4*align)]-48);
yetanotherarray[cat[row-(2*realign)][col-1]]+=1;
}}
//cout<<(data[4*row+countcat-8-(4*align)]-48)<<" ";
col+=1;
align=0;
//cout<<"\n\n";
}}}
//These loops print S-G distribution for one-legged caterpillars.
for (anothercounta=0; anothercounta <= 998; anothercounta+=1) {
for (anothercountc = 0; anothercountc <= 99; anothercountc++)
... justanothertemparray[anothercountc]=0;
for (anothercountb=0; anothercountb <= (anothercounta/2); anothercountb+=1) {
yetanotherarray[cat[anothercounta][anothercountb]]+=1;
... justanothertemparray[cat[anothercounta][anothercountb]]+=1; yetanotherarray[100]+=1;
}
grundytriangle<<anothercounta<<"\n";
for (finalcount=0; finalcount<=99; finalcount+=1)
... grundytriangle<<yetanotherarray[finalcount]<<"graphs had grundy value
... "<<finalcount<<", this represents "<<(100*
... ((double (yetanotherarray[finalcount]))/(double (yetanotherarray[100]))))
... <<"% of all graphs. +"<<justanothertemparray[finalcount]<<"\n\n";
}
//cout<<cat[998][498];
//cin>>row;
//grundytriangle<<data<<endl;
/*
while (!dudley.eof()){
 }
```

```
{{5,5,0,0,0,0,0,0,0,0,0,0,0,0,0},
{6,6,0,0,0,0,0,0,0,0,0,0,0,0,0},
{2,7,2,0,0,0,0,0,0,0,0,0,0,0,0},
{1,8,8,0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,5,3,0,0,0,0,0,0,0,0,0,0,0},
{8,10,2,10,0,0,0,0,0,0,0,0,0,0,0},
{6,6,6,8,3,0,0,0,0,0,0,0,0,0,0},
{0,8,12,5,6,0,0,0,0,0,0,0,0,0,0},
{1,1,1,4,1,4,0,0,0,0,0,0,0,0,0},
{2,6,2,11,14,2,0,0,0,0,0,0,0,0,0},
{3,3,8,3,6,8,3,0,0,0,0,0,0,0,0},
{8,8,9,4,8,1,10,0,0,0,0,0,0,0,0},
{5,5,11,8,0,4,5,4,0,0,0,0,0,0,0},
{12,7,14,10,3,3,3,14,0,0,0,0,0,0,0},
{7,3,6,23,6,6,2,2,2,0,0,0,0,0,0},
{1,8,0,8,8,8,1,13,1,0,0,0,0,0,0},
{0,0,5,16,15,10,0,0,4,4,0,0,0,0,0},
{8,10,14,13,10,13,16,15,10,8,0,0,0,0,0},
{9,13,12,8,3,7,8,6,7,6,3,0,0,0,0},
{14,4,16,4,9,15,12,4,9,4,13,0,0,0,0},
{1,0,14,17,1,4,4,0,1,1,1,4,0,0,0},
{2,2,3,16,18,16,2,16,10,8,15,11,0,0,0},
{3,3,8,18,3,8,3,3,6,7,7,3,2,0,0},
{11,8,16,1,8,18,10,18,8,15,13,8,0,0},
{4,13,4,11,4,4,11,4,4,4,19,4,1,4,0},
{7,10,14,10,10,13,17,7,15,3,3,10,11,7,0},
{12,16,6,18,6,6,2,3,2,2,2,6,2,3,2}};*/

// int tri[1000][501];
int hold[2100],temp1,temp2,countnum,countshft,cases,lcount,ucount,grundy,
... count,i,j,temp,count2,count3,clear,upperbound1,upperbound2,count4,
... holdcount,count12,anotherarray[101];
ucount=0;
for (i = 0; i <= 100; i++) {anotherarray[i]=0;}
count12=0;
for (i = 0; i <= 999; i+=1) {for (j=0; j<=500; j+=1) tri[i][j]=999;}
tri[0][0]=0;
tri[1][0]=4;
tri[2][0]=2;
tri[2][1]=0;
grundytriangle<<"0\n0\n\n1\n4\n\n2\n2,0\n\n3\n";
anotherarray[0]=2;
anotherarray[2]=1;
anotherarray[4]=1;
//these loops form the actual algorithm that calculates S-G values for tri-paths.
for (countnum=3;countnum<=999;countnum+=1)
```

```
{for (countshft=0;countshft<=(countnum/2);countshft+=1)
{for (clear = 0; clear<=2099; clear+=1) {hold[clear]=999;}
{holdcount=0;
for (cases=1;cases<=16;cases+=1)
switch (cases)
{
//these are the possible moves in tri-paths.
case 1: if (countshft==0) hold[1]=path[countnum+1];
... else hold[1]=cat[countnum-1][countshft-1]; break;

case 2: hold[2]=path[countnum+1]; break;

case 3: if ((countnum/2)!=countshft) hold[3]=cat[countnum-1][countshft];
... else hold[3]=cat[countnum-1][countshft-1]; break;

case 4: temp1=path[countshft];temp2=path[countnum-countshft-1];
... hold[4]=sum(temp1,temp2); break;

case 5: if (countshft==0) hold[5]=path[countnum];
... else hold[5]=sum(path[countnum-countshft],path[countshft-1]); break;

case 6: hold[6]=path[countnum]; break;

case 7: if (countshft==0) hold[7]=sum(path[countnum-2],2);
... else hold[7]=sum(path[countnum-countshft-2],cat[countshft-1][0]); break;

case 8: hold[8]=sum(path[countnum-countshft-2],path[countshft+1]); break;

case 9: if (countshft==0) hold[9]=(999); else {if (countshft==1)
... hold[9]=cat[countnum-2][0];
else hold[9]=sum(cat[countnum-countshft-1][0],path[countshft-2]);} break;

case 10: if (countshft==0) hold[10]=(999); else {if (countshft==1) hold[10]=path[countnum];
else hold[10]=sum(path[countnum-countshft+1],path[countshft-2]);} break;

case 11: for (lcount=0;lcount<(countnum-countshft-2);lcount+=1)
{if (countshft>lcount) {hold[11+holdcount]=sum(path[countnum-countshft-lcount-3],
... tri[lcount+countshft][lcount]); holdcount+=1;}
else {hold[11+holdcount]=sum(tri[lcount+countshft][countshft],
... path[countnum-countshft-lcount-3]); holdcount+=1;}}
... if ((countshft)>((countnum-countshft)-2))
... {hold[11+holdcount]=tri[countnum-2][(countnum-countshft)-2]; holdcount+=1;}
else {hold[11+holdcount]=tri[countnum-2][countshft]; holdcount+=1;} break;

case 12: if (countshft<2) {hold[11+holdcount]=(999); holdcount+=1;}
... else {if (countshft<3) {hold[11+holdcount]=tri[countnum-2][0]; holdcount+=1;}
else {for (ucount=0;ucount<(countshft-2);ucount+=1)
... {hold[11+holdcount]=sum(tri[countnum-countshft+ucount][ucount],
```

```
... path[countshft-3-ucount]); holdcount+=1;}
{hold[11+holdcount]=tri[countnum-2][countshft-2]; holdcount+=1;}}} break;

case 13: for (lcount=0;lcount<(countnum-countshft-1);lcount+=1) {if (lcount>countshft)
... {hold[11+holdcount]=sum(path[countnum-countshft-lcount-2],
... tri[countshft+lcount][countshft]); holdcount+=1;}
else  {hold[11+holdcount]=sum(path[countnum-countshft-lcount-2],
... tri[countshft+lcount][lcount]); holdcount+=1;}}
if ((countshft!=(countnum/2))||((countnum%2)==1))
... {hold[11+holdcount]=tri[countnum-1][countshft]; holdcount+=1;}
... else hold[11+holdcount]=tri[countnum-1][countshft-1]; holdcount+=1; break;

case 14: if (countshft>0) {for (ucount=0;ucount<(countshft-1);ucount+=1)
... {hold[11+holdcount]=sum(path[countshft-ucount-2],
... tri[countnum-countshft+ucount][ucount]); holdcount+=1;}
... {hold[11+holdcount]=tri[countnum-1][countshft-1]; holdcount+=1;}} break;

case 15: hold[11+holdcount]=sum(path[countnum-countshft-2],path[countshft]);
... holdcount+=1; break;

case 16: if (countshft>0) {if (countshft>1)
... hold[11+holdcount]=sum(path[countnum-countshft],path[countshft-2]);
... else hold[11+holdcount]=path[countnum-1];} break;
}}
grundy=0;
count4=0;
upperbound1=(12+(countnum+lcount+ucount+1));
upperbound2 = (12+(countnum+lcount+ucount));
//this sorts the follower S-G's.
for (count2=0; count2<2100; count2+=1)
{for (count=0;count<2099;count+=1)
if (hold[count]>hold[count+1]) {temp=hold[count];
hold[count]=hold[count+1];
hold[count+1]=temp;}}
//this finds the mex.
while (hold[count4]==grundy)
{if (hold[count4]<hold[count4+1]) grundy+=1;
count4+=1;}
tri[countnum][countshft]=grundy;
grundytriangle<</*"The grundy value of ("<<countnum<<",
... "<<countshft<<") is "<<*/grundy<<", ";
anotherarray[grundy]+=1;
anotherarray[100]+=1;}
count12+=1;
//this prints the raw data for tri-paths
grundytriangle<<"\n\n\n"<<countnum+1<<"\n";
cout<<countnum<<"\n\n";}
grundytriangle<<"\n\n\n";
```

```
//this prints S-G distribution for tri-paths.
for (anothercountc=0; anothercountc<=100; anothercountc+=1)
,,, yetanotherarray[anothercountc]=0;
for (anothercounta=0; anothercounta <= 999; anothercounta+=1) {
for (anothercountc = 0; anothercountc <= 99; anothercountc++)
... justanothertemparray[anothercountc]=0;
for (anothercountb=0; anothercountb <= (anothercounta/2); anothercountb+=1) {
yetanotherarray[tri[anothercounta][anothercountb]]+=1;
... justanothertemparray[tri[anothercounta][anothercountb]]+=1;
... yetanotherarray[100]+=1;
}
grundytriangle<<anothercounta<<"\n";
for (finalcount=0; finalcount<=99; finalcount+=1)
... grundytriangle<<yetanotherarray[finalcount]<<"graphs had grundy value
...  "<<finalcount<<", this represents "<<(100*((double (yetanotherarray[finalcount]))
... /(double (yetanotherarray[100]))))<<"%
... of all graphs. +"<<justanothertemparray[finalcount]<<"\n\n";
grundytriangle<<yetanotherarray[100]<<" graphs were checked.";
}
grundytriangle.close();
}
//this function nim-sums.
int sum(int a,int b)
{
int binarray[10],bincount,tot;
for (bincount=0;bincount<=9;bincount+=1)
{if ((int (b)%2)==(int((a)%2))) binarray[bincount]=0;
else binarray[bincount]=1;
b-=(b%2);
b=(b/2);
a-=(a%2);
a=(a/2);
}
tot=0;
for (bincount=0;bincount<=9;bincount+=1)
{if (binarray[bincount]==1) tot+=pow(2,bincount);}
return (tot);
}
```

14.4. **Losing Graphs Code.**

```
import networkx
from sage.graphs.graph_isom import perm_group_elt, orbit_partition
from sage.groups.perm_gps.partn_ref.refinement_graphs import search_tree
__all__ = ['read_graph6', 'parse_graph6', 'read_graph6_list',
          'read_sparse6', 'parse_sparse6', 'read_sparse6_list']
from networkx.exception import NetworkXException, NetworkXError
from networkx.utils import _get_fh
def read_graph6_list(path):
    """Read simple undirected graphs in graph6 format from path.
```

```
    Returns a list of Graphs, one for each line in file."""
    fh=_get_fh(path,mode='r')
    glist=[]
    for line in fh:
        line = line.strip()
        if not len(line): continue
        glist.append(parse_graph6(line))
    return glist
def graph6data(str):
    """Convert graph6 character sequence to 6-bit integers."""
    v = [ord(c)-63 for c in str]
    if min(v) < 0 or max(v) > 63:
        return None
    return v
def graph6n(data):
    """Read initial one or four-unit value from graph6 sequence.
    Return value, rest of seq."""
    if data[0] <= 62:
        return data[0], data[1:]
    return (data[1]<<12) + (data[2]<<6) + data[3], data[4:]
def parse_graph6(str):
    """Read undirected graph in graph6 format."""
    if str.startswith('>>graph6<<'):
        str = str[10:]
    data = graph6data(str)
    n, data = graph6n(data)
    nd = (n*(n-1)//2 + 5) // 6
    if len(data) != nd:
        raise NetworkXError, 'Expected %d bits but got %d in graph6' %
        ... (n*(n-1)//2, len(data)*6)
    def bits():
        """Return sequence of individual bits from 6-bit-per-value
        list of data values."""
        for d in data:
            for i in [5,4,3,2,1,0]:
                yield (d>>i)&1
    G=networkx.Graph()
    G.add_nodes_from(range(n))
    for (i,j),b in zip([(i,j) for j in range(1,n) for i in range(j)], bits()):
        if b: G.add_edge(i,j)
    return G
# All of the above functions are used to convert textfiles of graphs saved in
#graph6 format into lists of graphs as Sage object.

def getGraphs(path):
ret = []
temp = read_graph6_list(path)
for t in temp:
```

```
ret.append(Graph(t))
return ret
def getAntiMoves(lis, k):
remain=0
temp = []
shorter = []
ret = []
if k == 0:
for i in range(len(lis)):
temp.append(0)
return [temp]
current = 0
if len(lis) == 0:
return [[]]
if len(lis) == 1:
return[[k]]
if lis[0]<=k:
mini=lis[0]
else:
mini=k
shorter = lis[:]
del shorter[0]
for d in shorter:
remain+=d
for j in range(mini+1):
if remain>=k-mini+j:
temp = getAntiMoves(shorter,k-mini+j)
# This function works in a recursive manner, which is why it calls itself.

for t in temp:
ret.append([mini-j] + t)
return ret
def getVerts(o2, lis):
ret = []
for l in lis:
temp = []
for ll in range(len(l)):
if l[ll]==1:
temp.append(o2[ll])
ret.append(temp)
return ret
def findLosses(vertices, edges, los):
lose=[]
# 'lose' will be the list of losing graphs

group=[]
path2 = '/Users/brice/Desktop/sage/textfiles/losing/' + str(los)
lose = getGraphs(path2)
```

```
for r in range(los+1,edges+1):
path = '/Users/brice/Desktop/sage/textfiles/ninegraphs/errors/' + str(r)
Ltemp = getGraphs(path)
# 'Ltemp' is the list of graphs with 'r' edges and 'vertices' vertices

L = []
for i in range(vertices):
L.append([])
for j in range(vertices):
L[i].append([])
for k in range(vertices):
L[i][j].append([])
for z in range(vertices):
L[i][j][k].append([])
for y in range(vertices):
L[i][j][k][z].append([])
for l in Ltemp:
ll = l.canonical_label()
L[ll.degree(0)][ll.degree(1)][ll.degree(2)][ll.degree(3)][ll.degree(4)].append(ll)
# Here we store the graphs in 'Ltemp' in a more organized
#fashion in the list (of lists) we call 'L'

for t in range(len(lose)):
if (lose[t].num_edges()+vertices-1)>=r:
# The above if statement ensures that the losing graph has
#enough edges to have a parent with 'r' edges

checked = []
group = lose[t].automorphism_group()
orb = group.orbits()
group=[]
verts = []
for i in range(vertices):
verts.append(i)
for g in orb:
for gg in g:
if gg in verts:
verts.remove(gg)
if len(verts)>0:
for l in verts:
orb.append([l])
for g in orb:
#We only want to make antimoves to one vertex from each orbit;
#this cuts down on redundancy.

a = g[0]%vertices
# 'a' is the vertex to which we will be adding edges (i.e., making an anti-move)
```

```
tempo = lose[t].degree(a)
if (lose[t].num_edges()+vertices-(tempo+1))>=r:
# The above if statement ensures that the degree of the vertex
#is small enough to allow us to make an anti-move that gets a
#parent graph with 'r' edges

orb2=[]
for ggg in range(vertices):
orb2.append(ggg)
orb2.remove(a)
for j in lose[t].neighbor_iterator(a):
orb2.remove(j)
moves1=[]
for o in orb2:
moves1.append(1)
moves = getVerts(orb2, getAntiMoves(moves1, r-lose[t].num_edges()))
for o in range(len(moves)):
for oo in moves[o]:
lose[t].add_edge(a,oo)
# Here we make the anti-move by adding edges

tempgraph = lose[t].canonical_label()
# Above we canonically label the parent graph, which we call 'tempgraph'


deg0 = tempgraph.degree(0)
deg1 = tempgraph.degree(1)
deg2 = tempgraph.degree(2)
deg3 = tempgraph.degree(3)
deg4 = tempgraph.degree(4)
if tempgraph not in checked:
checked.append(tempgraph.copy())
if tempgraph in L[deg0][deg1][deg2][deg3][deg4]:
L[deg0][deg1][deg2][deg3][deg4].remove(tempgraph)
# Above we remove 'tempgraph' from 'L', since we know 'tempgraph' is winning

for oo in moves[o]:
lose[t].delete_edge(a,oo)
# Here we undo the anti-move by removing the edges we added


for i in range(vertices):
for j in range(vertices):
for k in range(vertices):
for z in range(vertices):
for y in range(vertices):
lose = lose + L[i][j][k][z][y]
# Above we add any graphs left over in 'L' to 'lose'
```

```
makenew = str(r)
filename = 'losing' + makenew + '.txt'
f = open(filename, 'w')
for l in lose:
f.write(l.graph6_string())
f.write('\n')
return lose
```

14.5. **Graph S-G Number Code.**

```
def getMoves(lis, k):
remain=0
temp = []
shorter = []
ret = []
if k == 0:
for i in range(len(lis)):
temp.append(0)
return [temp]
current = 0
if len(lis) == 0:
return [[]]
if len(lis) == 1:
return[[k]]
if lis[0]<=k:
mini=lis[0]
else:
mini=k
shorter = lis[:]
del shorter[0]
for d in shorter:
remain+=d
for j in range(mini+1):
if remain>=k-mini+j:
temp = getMoves(shorter,k-mini+j)
for t in temp:
ret.append([mini-j] + t)
return ret
def getVerts(o2, lis):
ret = []
for l in lis:
temp = []
for ll in range(len(l)):
if l[ll]==1:
temp.append(o2[ll])
ret.append(temp)
return ret
def makeNeat(graphs, grunds, vertices):
```

```
ret = []
retg = []
for i in range(vertices):
ret.append([])
retg.append([])
for j in range(vertices):
retg[i].append([])
ret[i].append([])
for k in range(vertices):
retg[i][j].append([])
ret[i][j].append([])
for t in range(len(graphs)):
tt = graphs[t].canonical_label()
deg0 = tt.degree(0)
deg1 = tt.degree(1)
deg2 = tt.degree(2)
ret[deg0][deg1][deg2].append(tt)
retg[deg0][deg1][deg2].append(grunds[t])

return ret, retg
def findMex(gruns):
sett = set(gruns)
for j in range(len(sett)+1):
if j not in sett:
return j
def getSG(vertices):
u = (vertices*(vertices-1)/2) + 1
$ u equals 'vertices' choose 2

unord = []
#'unord' will be the unordered list of graphs,
#separated into sub-lists by the number of edges

unordG = []
# 'unordG' will be the list of S-G numbers of the graphs
#in unord, stored in the same order

master = []
# 'master' will be the list of graphs once they
#have been organized by the makeNeat function

masterG = []
# 'masterG' will be the list of S-G numbers of
#the graphs in 'master', stored in the same order

empty_graph = Graph()
for i in range(vertices):
empty_graph.add_vertex(i)
```

```
gees = [empty_graph]
grees = [0]
unord.append(gees)
unordG.append(grees)
gs, grs = makeNeat(gees, grees, vertices)
master.append(gs)
masterG.append(grs)
for e in range(1, u):
G= GraphQuery(display_cols=['graph6'],num_vertices=vertices, num_edges=e)
C = G.get_graphs_list()
# 'C' is the list of graphs with 'e' edges

unord.append(C)
grundies = []
for g in C:
grunds = []
for i in range(vertices):
verts = []
movelen = []
move1 = []
moves = []
for j in g.neighbor_iterator(i):
verts.append(j)
for kk in range(len(verts)):
movelen.append(1)
for k in range(1, len(verts)+1):
tried = []
move1 = getMoves(movelen, k)
moves = getVerts(verts, move1)
LL = master[e-k]
GG = masterG[e-k]
for m in moves:
for mm in m:
g.delete_edge(i, mm)
# Here we make a move on 'g' by deleting edges

tempgraph = g.canonical_label()
deg0 = tempgraph.degree(0)
deg1 = tempgraph.degree(1)
deg2 = tempgraph.degree(2)
if tempgraph not in tried:
tried.append(tempgraph.copy())
index = LL[deg0][deg1][deg2].index(tempgraph)
grunds.append(GG[deg0][deg1][deg2][index])
for mm in m:
g.add_edge(i, mm)
# Here we undo the move we had made on 'g', by adding back the edges
```

```
grundies.append(findMex(grunds))
# Here we find the mex of the S-G numbers of all child graphs of 'g'

unordG.append(grundies)
newG, newGr = makeNeat(C, grundies, vertices)
master.append(newG)
masterG.append(newGr)
return unord, unordG
```

14.6. **Graph S-G Heuristic Code.**

```
def adjust(graphs, gruns):
ret = []
t = graphs[0][0].num_verts()
m = (t*(t-1)/2) + 1
for i in range(m):
temp=[]
for j in range(i+1):
temp.append(0)
for dd in range(len(gruns[i])):
temp[gruns[i][dd]] = temp[gruns[i][dd]] + factorial(t)/
... graphs[i][dd].automorphism_group(return_group=False, order=True)
ret.append(temp)
return ret

def percentage(distr):
ret = []
for i in range(len(distr)):
temp = []
total = 0
for h in distr[i]:
total+=h
for j in range(i+1):
tot = distr[i][j]
a = tot/total*100.
if a == 0:
temp.append(0)
elif a == 100:
temp.append(100)
else:
temp.append(round(a,1))
ret.append(temp)
return ret


def typicalDegree(m,grafs):
degre = []
vertices = grafs[0][0].num_verts()
M = (vertices*(vertices-1)/2)
x = factorial(M)/(factorial(m)*factorial(M-m))
```

```
tot = 2**x
for i in range(vertices):
degre.append(0)
for g in grafs[m]:
se = g.degree()
se.sort()
for i in range(vertices):
degre[i] = degre[i] + se[i]*(factorial(vertices)/
... g.automorphism_group(return_group=False, order=True))
for i in range(vertices):
degre[i] = degre[i]/(1.*x)
degre[i] = round(degre[i],0)

return degre

def predictG(pers,deg,m):
prob = []
for i in range(m+1):
prob.append(1)
for i in range(m+1):
for k in range(i):
prob[i] *= (1-probCant(pers,deg,m,k))
prob[i] *= probCant(pers,deg,m,i)
prob[i] *= 100
prob[i] = round(prob[i],1)
if prob[i] == 0:
prob[i] = 0
return prob


def n_k(n,k):
#This function returns n choose k

return factorial(n)/(factorial(k)*factorial(n-k))

def probCant(pers, deg, m, s):
prob = 1
if s <= m-1:
prob *= ((1-(pers[m-1][s]/100))^m)
for d in deg:
for k in range(2,d+1):
if s<=m-k:
num = n_k(d,k)
prob *= ((1-(pers[m-k][s]/100))^num)
return prob

def generateH(n):
top = n*(n-1)/2 + 1
```

```
graphs, grundy = getSG(n)
pats = adjust(graphs,grundy)
perc = percentage(pats)
ret = []
for j in range(4):
ret.append(perc[j])
for j in range(4, top):
degr = typicalDegree(j,graphs)
ret.append(predictG(perc, degr, j))
return ret, perc
```

15. Appendix C: $C_{n,k}$ periods

The following are the periods for one-legged caterpillars $C_{n,k}$. For each period, a leg has been fixed on the index denoted and the length of the caterpillar is increased. Exceptions before the period begins have been listed for caterpillars with periods of twelve up to index 25. "*""s denote lengths on on which the index is out of range.

Table for $C_{n,1}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	$*$	2	3	4	5	6	2	1	0	8	6	0
12	1	2	3	8	5	12	7	1	0	8	9	14
24	1	2	3	11	4	7	12	14	0	16	2	4
36	12	2	3	10	4	7	15	1	16	9	18	16
48	12	2	3	10	16	7	12	1	16	18	11	16
60	12	2	22	11	16	7	12	1	20	24	16	26
72	12	13	22	11	16	24	15	14	16	22	19	16
84	12	13	19	11	16	24	15	14	16	25	11	16
96	12	13	22	11	16	7	15	1	20	25	19	11
108	12	13	22	11	32	19	22	14	20	22	19	11
120	12	13	22	11	25	19	22	14	16	22	19	11
132	21	13	22	11	25	19	22	14	21	25	19	11
144	21	13	22	11	25	19	22	14	20	22	19	11
156	21	13	22	11	25	19	22	14	21	22	19	11

Table for $C_{n,2}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	$*$	$*$	3	4	5	6	7	8	0	10	6	8
12	1	6	3	8	5	7	3	8	0	10	13	4
24	0	2	3	8	13	10	16	1	0	10	17	8
36	18	10	3	8	18	6	13	9	18	20	6	5
48	18	10	3	20	5	6	16	13	18	10	17	4
60	18	10	3	8	19	6	16	13	22	10	17	4
72	18	14	3	8	18	6	16	13	19	10	17	8
84	4	14	3	8	18	6	16	13	18	10	17	4
96	18	14	3	8	18	7	16	13	18	10	17	4
108	18	14	3	8	18	7	16	13	18	10	17	4
120	18	14	3	8	18	27	16	13	18	10	17	4
132	18	14	3	8	18	7	16	13	18	10	17	4

Table for $C_{n,3}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	1	5	6	2	8	5	2	6	12
12	1	2	8	9	11	14	6	0	5	14	12	16
24	14	3	8	16	4	14	6	8	11	16	7	13
36	5	3	2	9	11	14	16	8	11	16	12	5
48	14	15	2	16	11	22	7	8	1	11	8	13
60	14	11	2	16	11	21	7	8	10	11	12	16
72	14	15	2	8	11	21	24	12	16	11	21	5
84	14	15	2	8	11	21	7	8	16	11	25	5
96	14	21	2	8	11	16	21	8	16	11	25	5
108	14	15	2	8	11	21	16	12	19	11	25	5
120	14	15	2	8	11	21	28	12	16	11	19	5
132	14	15	2	8	11	21	28	8	16	11	25	5
144	14	15	2	8	11	21	28	12	16	11	25	5
156	14	15	2	8	11	21	16	12	19	11	25	5

Table for $C_{n,4}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	$*$	$*$	$*$	$*$	4	6	7	8	3	10	8	5
12	4	11	3	4	8	10	13	8	16	13	8	4
24	17	16	18	1	11	10	18	8	1	2	19	13
36	4	19	13	1	11	4	22	8	17	21	22	13
48	4	25	13	26	11	14	7	4	13	24	8	13
60	4	24	13	8	11	14	28	4	13	24	19	13
72	4	19	13	1	11	14	7	4	1	24	22	13
84	4	7	13	8	11	14	7	4	14	24	22	13
96	4	7	13	8	11	14	7	4	13	11	22	13
108	4	7	13	8	11	14	7	4	14	24	22	13
120	4	7	13	8	11	14	7	4	14	11	22	13

Table for $C_{n,5}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	$*$	$*$	$*$	$*$	$*$	3	2	8	5	10	3	6
12	1	14	6	8	0	3	6	8	15	10	3	9
24	1	18	3	8	4	10	6	16	0	6	7	8
36	5	10	3	17	0	15	17	12	5	10	3	8
48	5	15	6	17	0	15	6	12	5	6	3	12
60	5	10	16	9	4	10	25	12	5	6	3	8
72	5	10	3	5	4	19	17	12	5	6	3	8
84	5	10	6	17	26	22	25	12	5	6	3	8
96	5	10	16	5	4	22	17	12	26	6	3	8
108	5	10	24	5	23	22	25	12	19	6	3	8
120	5	10	24	5	23	22	25	12	5	6	3	8
132	5	10	24	5	26	22	25	12	19	6	3	8

Table for $C_{n,6}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	$*$	$*$	$*$	$*$	$*$	$*$	2	1	0	2	8	6
12	4	2	8	1	4	3	6	8	10	13	7	15
24	4	16	8	18	4	13	6	18	11	10	7	4
36	5	16	3	21	4	7	6	18	24	2	13	18
48	5	16	18	1	4	21	6	24	8	2	23	18
60	5	24	8	1	4	21	6	18	11	8	23	22
72	5	17	8	1	4	21	6	18	11	2	7	22
84	5	16	8	1	4	21	6	24	25	2	23	22
96	5	29	8	1	4	21	6	18	11	2	23	22
108	5	17	8	1	4	21	6	18	11	2	23	22
120	5	30	8	1	4	30	6	18	11	2	23	22
132	5	32	8	1	4	21	6	18	11	2	23	22
144	5	30	8	1	4	30	6	18	11	2	23	29

Table for $C_{n,8}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	4	8	6	12
12	4	14	8	10	4	14	2	13	0	15	6	4
24	0	16	3	18	4	7	3	1	0	10	7	13
36	1	2	3	8	23	24	18	19	16	11	7	21
48	15	16	3	8	5	21	2	13	24	27	6	22
60	1	14	3	22	25	7	2	13	16	32	7	22
72	1	16	19	22	20	21	2	13	1	17	23	22
84	1	15	3	8	25	21	2	13	1	32	27	22
96	1	14	27	32	24	27	2	13	1	32	7	4
108	1	14	18	22	24	32	2	13	1	32	23	4
120	1	14	27	32	24	21	2	13	1	32	27	22
132	1	14	27	35	25	21	2	13	1	32	33	4
144	1	14	18	35	24	21	2	13	1	32	27	4
156	1	14	18	35	24	21	2	13	1	32	27	4
168	1	14	27	32	24	21	2	13	1	32	27	4
180	1	14	18	35	24	21	2	13	1	32	27	4
192	1	14	18	35	24	21	2	13	1	32	34	4
204	1	14	18	42	24	21	2	13	1	35	27	4
216	1	14	18	38	24	21	2	13	1	32	27	4
228	1	14	18	35	24	21	2	13	1	32	27	4
240	1	14	18	42	24	21	2	13	1	44	27	4
252	1	14	18	38	24	21	2	13	1	37	43	4
264	1	14	18	38	24	21	2	13	1	32	27	4
276	1	14	18	37	24	21	2	13	1	44	27	4
288	1	14	18	37	24	21	2	13	1	37	27	4
300	1	14	18	38	24	21	2	13	1	37	27	4
312	1	14	18	37	24	21	2	13	1	37	42	4
324	1	14	18	37	24	21	2	13	1	37	27	4
336	1	14	18	37	24	21	2	13	1	37	27	4
348	1	14	18	37	24	21	2	13	1	37	27	4
360	1	14	18	37	24	21	2	13	1	37	27	4
372	1	14	18	37	24	21	2	13	1	37	42	4
384	1	14	18	37	24	21	2	13	1	37	27	4
396	1	14	18	37	24	21	2	13	1	37	27	4
408	1	14	18	37	24	21	2	13	1	37	27	4
420	1	14	18	37	24	21	2	13	1	37	27	4
432	1	14	18	37	24	21	2	13	1	37	42	4
444	1	14	18	37	24	21	2	13	1	37	27	4

Table for $C_{n,9}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	2	6	8
12	1	11	6	1	5	14	2	1	4	10	7	9
24	1	10	6	8	4	15	2	8	16	11	7	12
36	1	11	16	8	1	11	2	13	1	11	2	12
48	1	10	21	8	4	14	19	1	4	11	2	12
60	1	2	7	8	1	11	2	1	4	11	2	12
72	1	11	7	8	21	11	6	1	19	11	2	12
84	1	11	16	8	21	11	25	1	4	11	2	12
96	1	11	7	8	21	11	16	13	1	11	2	12
108	1	11	7	8	21	11	19	13	1	11	2	12
120	1	11	7	8	16	11	25	13	1	11	2	12
132	1	11	7	8	16	11	19	13	1	11	2	12

Table for $C_{n,10}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	6	0
12	1	2	3	8	4	3	2	1	4	8	6	4
24	1	8	7	15	4	3	2	1	4	6	7	4
36	1	16	3	5	4	7	2	14	15	16	7	4
48	1	2	3	1	4	7	2	1	21	2	7	4
60	1	2	8	1	4	7	2	1	4	16	7	4
72	1	2	8	1	4	7	2	1	21	16	7	4
84	1	2	24	1	4	7	2	1	21	2	7	4
96	1	2	8	1	4	7	2	1	11	2	7	4
108	1	2	8	1	4	7	2	1	21	2	7	4
120	1	2	22	1	4	7	2	1	21	2	7	4
132	1	2	22	1	4	7	2	1	21	2	7	4
144	1	2	8	1	4	7	2	1	21	2	7	4
156	1	2	22	1	4	7	2	1	21	2	7	4

Table for $C_{n,11}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	4
12	1	6	8	4	0	3	2	13	4	8	3	13
24	1	15	7	16	19	3	2	10	0	21	22	
36	0	1	16	3	13	19	3	2	24	0	2	21
48	0	1	15	16	19	24	3	2	21	25	2	21
60	0	1	16	19	1	16	3	13	28	21	24	27
72	0	1	16	19	1	24	3	13	32	25	2	21
84	16	1	30	19	1	24	3	2	30	21	2	32
96	21	1	32	33	1	16	3	13	30	21	2	27
108	21	1	30	19	1	35	3	13	21	38	2	21
120	37	1	30	32	1	24	3	13	21	22	2	27
132	21	1	30	32	1	35	3	13	21	34	2	16
144	21	1	30	32	1	35	3	13	21	22	2	16
156	21	1	32	33	1	35	3	13	21	22	2	16
168	21	1	30	32	1	19	3	13	21	22	2	16
180	21	1	32	37	1	19	3	13	21	22	2	16
192	21	1	37	21	1	19	3	13	21	22	2	16
204	21	1	37	21	1	19	3	13	21	22	2	16
216	21	1	32	21	1	19	3	13	21	22	2	16
228	21	1	37	21	1	19	3	13	21	22	2	16
240	21	1	32	21	1	19	3	13	21	22	2	16

Table for $C_{n,13}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	2	3	8	11	10	6	8	0	15	7	4
24	1	11	2	8	1	11	2	8	0	18	7	9
36	1	11	2	1	4	11	2	16	1	10	6	4
48	1	14	2	1	19	10	2	12	1	19	26	4
60	1	14	2	20	24	7	2	12	1	14	7	4
72	1	11	3	1	24	7	2	12	1	18	7	8
84	1	19	2	24	27	7	2	16	1	22	7	8
96	1	19	13	8	24	7	2	21	1	23	7	8
108	1	19	29	21	4	7	2	21	1	22	7	8
120	1	19	13	24	4	7	2	16	1	22	7	8
132	1	19	13	21	4	7	2	33	1	22	7	8
144	1	19	13	21	4	7	2	21	1	22	7	8
156	1	19	13	21	4	7	2	31	1	22	7	8
168	1	19	13	21	4	7	2	32	1	22	7	8

Table for $C_{n,15}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	1	5	7	6	8	15	13	8	4
24	1	8	7	13	1	7	2	1	4	14	7	4
36	1	11	8	13	4	21	2	8	18	19	7	4
48	1	8	21	1	4	7	6	8	1	2	7	4
60	5	8	2	1	4	7	8	16	1	14	7	4
72	1	16	2	13	4	7	22	8	1	2	7	4
84	1	24	2	1	4	7	8	19	1	2	7	4
96	1	24	2	1	4	7	8	24	1	2	7	4
108	1	16	2	1	4	7	2	8	1	2	7	4

Table for $C_{n,17}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	3	7	8	5	13	3	15
24	4	16	6	15	19	10	2	16	4	10	3	8
36	1	19	3	15	1	10	2	16	0	15	2	22
48	1	10	3	22	0	21	6	9	18	19	3	22
60	1	10	6	24	4	21	2	19	21	24	3	22
72	1	30	21	24	0	21	6	16	5	24	3	22
84	5	29	32	27	4	22	6	4	32	15	3	21
96	1	37	6	16	4	29	30	24	35	32	3	21
108	10	37	21	24	4	22	6	4	22	24	3	29
120	10	22	21	24	4	29	37	4	19	24	3	29
132	10	22	32	24	4	29	37	4	35	24	3	21
144	10	22	37	24	4	29	32	4	22	16	3	29
156	10	32	21	24	4	29	32	4	22	24	3	37
168	10	22	32	24	4	29	32	4	41	24	3	29
180	10	22	29	24	4	29	32	4	37	16	3	29
192	10	22	32	24	4	29	32	4	22	16	3	29
204	10	22	32	24	4	29	32	4	38	24	3	29
216	10	22	29	24	4	29	32	4	38	16	3	29
228	10	22	29	24	4	29	32	4	41	16	3	29
240	10	22	32	24	4	29	32	4	37	16	3	29
252	10	22	29	24	4	29	32	4	37	16	3	29
264	10	22	29	24	4	29	32	4	38	16	3	29
276	10	22	29	24	4	29	32	4	37	16	3	29

Table for $C_{n,18}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	*	1	0	14	8	9	4
24	2	3	8	4	3	6	8	0	14	6	8	4
36	11	3	1	4	14	6	8	17	10	7	4	5
48	11	18	8	4	7	6	12	17	10	18	13	5
60	15	3	1	4	18	6	12	17	2	7	8	5
72	15	17	8	4	21	6	12	18	2	7	18	5
84	15	13	8	4	24	6	13	17	28	7	24	5
96	14	13	8	4	17	6	13	17	2	7	23	5
108	29	13	8	4	17	6	13	27	2	7	23	27
120	29	13	8	4	20	24	13	18	2	7	8	24
132	32	13	8	4	32	6	13	18	2	7	8	20
144	24	13	8	4	11	20	13	17	2	7	8	32
156	29	13	8	4	11	20	13	17	2	7	8	20
168	29	13	8	4	11	23	13	18	2	7	8	20
180	29	13	8	4	11	20	13	18	2	7	8	20
192	29	13	8	4	11	20	13	17	2	7	8	20
204	29	13	8	4	11	20	13	17	2	7	8	20
216	29	13	8	4	11	20	13	18	2	7	8	20
228	29	13	8	4	11	20	13	17	2	7	8	20

Table for $C_{n,19}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	*	*	1	0	10	12	4
24	1	16	3	18	4	3	2	1	0	10	7	9
36	1	11	3	21	4	11	2	8	0	14	7	4
48	1	11	3	8	4	7	6	12	0	16	20	4
60	1	15	3	9	25	7	2	12	0	10	19	4
72	1	15	3	8	4	14	2	12	4	21	26	13
84	1	15	3	8	4	28	6	25	0	14	25	12
96	1	25	3	8	4	28	2	25	16	32	26	13
108	1	28	3	8	4	14	2	19	32	25	7	13
120	1	15	3	8	31	28	2	19	16	32	7	22
132	1	16	3	8	4	31	2	19	16	32	7	22
144	1	33	3	8	31	28	2	19	16	32	7	13
156	1	25	3	8	31	28	2	19	16	32	7	22
168	1	25	3	8	31	28	2	19	16	32	7	22
180	1	32	3	8	31	28	2	19	16	32	7	22
192	1	25	3	8	31	28	2	19	16	32	7	22
204	1	25	3	8	31	28	2	19	16	37	7	22

Table for $C_{n,20}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	*	*	*	4	8	13	16
24	17	18	8	10	4	15	2	10	0	18	7	4
36	13	19	3	21	4	7	3	1	0	10	7	4
48	1	2	3	21	4	13	6	16	11	18	26	21
60	5	28	27	24	25	13	6	16	28	8	7	21
72	1	23	3	17	4	7	13	32	0	8	26	31
84	1	32	31	36	4	7	2	31	1	25	23	21
96	1	28	31	21	23	22	6	31	1	39	36	15
108	1	28	33	32	38	7	6	28	1	8	20	4
120	1	40	33	32	39	7	6	31	1	35	26	4
132	1	28	33	40	43	7	6	35	1	23	26	17
144	1	28	41	40	38	7	6	28	1	35	26	4
156	1	28	37	40	43	7	6	28	1	26	27	4
168	1	43	33	40	38	7	6	28	1	43	26	4
180	1	31	33	40	38	7	6	28	1	26	27	4
192	1	48	33	40	38	7	6	28	1	26	27	4
204	1	46	33	40	38	7	6	28	1	26	40	4
216	1	45	33	48	38	7	6	28	1	26	27	4
228	1	46	33	40	38	7	6	28	1	26	27	4

Table for $C_{n,21}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	*	*	*	*	6	9	4
24	14	16	3	18	11	7	2	1	0	14	7	4
36	1	2	3	1	4	7	2	1	11	2	7	4
48	1	2	8	9	4	7	2	0	4	2	7	4
60	18	20	8	1	4	7	18	25	21	14	7	4
72	14	15	2	13	4	7	2	24	21	2	7	4
84	1	3	2	24	4	7	13	18	24	2	7	4
96	14	17	2	9	4	16	13	27	32	2	7	4
108	14	2	27	32	4	7	13	32	24	2	7	4
120	14	2	27	32	4	35	13	18	32	2	7	4
132	14	2	13	22	4	16	13	18	32	2	7	4
144	14	2	13	37	4	27	13	18	27	2	7	4
156	14	2	24	29	4	32	13	18	27	2	7	4
168	14	2	13	32	4	27	13	18	27	2	7	4
180	14	2	13	37	4	27	13	18	27	2	7	4
192	14	2	13	29	4	27	13	18	27	2	7	4

Table for $C_{n,22}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	*	*	*	*	*	7	14
24	0	3	18	8	4	17	3	8	4	21	6	9
36	0	11	6	15	4	11	3	1	4	6	7	18
48	1	16	3	5	4	15	2	19	1	10	7	23
60	1	16	3	18	4	20	26	8	1	20	7	24
72	0	27	2	19	4	20	3	8	32	16	6	18
84	32	20	28	18	4	20	32	1	36	10	7	12
96	1	3	27	9	4	31	35	8	1	10	29	22
108	32	20	38	23	4	21	34	8	1	10	35	32
120	40	3	35	9	5	20	42	8	32	10	29	22
132	32	3	35	9	4	17	38	1	32	10	35	41
144	32	3	35	9	5	21	35	8	32	10	29	22
156	41	3	35	9	5	21	47	8	32	10	35	22
168	32	3	35	9	5	17	35	8	32	10	35	22
180	32	3	35	9	5	21	35	8	32	10	35	22

Table for $C_{n,23}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	*	*	*	*	*	*	4
24	1	2	8	1	4	13	2	1	16	6	22	4
36	1	2	8	4	1	14	2	1	11	6	4	13
48	1	2	18	1	4	13	2	1	17	2	21	4
60	1	2	13	1	4	20	2	1	10	8	7	4
72	1	2	8	13	4	7	2	1	16	6	7	4
84	1	2	8	1	4	7	2	1	25	2	7	4
96	1	27	8	1	4	7	21	1	16	2	7	4
108	1	2	8	1	4	7	13	1	16	8	7	4
120	1	30	8	1	4	7	2	1	16	2	7	4
132	1	30	8	1	4	7	13	1	8	2	7	4
144	1	29	8	1	4	7	13	1	16	2	7	4
156	1	29	8	1	4	7	13	1	24	2	7	4
168	1	21	8	1	4	7	13	1	24	2	7	4
180	1	29	8	1	4	7	13	1	24	2	7	4
192	1	21	8	1	4	7	13	1	24	2	7	4

Table for $C_{n,25}$

	0	1	2	3	4	5	6	7	8	9	10	11
0	*	*	*	*	*	*	*	*	*	*	*	*
12	*	*	*	*	*	*	*	*	*	*	*	*
24	*	2	3	8	4	10	6	18	0	10	7	4
36	1	11	2	8	4	15	2	8	0	10	7	18
48	1	11	2	1	4	11	2	24	1	10	6	4
60	5	14	7	1	4	10	2	9	1	10	23	4
72	1	14	17	9	4	7	2	12	1	10	7	4
84	1	11	3	1	4	7	2	12	1	10	21	4
96	5	26	20	1	4	7	2	12	4	10	28	8
108	1	18	24	1	4	7	2	12	1	10	29	28
120	5	15	25	1	4	7	2	12	26	10	28	8
132	5	16	25	1	4	7	2	19	17	10	28	8
144	5	26	18	1	4	11	2	18	26	10	28	8
156	5	32	18	1	4	7	2	12	17	10	28	8
168	5	32	24	1	4	11	2	37	17	10	28	8
180	5	32	18	1	4	11	2	40	17	10	28	8
192	5	32	18	1	4	11	2	39	17	10	28	8
204	5	36	18	1	4	11	2	39	17	10	28	8
216	5	32	18	1	4	11	2	35	17	10	28	8
228	5	32	18	1	4	11	2	35	17	10	28	8
240	5	32	18	1	4	11	2	39	17	10	28	8
252	5	32	18	1	4	11	2	35	17	10	28	8

The following sets of caterpillars have the given periods of twelve starting at the listed index: $C_{n,26}$ (Period beginning at Index 240):

1, 14, 2, 1, 16, 7, 2, 13, 16, 2, 7, 4

$C_{n,27}$ (Period beginning at Index 252):
30, 7, 25, 1, 26, 32, 28, 4, 26, 11, 24, 8

$C_{n,28}$ (Period beginning at Index 264):
42, 15, 16, 1, 4, 14, 2, 29, 32, 2, 7, 4

$C_{n,29}$ (Period beginning at Index 252):
1, 35, 26, 1, 19, 28, 2, 37, 14, 2, 7, 21

$C_{n,30}$ (Period beginning at Index 252):
13, 38, 47, 1, 4, 11, 14, 22, 21, 2, 7, 8

$C_{n,31}$ (Period beginning at Index 156):
1, 14, 18, 1, 34, 11, 2, 12, 1, 2, 19, 8

$C_{n,37}$ (Period beginning at Index 168):
1, 24, 2, 1, 4, 11, 2, 12, 1, 2, 7, 8

$C_{n,45}$ (Period beginning at Index 276):
25, 7, 32, 25, 11, 32, 13, 4, 32, 25, 22, 32

$C_{n,46}$ (Period beginning at Index 276):
14, 3, 2, 1, 4, 37, 13, 4, 14, 2, 7, 4

$C_{n,47}$ (Period beginning at Index 204):
16, 7, 2, 1, 22, 11, 19, 4, 1, 2, 7, 8

The following sets of caterpillars have periods of 60 starting at the given indices:

$C_{n,7}$ (Period beginning at Index 360):
5, 12, 18, 27, 17, 24, 6, 15, 10, 37, 41, 29
5, 12, 18, 27, 17, 29, 6, 15, 10, 27, 41, 30
5, 12, 18, 27, 17, 29, 6, 15, 10, 37, 41, 30
5, 12, 18, 27, 17, 29, 6, 15, 10, 37, 41, 29
5, 12, 18, 27, 17, 24, 6, 15, 10, 37, 41, 29

$C_{n,12}$ (Period beginning at Index 300):
22, 14, 28, 1, 4, 11, 25, 12, 28, 21, 7, 8
22, 14, 28, 1, 4, 11, 25, 12, 28, 21, 7, 8
22, 14, 25, 1, 4, 11, 25, 12, 28, 21, 7, 8
22, 14, 28, 1, 4, 11, 25, 12, 28, 21, 7, 8
22, 14, 28, 1, 4, 11, 25, 12, 31, 21, 7, 8

$C_{n,14}$ (Period beginning at Index 240):
33, 14, 26, 8, 25, 22, 12, 13, 21, 10, 26, 32
33, 14, 26, 8, 25, 22, 12, 13, 21, 10, 26, 32
28, 14, 26, 8, 28, 22, 12, 13, 21, 10, 26, 32
28, 14, 26, 8, 25, 22, 12, 13, 21, 10, 26, 32
28, 14, 26, 8, 25, 22, 12, 13, 21, 10, 26, 32

$C_{n,16}$ (Period beginning at Index 240):
31, 3, 28, 8, 32, 21, 28, 16, 21, 10, 25, 21
32, 3, 28, 8, 31, 21, 28, 16, 21, 10, 25, 21

32, 3, 25, 8, 31, 21, 32, 16, 21, 10, 25, 21
32, 3, 25, 8, 31, 21, 28, 16, 21, 10, 25, 21
32, 3, 25, 8, 31, 21, 28, 16, 21, 10, 25, 21

$C_{n,24}$ (Period beginning at Index 240):
5, 14, 37, 1, 4, 11, 23, 12, 38, 11, 7, 8
5, 14, 37, 1, 4, 11, 23, 12, 29, 11, 7, 8
5, 14, 37, 1, 4, 11, 23, 12, 29, 11, 7, 8
5, 14, 30, 1, 4, 11, 23, 12, 29, 11, 7, 8
5 14, 37, 1, 4, 11, 23, 12, 29, 11, 7, 8

$C_{n,32}$ (Period beginning at Index 180):
24, 14, 18, 1, 4, 17, 32, 8, 1, 2, 7, 8
24, 14, 18, 1, 4, 17, 35, 8, 1, 2, 7, 8
24, 14, 18, 1, 4, 17, 35, 8, 1, 2, 7, 8
24, 14, 18, 1, 4, 17, 32, 8, 1, 2, 7, 8
24, 14, 18, 1, 4, 17, 32, 8, 1, 2, 7, 8

$C_{n,33}$ (Period beginning at Index 300):
7, 32, 14, 27, 8, 41, 4, 7, 13, 27, 1, 2
7, 32, 14, 27, 8, 41, 4, 7, 13, 30, 1, 2
7, 32, 14, 27, 8, 41, 4, 7, 13, 32, 1, 2
7, 32, 14, 27, 8, 41, 4, 7, 13, 27, 1, 2
7, 29, 14, 30, 8, 41, 4, 7, 13, 27, 1, 2

$C_{n,34}$ (Period beginning at Index 240):
14, 3, 2, 1, 4, 38, 13, 4, 37, 2, 3, 29
14, 3, 2, 1, 4, 38, 13, 4, 27, 2, 3, 29
14, 3, 2, 1, 4, 38, 13, 4, 27, 2, 3, 29
14, 3, 2, 1, 4, 38, 13, 4, 37, 2, 3, 29
14, 3, 2, 1, 4, 38, 13, 4, 37, 2, 3, 29

$C_{n,35}$ (Period beginning at Index 360):
5, 15, 27, 1, 37, 38, 31, 32, 1, 35, 21, 8
5, 15, 27, 1, 37, 38, 31, 32, 1, 21, 38, 8
5, 15, 27, 1, 37, 38, 31, 32, 1, 21, 38, 8
5, 15, 27, 1, 37, 38, 31, 32, 1, 35, 21, 8
5, 15, 27, 1, 37, 38, 31, 32, 1, 35, 21, 8

$C_{n,36}$ (Period beginning at Index 360):
14, 42, 22, 1, 4, 22, 23, 12, 37, 2, 11, 32
14, 42, 22, 1, 4, 22, 23, 12, 42, 2, 11, 21

14, 42, 22, 1, 4, 22, 23, 12, 37, 2, 11, 21
14, 42, 22, 1, 4, 22, 23, 12, 37, 2, 11, 32
14, 42, 22, 1, 4, 22, 23, 12, 37, 2, 11, 32

$C_{n,38}$ (Period beginning at Index 240):
32, 7, 37, 1, 28, 32, 27, 13, 38, 10, 7, 18
32, 7, 37, 1, 28, 32, 27, 13, 38, 10, 7, 18
32, 7, 37, 1, 28, 32, 27, 13, 38, 10, 7, 18
32, 7, 37, 1, 28, 32, 27, 13, 38, 10, 7, 18
32, 7, 37, 1, 32, 41, 27, 13, 38, 10, 7, 18

$C_{n,39}$ (Period beginning at Index 300):
5, 48, 3, 21, 41, 35, 32, 21, 35, 22, 42, 25
5, 48, 3, 21, 41, 35, 32, 21, 35, 22, 42, 25
5, 48, 3, 21, 41, 35, 32, 21, 35, 22, 42, 25
5, 48, 3, 21, 41, 35, 32, 21, 35, 22, 42, 25
5, 48, 3, 21, 41, 35, 50, 21, 35, 22, 42, 25

$C_{n,40}$ (Period beginning at Index 300):
41, 7, 37, 44, 48, 27, 42, 4, 37, 10, 18, 24
41, 7, 37, 44, 48, 27, 42, 4, 37, 10, 18, 24
41, 7, 38, 44, 48, 27, 42, 4, 37, 10, 18, 24
41, 7, 38, 44, 48, 27, 42, 4, 37, 10, 18, 24
41, 7, 37, 44, 48, 27, 42, 4, 37, 10, 18, 24

$C_{n,42}$ (Period beginning at Index 300):
5, 15, 38, 1, 41, 11, 42, 28, 37, 16, 7, 8
5, 15, 38, 1, 41, 11, 42, 28, 37, 16, 7, 8
5, 15, 38, 1, 32, 11, 41, 28, 37, 16, 7, 8
5, 15, 38, 1, 32, 11, 41, 28, 37, 16, 7, 8
5, 15, 38, 1, 41, 11, 42, 28, 37, 16, 7, 8

$C_{n,44}$ (Period beginning at Index 360):
37, 16, 14, 44, 9, 25, 38, 37, 12, 38, 31, 26
37, 16, 14, 44, 9, 25, 38, 37, 12, 38, 31, 26
37, 16, 14, 44, 9, 25, 38, 44, 12, 38, 31, 26
37, 16, 14, 44, 9, 25, 38, 44, 12, 38, 31, 26
37, 16, 14, 44, 9, 25, 38, 37, 12, 38, 31, 26

$C_{n,48}$ (Period beginning at Index 360):
32, 48, 7, 35, 1, 11, 47, 44, 4, 21, 2, 11
32, 48, 7, 31, 1, 11, 47, 44, 4, 21, 2, 11

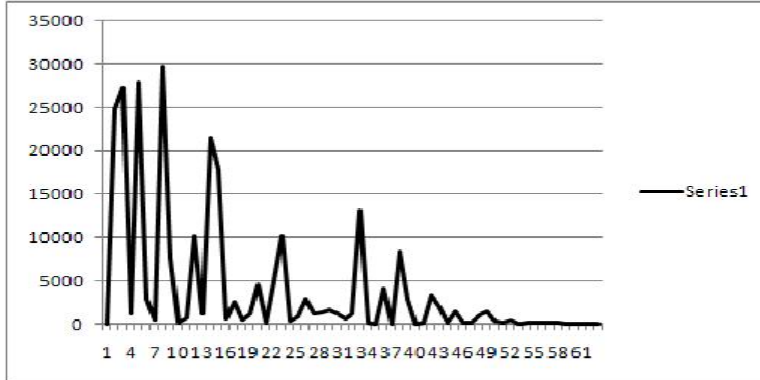
32, 48, 7, 31, 1, 11, 47, 44, 4, 21, 2, 11
44, 32, 7, 31, 1, 11, 47, 44, 4, 21, 2, 11
44, 32, 7, 31, 1, 11, 47, 44, 4, 21, 2, 11

$C_{n,49}$ (Period beginning at Index 360):
1, 32, 2, 1, 4, 7, 2, 32, 1, 2, 7, 8
1, 32, 2, 1, 4, 7, 2, 32, 1, 2, 7, 8
1, 32, 2, 1, 4, 7, 2, 32, 1, 2, 7, 8
1, 35, 2, 1, 4, 7, 2, 32, 1, 2, 7, 8
1, 35, 2, 1, 4, 7, 2, 32, 1, 2, 7, 8

$C_{n,50}$ (Period beginning at Index 360):
31, 14, 51, 32, 25, 22, 35, 12, 25, 32, 26, 37
31, 14, 51, 32, 25, 22, 31, 12, 25, 32, 26, 37
31, 14, 51, 32, 25, 22, 31, 12, 25, 32, 26, 37
42, 14, 48, 32, 25, 22, 31, 12, 25, 32, 26, 37
42, 14, 48, 32, 25, 22, 31, 12, 25, 32, 26, 37

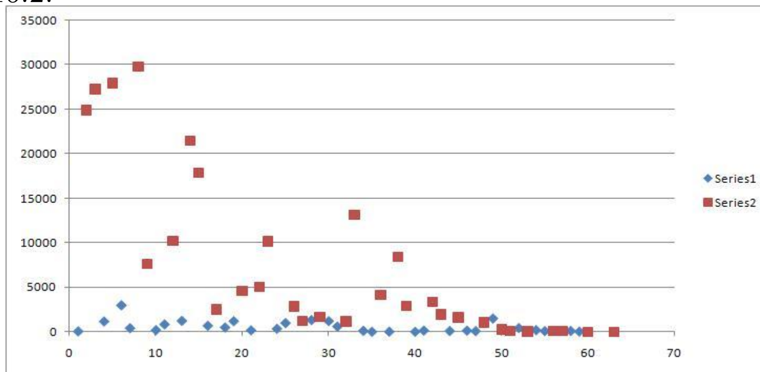
16. Appendix D: Tri-Path Charts

16.1.



Series 1 indicates the frequency of S-G numbers. The x-axis represents the S-G numbers and the y-axis is the number of times a given number occurs.

16.2.



This chart shows the frequency of evil and odious numbers. Series 1 is the evil numbers and series 2 is the odious numbers. Again, the x-axis represents the S-G numbers and the y-axis represents their frequency.

17. Appendix E: Supplemental Heursitics Charts

Below are some charts comparing the predicted vs. actual distributions of the S-G numbers for labeled graphs of order 5, 6, and 7. The numbers in the top row are the S-G numbers, while the numbers in the leftmost column indicate the number of edges.

5 vertices:

Figure 4.

edges	*type*	0	1	2	3	4	5	6	7	8	9	10
4	*heuristic*	9.3	0	0.4	0	90.3	*	*	*	*	*	*
4	*computed*	7.1	33.3	0	0	59.5	*	*	*	*	*	*
5	*heuristic*	27.4	9.6	21.0	0	0.5	41.5	*	*	*	*	*
5	*computed*	28.6	11.9	23.8	0	0	35.7	*	*	*	*	*
6	*heuristic*	6.2	1.7	18.0	0.5	0.1	5.2	68.3	*	*	*	*
6	*computed*	0	4.8	2.4	0	0	0	92.9	*	*	*	*
7	*heuristic*	0.5	1.1	1.8	8.0	0.4	0.2	0	88.0	*	*	*
7	*computed*	8.3	0	0	41.7	0	0	0	50.0	*	*	*
8	*heuristic*	1.5	4.8	3.9	1.2	14.5	0.9	0	0.3	72.9	*	*
8	*computed*	0	0	33.3	0	0	0	0	0	66.7	*	*
9	*heuristic*	4.5	32.8	0.5	0	62.2	0	0	0	0	0	*
9	*computed*	0	0	0	0	100	0	0	0	0	0	*
10	*heuristic*	17.7	64.4	0	0	0	17.9	0	0	0	0	0
10	*computed*	100	0	0	0	0	0	0	0	0	0	0

See following pages for charts of 6 and 7 vertices (Figure 5 and Figure 6, respectively).

Figure 5.

edges	*type*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	*heuristic*	27.2	63.6	1.7	0	7.5	*	*	*	*	*	*	*	*	*	*	*
4	*computed*	9.9	30.8	17.6	0	41.8	*	*	*	*	*	*	*	*	*	*	*
5	*heuristic*	27.1	9.8	10.3	0	3.5	49.3	*	*	*	*	*	*	*	*	*	*
5	*computed*	14.4	7.5	14.0	0	12.0	52.1	*	*	*	*	*	*	*	*	*	*
6	*heuristic*	15.6	2.6	7.0	0.4	0.5	0.9	73.0	*	*	*	*	*	*	*	*	*
6	*computed*	14.0	3.6	0.3	0	0	0	82.1	*	*	*	*	*	*	*	*	*
7	*heuristic*	2.0	2.7	3.5	7.1	0.5	0	0	84.1	*	*	*	*	*	*	*	*
7	*computed*	0.9	11.2	14.0	31.5	1.4	0	0	41.0	*	*	*	*	*	*	*	*
8	*heuristic*	4.0	9.6	8.3	3.8	17.9	0.7	0	0.8	54.9	*	*	*	*	*	*	*
8	*computed*	5.8	5.6	7.0	4.2	21.4	0	0	0	55.9	*	*	*	*	*	*	*
9	*heuristic*	7.9	3.0	1.6	0	5.8	18.7	0	0	0	62.8	*	*	*	*	*	*
9	*computed*	3.8	8.7	0	0	5.4	28.8	0	1.2	0	52.1	*	*	*	*	*	*
10	*heuristic*	8.5	1.5	1.8	0.2	0.1	2.9	0.5	0	0	0.1	84.4	*	*	*	*	*
10	*computed*	4.7	0	5.4	0	0	8.0	0	0	0	0	81.9	*	*	*	*	*
11	*heuristic*	3.0	0.4	2.7	1.6	0	0	16.5	0.7	0	0	0	74.9	*	*	*	*
11	*computed*	4.4	2.2	6.6	3.3	0	0	17.6	0	0	0	0	65.9	*	*	*	*
12	*heuristic*	1.0	1.4	1.3	17.3	0.5	0	7.7	16.5	0	0	0	0	54.2	*	*	*
12	*computed*	3.3	39.6	0	0	0	0	13.2	44.0	0	0	0	0	0	*	*	*
13	*heuristic*	0.6	0	0.4	19.2	15.1	0	0	0	5.5	0	0	0	59.1	0	*	*
13	*computed*	0	0	0	42.9	0	0	0	0	0	0	0	0	57.1	0	*	*
14	*heuristic*	0.6	0	1.1	0	78.7	0.8	0	0	18.8	0	0	0	0	0	0	*
14	*computed*	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	*
15	*heuristic*	2.6	0	0	0	97.4	0	0	0	0	0	0	0	0	0	0	0
15	*computed*	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0

Figure 6.

edges	*type*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
4	*heuristic*	22.5	55.8	5.4	0	16.3	*	*	*	*	*	*	*	*	*	*	*	*	*	*
4	*computed*	12.3	24.6	33.3	0	29.8	*	*	*	*	*	*	*	*	*	*	*	*	*	*
5	*heuristic*	22.7	12.5	4.3	0	10.3	50.2	*	*	*	*	*	*	*	*	*	*	*	*	*
5	*computed*	14.1	13.9	8.3	0	12.4	51.3	*	*	*	*	*	*	*	*	*	*	*	*	*
6	*heuristic*	17.8	5.7	4.0	7.6	3.5	0.8	60.6	*	*	*	*	*	*	*	*	*	*	*	*
6	*computed*	9.0	4.7	0.6	9.5	3.5	0	72.6	*	*	*	*	*	*	*	*	*	*	*	*
7	*heuristic*	10.1	9.5	15.7	32.2	3.8	0	0	28.7	*	*	*	*	*	*	*	*	*	*	*
7	*computed*	6.7	4.3	18.4	15.1	6.7	0	0	48.8	*	*	*	*	*	*	*	*	*	*	*
8	*heuristic*	5.4	10.4	6.0	5.2	8.1	0.9	0	0.3	63.7	*	*	*	*	*	*	*	*	*	*
8	*computed*	6.9	16.8	3.1	2.8	12.2	3.4	0.6	0	54.1	*	*	*	*	*	*	*	*	*	*
9	*heuristic*	7.2	5.2	1.8	2.0	5.5	27.9	0	0	0	50.3	*	*	*	*	*	*	*	*	*
9	*computed*	4.2	3.8	1.7	3.7	2.6	22.4	0.0	0.1	0.9	60.7	*	*	*	*	*	*	*	*	*
10	*heuristic*	6.4	0.9	5.4	5.4	2.2	3.2	5.1	0.1	0	0	71.3	*	*	*	*	*	*	*	*
10	*computed*	2.1	0.3	2.2	6.1	2.0	5.1	5.3	0.7	0	0	76.1	*	*	*	*	*	*	*	*
11	*heuristic*	2.9	0.5	5.8	2.4	1.4	0	11.6	0.6	0	0	0	74.7	*	*	*	*	*	*	*
11	*computed*	4.2	0.8	7.3	3.1	1.5	0.5	16.0	12.7	0	0	0	53.9	*	*	*	*	*	*	*
12	*heuristic*	7.0	9.0	7.6	2.2	7.0	0	1.4	5.2	0.2	0	0	0	60.4	*	*	*	*	*	*
12	*computed*	6.1	9.4	3.6	1.0	6.6	0.2	0.9	10.1	2.3	0	0	0	59.9	*	*	*	*	*	*
13	*heuristic*	2.3	7.9	1.1	2.1	6.1	0.6	0	0.1	11.2	0	0	0	0	68.7	*	*	*	*	*
13	*computed*	3.9	3.0	3.0	9.1	1.0	1.9	0	1.8	15.1	4.3	0	0	0.2	56.8	*	*	*	*	*
14	*heuristic*	0.6	0.6	0.7	2.3	1.8	16.9	0.1	0	2.7	6.2	0	0	0	0	68.1	*	*	*	*
14	*computed*	0	0	2.3	3.5	1.7	5.1	0	0	3.3	9.6	0	0	0	2.2	72.4	*	*	*	*
15	*heuristic*	0.1	0.1	0.2	0	0.6	6.2	0.3	0	0	0.7	0	0	0	0	0	91.9	*	*	*
15	*computed*	5.2	2.0	1.2	0.8	0	14.5	9.4	0	0.1	14.3	3.4	0	0	0	0	49.1	*	*	*
16	*heuristic*	0.4	0.5	0.5	0	1.9	0.1	2.9	0.3	0	0	12.8	0.1	0	0	0	0	80.5	*	*
16	*computed*	6.2	0	0	0	11.2	0.5	0	0	6.2	1.5	7.2	10.3	0	0	0	0	56.8	*	*
17	*heuristic*	0	1.2	0.8	0	0.7	0	0	6.4	0	0	2.1	3.0	0	0	0	0	0	85.6	*
17	*computed*	10.5	10.5	0	0	1.8	0	0	1.8	0	0	21.1	0	12.3	0	0	0	0	42.1	*
18	*heuristic*	0	1.3	6.2	1.6	0	0	0	39.6	0	0	0	0	0.8	0	0	0	0	0	50.5
18	*computed*	2.6	0	0	0	0	0	0	39.5	0	0	0	0	0	0	0	0	0	0	57.9

References

[1] Theory of impartial games. Lecture Notes, MIT, 2009.
[2] Julius G. Baron. The game of nim-a heuristic approach. Mathematics Magazine, 1974.
[3] Richard K. Guy Elwyn R. Berlekamp, John H. Conway. Winning ways for your mathematical plays, second edition. A K Peters Ltd., 2001.
[4] Thomas S. Ferguson. Game theory. Lecture Notes, Math 167, School, 2000.
[5] Masahiko Fukuyama. A nim game played on graphs. Theoretical Computer Science.