Daniel Thornton
2010-05-19
CS 493, Dr. Levy
Honors Thesis

Steinhaus Graph Connectivity: Initial Data and Analysis

Purpose

The goal of this thesis is to obtain and analyze data on the connectivity of Steinhaus graphs.

Definitions

A *graph* **G** = (V, E), where V is a set of vertices and E a set of edges, each of which connects two of the vertices. All the vertices and edges in a graph may be expressed in a binary adjacency matrix, where any entry (i, j) containing a '1' signifies an edge between the vertices numbered i and j. Naturally, this matrix is symmetric, since the same edge will appear for (i, j) and (j, i). Also, the diagonal of the matrix is composed of zeroes, since no edge may connect a vertex to itself. A graph's *degree sequence* is an array of numbers, where the nth number corresponds to the number of edges incident to the nth vertex. If all vertices have the same degree, the graph is said to be *regular.*

Steinhaus graphs are a special class of graph, each of which for $T = a_{0,0}a_{0,1}...a_{0,n-1}$ (an n-long string of 0's and 1's) has as its adjacency matrix the *Steinhaus matrix* $A = [a_{i,j}]$ where

$$a_{i,j} = \begin{cases} 0, & \text{if } 0 \le i = j \le n-1; \\ (a_{i-1,j-1} + a_{i-1,j}) \bmod 2, & \text{if } 0 < i < j \le n-1; \\ a_{j,i}, & \text{if } 0 \le j < i \le n-1. \end{cases}$$

That is, each entry *a* in the matrix results from the binary addition (see *Figure 1*) of the two entries above it (see [1]). Thus the entire Steinhaus matrix may be generated from the binary string *T*. In addition to this standard generator, the adjacency matrix may be constructed from a *diagonal generator,* defined in [1] as the entries $a_{i,\,i+1}$. Matrix generation from a diagonal generator is essentially the same process, where the addition instead propagates upward from the diagonal.

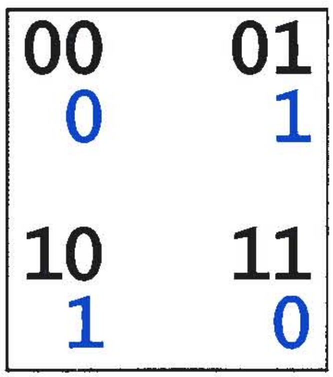

Figure 1: Binary addition rules.

A graph *component* is the set of all vertices connected by some path to a given vertex. That is, any two vertices in a component are connected by a series of edges. A graph consisting of only one component is said to be *connected.*

A graph's *connectivity* describes how many vertices may be removed before disconnecting the graph into multiple components. If any one vertex may be removed without disconnection, the graph is said to be two-connected. If any two vertices, the graph is three-connected, and so on. A *disconnecting set* is a set of vertices whose removal will disconnect the graph.

```
0010000101100001101100
0011000111010001011010
1100100100111001110111
0100110110100101001100
0011001101110111101010
0001001011001100011111
0000110110101010010000
1111101001111111011000
0101011001000000110100
1100110110100000101110
1011101101010000111001
0110100100101000100101
0010011100010100110111
0001110100001010101100
0000101100000101111010
1111100100000010000111
1010100011111110000100
0110011110101010000110
1101110101100110000101
1011010011011101111011
0110110001001011010100
0010010000111001001100
```

Figure 2: Example Steinhaus adjacency matrix.

As noted above, the aim of this work is to describe the connectivity of broad categories of Steinhaus graphs. The notation **C**(n, k) is the number of Steinhaus graphs on n vertices which are k-connected. My first goal was to write a program capable of determining **C**(n, k) for various values of n and k.

Original Program

I wrote the beginnings of this program during a Graph Theory course in Spring 2009. After requesting a binary or decimal generator, the program calculated and displayed whether the graph was regular, as well as its adjacency matrix, degree sequence, and other statistics. Given a vertex range, the program would iterate through all Steinhaus graphs in the range and display those which were regular. I would later use a similar approach to test large numbers of graphs for connectivity.

Connectivity Program

The determination of a graph's connectivity requires knowledge of whether a graph is connected or disconnected. For this reason, I began by writing an algorithm to count the components in a graph, with the following steps:

[1] Push the first/next vertex to the stack.
[2] Iterate through all vertices adjacent to this vertex, adding them to the stack and a vertex tracker (a binary array indicating which vertices have been traversed).
[3] Empty the stack, logging all vertices' adjacencies in the stack and the vertex tracker.
[4] Repeat step [3] until the stack is empty.
[5] Now one complete component has been traversed; increment the component count by one.
[6] If the vertex tracker shows all vertices have been examined, end algorithm.
[7] Otherwise, scan through the vertex tracker to find the next vertex and return to step [2].

Using method overloading, I later created a slightly-modified version of this algorithm which merely tested connectedness. It counted components as before, returning "false" if the first completed component did not account for the entire graph, and "true" otherwise.

The next step was to implement a connectivity test. Initially unsure of how to craft a *k*-connectivity test, I first implemented one- and two-connectedness tests. After testing these successfully, I generalized the approach to produce a *k*-connectedness test.

Essentially, this algorithm iterated through the graph's vertices, removing each in turn and keeping track of those removed. For each removal, the program recursed and began testing removals of each remaining vertex. In this way, it tried all possible disconnecting sets of size two, three, and so on up to a specified k. For each candidate disconnecting set, the algorithm queried the connectedness test above. As soon as it encountered a successful disconnecting set, the algorithm ended, having obtained the minimum k number of vertices which could disconnect the graph.

To determine a $C(n, k)$ count, the main program iterated over all Steinhaus generators of length n in order, running the k-connectedness algorithm on each. On completion, it displayed the total count of k-connected graphs and their standard generators.

Using this approach, I obtained data for many $C(n, k)$ cases. However, the algorithm began to hang on some larger cases, for example C(17, 10). In general, I found the algorithm could not successfully complete tests on graphs with diagonal generators of the form [100]1 (in all future discussion of generators, a bracketed portion indicates an element which may be repeated an arbitrary number of times).

Because of this inadequacy, I coded a new attempt at a successful k-connectivity test. For a given $C(n, k)$, the new algorithm moved recursively through all possible sets of $\binom{n}{k}$ vertices, running a connectedness test on each. This new approach succeeded where the previous failed, completing the C(17, 10) case in seconds.

Enhancements

While obtaining data for various C(n, k) cases, I also added some logical enhancements to the k-connectivity test. These included some conditional checks before the main algorithm. First, k obviously must be less than n, and the algorithm terminates if this is not the case. Also, k must be less than the vertex of minimal degree. Otherwise, the k vertices adjacent to this vertex may be removed, creating a single-vertex second component.

I also modified the k-connectivity test to first try a disconnecting set composed of the k vertices with highest degree. These modifications, especially the latter, significantly reduced the execution time of the algorithm. Later, I also added some initial tests with other probable disconnecting sets, based on patterns observed below.

Analysis

With the above alrogithms, I obtained a large amount of C(n, k) data (see Appendix for the full table). Among the most-connected graphs, which I term border cases, general patterns begin to emerge.

C(n, k)						
k	7	8	9	10	11	12
n						
10	0	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0
13	2	0	0	0	0	0
14	6	1	0	0	0	0
15	106	3	0	0	0	0
16	1018	19	2	0	0	0
17	5160	233	5	1	0	0
18	20178	2116	35	3	0	0
19	66281	12542	458	8	2	0
20	UNK	UNK	4473	74	5	1
21	UNK	UNK	UNK	849	17	3
22	UNK	UNK	UNK	UNK	93	7
23	UNK	UNK	UNK	UNK	UNK	21

Figure 3: Border cases from C(*n*, *k*) table.

As seen in *Figure* 3, all border cases eventually settle into a pattern of 1's, 2's, and 3's. Eventually, cases further from the border settle as well, as shown in *Figure* 4 below.

C(n, k)							
k	12	13	14	15	16	17	18
n							
18	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0
20	1	0	0	0	0	0	0
21	3	0	0	0	0	0	0
22	7	2	0	0	0	0	0
23	21	5	1	0	0	0	0
24	UNK	18	3	0	0	0	0
25	UNK	33	7	2	0	0	0
26	UNK	UNK	19	5	1	0	0
27	UNK	UNK	UNK	17	3	0	0
28	UNK	UNK	UNK	24	7	2	0
29	UNK	UNK	UNK	UNK	19	5	1
30	UNK	UNK	UNK	UNK	UNK	17	3
31	UNK	UNK	UNK	UNK	UNK	22	7
32	UNK	UNK	UNK	UNK	UNK	UNK	19

Figure 4: Regularity increases with greater *n* and *k* values.

Unfortunately, the C(*n*, *k*) cases do not appear to settle to constant values with any regularity. For example, 3 settles into regularity at $n = 15$ and 5 settles at $n = 17$, but 7 does not settle until $n = 22$. One easily-generalized pattern, however, is the relations of *n* and *k* where specific C(*n*, *k*) values occur. These are shown in *Figure 5* below. Note that *k* will always be even in those equations with `(k / 2)` and odd in those with `((k + 1) / 2)`. I did not obtain enough data to determine if this sequence of relations continues along alternating prime values of C(*n*, *k*).

C(*n*, *k*)	*n*/*k* Relation
1	`n = 3 * (k / 2) + 2`
3	`n = 3 * (k / 2) + 3`
7	`n = 3 * (k / 2) + 4`
19	`n = 3 * (k / 2) + 5`
2	`n = 3 * ((k + 1) / 2) + 1`
5	`n = 3 * ((k + 1) / 2) + 2`
17	`n = 3 * ((k + 1) / 2) + 3`

Figure 5: General *n*/*k* relations for regular C(*n*, *k*) values.

Next, I examined patterns in the generators for border and near-border C(*n*, *k*) cases. I tested graphs with generators composed of regular substrings (110 and 111000 for example) to see if these consistently placed in the border. Though I tried both standard and diagonal generators made from these substrings, there was no consistent pattern apparent. Looking at the generators for border cases, however, I observed significant patterns. All graphs in C(*n*, *k*) = 1 cases, for example, have standard generators of the form 01[110]. Below, *Figure 6* shows the pattern for all observed border cases.

```
1:

    01[110]

2:

   000[110]1
    01[110]11

3:

   000[110]
  0010[110]11
    01[110]1

5:

 00010[110]
   000[110]11
  0010[110]1
 01010[110]
    01[110]

7:

 00010[110]11
   000[110]1
  0010[110]
   001[110]1
 01010[110]11
    01[110]11
   011[110]1

17:

   0000[110]11
  00010[110]1
    000[110]
   0010[110]11
  00101[110]1
    001[110]
  00111[110]1
 010000[110]
0100010[110]11
   0100[110]11
  01001[110]1
 010100[110]
  01010[110]1
0110010[110]11
0111010[110]11
     01[110]1
    011[110]

19:

   0000[110]1
  00010[110]
    000[110]11
  00011[110]
   0010[110]1
  00101[110]
    001[110]11
  00111[110]
 010000[110]11
0100010[110]1
   0100[110]1
  01001[110]
 010100[110]11
  01010[110]
0110010[110]1
 011100[110]11
0111010[110]1
     01[110]
    011[110]11
```

Figure 6: Patterns of standard generators for border case graphs.

Another way to categorize these recurring generators is by the patterns that are present for a fixed value of C(n, k) that are not present for C(n, k) – 1. These added patterns are found in *Figure 7*.

```
1:

    01[110]

2 adds:

   000[110]1

3 adds:

  0010[110]11

5 adds:

 00010[110]
 01010[110]

7 adds:

   001[110]1
   011[110]1
```

```
17 adds:

   0000[110]11
  00101[110]1
  00111[110]1
 010000[110]
0100010[110]11
   0100[110]11
  01001[110]1
 010100[110]
0110010[110]11
0111010[110]11

19 adds:

  00011[110]
 011100[110]11
```

Figure 7: Patterns of standard generators for border cases, organized by additions.

I also performed the same organization on the diagonal generators for border C(n, k) cases, though no higher-level patterns emerged than those above. These are shown in *Figure 8*.

```
1:

1[001]

2 adds:

[001]

3 adds:

01[001]

5 adds:

00[111000]11
11[000111]00

7 adds:

0[111000]11
1[000111]00
```

```
17 adds:

0001110001110001110001110O
0100111110010100111110010l
0111110010100111110010100l
1111100101001111100101001l
1111001010011111001010011l
1110001110001110001110001l
1110010100111110010100111l
1100101001111100101001111l
1010011111001010011111001O
1001010011111001010011111O

19 adds:

0010100111110010100111110010
1001111100101001111100101001
```

Figure 8: Patterns of diagonal generators for border cases, organized by additions.

Another potential area for patterns to emerge is in the disconnecting sets of these border cases. The Figures below list the vertices *not* in the disconnecting set (numbering begins at one). These consist mostly of multiples of three, beginning at either one, two, or three, with minor additions. In the simpler cases, clear patterns emerge, as shown in *Figure 9*.

```
C(13, 7) = 2

0001101101101 = [2, 5, 8, 10, 11, 12] = 2-11; 10, 12 = 2-3, n-1, n-3
0111011011011 = [3, 6, 9, 11, 12, 13] = 3-12; 11, 13 = 3-3, n, n-2

C(16, 9) = 2

0001101101101101 = [2, 5, 8, 11, 13, 14, 15] = 2-14; 13, 15 = 2-3, n-1, n-3
0111011011011011 = [3, 6, 9, 12, 14, 15, 16] = 3-15; 14, 16 = 3-3, n, n-2

C(19, 11) = 2

0001101101101101101 = [2, 5, 8, 11, 14, 16, 17, 18] = 2-17; 16, 18 = 2-3, n-1, n-3
0111011011011011011 = [3, 6, 9, 12, 15, 17, 18, 19] = 3-18; 17, 19 = 3-3, n, n-2
```

Figure 9: Disconnecting set patterns for C(*n*, *k*) = 2 cases.

Each list is followed by a summary of the three-multiples and additions, then by a general notation listing the three-multiple pattern ("2-3" is multiples of three starting at two, &c) and additions. In some larger cases, such as C(*n*, *k*) = 17, some disconnecting sets are harder to classify, as shown in *Figure 10.*

```
C(27, 15) = 17

[illegible] = [1, 4, 7, 10, 13, 16, 19, 22, 23, 25, 26, 27]  = 1-25; 23, 26, 27          = 1-3, n, n-1, n-4
[illegible] = [3, 6, 9, 12, 15, 18, 21, 22, 24, 25, 26, 27]  = 3-27; 22, 25, 26          = 3-3, n-1, n-2, n-5
[illegible] = [2, 5, 8, 11, 14, 17, 20, 23, 25, 26, 27]      = 2-26; 25, 27              = 2-3, n, n-2
[illegible] = [1, 4, 7, 10, 13, 16, 19, 22, 24, 25, 26]      = 1-25; 24, 26              = 1-3, n-1, n-3
[illegible] = [3, 6, 9, 11, 12, 15, 16, 17, 19, 21, 24, 27]  = 3-27, '18; 11, 16, 17, 19 =
[illegible] = [2, 5, 8, 11, 14, 17, 20, 21, 23, 24, 25, 26]  = 2-26; 21, 24, 25          = 2-3, n-2, n-3, n-6
[illegible] = [3, 6, 9, 12, 15, 17, 19, 20, 21, 24, 25, 27]  = 3-27, '18; 17, 19, 20, 25 =
[illegible] = [3, 6, 11, 12, 13, 15, 16, 17, 18, 21, 24, 27) = 3-27, '9; 11, 13, 16, 27  =
[illegible] = [3, 6, 9, 12, 13, 14, 15, 17, 18, 19, 24, 27]  = 3-27, '21; 13, 14, 17, 19 =
[illegible] = [1, 4, 7, 10, 13, 16, 19, 22, 23, 24, 25, 27]  = 1-25; 23, 24, 27          = 1-3, n, n-3, n-4
[illegible] = [3, 6, 9, 12, 15, 17, 18, 21, 22, 23, 25, 27]  = 3-27, '24; 17, 22, 23, 25 =
[illegible] = [3, 6, 9, 10, 12, 13, 14, 15, 18, 21, 24, 27]  = 3-27, '6; 8, 10, 13, 14   =
[illegible] = [3, 6, 9, 12, 15, 18, 21, 22, 23, 24, 26, 27]  = 3-27; 22, 23, 26          = 3-3, n-1, n-4, n-5
[illegible] = [4, 5, 6, 8, 9, 10, 11, 14, 17, 20, 23, 26]    = 5-26; 4, 6, 9, 10         =
[illegible] = [3, 6, 9, 12, 15, 18, 21, 22, 23, 24, 26, 27]  = 3-27; 22, 23, 26          = 3-3, n-1, n-4, n-5
[illegible] = [3, 6, 9, 12, 15, 18, 21, 23, 24, 25, 27]      = 3-27; 23, 25              = 3-3, n-2, n-4
[illegible] = [2, 5, 8, 11, 14, 17, 20, 21, 22, 23, 25, 26]  = 2-26; 21, 22, 25          = 2-3, n-2, n-5, n-6
```

Figure 10: Disconnecting set patterns for C(27, 15) = 17.

I attempted to fit these odd disconnecting sets to the pattern of the rest by searching for alternate disconnecting sets. Unfortunately, substituting the disconnecting sets of other graphs in the group yielded no results. Slight modifications to the positions of additions were equally fruitless. These difficulties seem to indicate that the disconnecting sets of border and near-border C(*n*, *k*) cases are unique.

Future Research

First and foremost, I am intrigued to see if the observed pattern of settling and regularity at the border of the C(*n*, *k*) data continues. A description of the overall pattern of border regularity would allow a succinct characterization of the connectivity of a large subset of the Steinhaus graphs. Based on the patterns of disconnecting sets observed, future work could also produce a connectivity algorithm that first tests a wide array of likely disconnecting sets, based on past data.

References

[1] Wayne M. Dymàček, Matthew Koerlin, and Tom Whaley, A survey of Steinhaus graphs, Proc. 8th Quadrennial International Conf. on Graph Theory, Combinatorics, Algorithms and Application, Kalamazoo, Mich. (1996), volume I, pages 313–323.

Appendix

C(n, k)																		
k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
n																		
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	7	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	15	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	31	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	63	19	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	127	96	22	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	255	225	116	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	511	[illegible]	[illegible]	87	3	0	0	0	0	0	0	0	0	0	0	0	0	0
11	[illegible]	[illegible]	[illegible]	[illegible]	28	1	0	0	0	0	0	0	0	0	0	0	0	0
12	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	6	0	0	0	0	0	0	0	0	0	0	0	0
13	[illegible]	5003	3978	2607	[illegible]	79	3	0	0	0	0	0	0	0	0	0	0	0
14	[illegible]	8086	[illegible]	[illegible]	[illegible]	453	6	1	0	0	0	0	0	0	0	0	0	0
15	[illegible]	[illegible]	[illegible]	17907	[illegible]	2268	196	1	0	0	0	0	0	0	0	0	0	0
16	[illegible]	32629	[illegible]	[illegible]	[illegible]	7741	[illegible]	[illegible]	[illegible]	0	0	0	0	0	0	0	0	0
17	[illegible]	[illegible]	64253	[illegible]	[illegible]	[illegible]	[illegible]	233	5	1	0	0	0	0	0	0	0	0
18	[illegible]	[illegible]	[illegible]	122566	[illegible]	60888	[illegible]	[illegible]	35	3	0	0	0	0	0	0	0	0
19	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	8	2	0	0	0	0	0	0	0
20	[illegible]	[illegible]	[illegible]	[illegible]	468005	UNK	UNK	UNK	4471	34	[illegible]	1	0	0	0	0	0	0
21	[illegible]	1048335	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	859	[illegible]	8	0	0	0	0	0	0
22	[illegible]	2096896	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	93	7	[illegible]	0	0	0	0	0
23	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	21	[illegible]	1	0	0	0	0
24	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	[illegible]	3	0	0	0	0
25	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	33	7	2	0	0	0
26	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	19	5	1	0	0
27	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	[illegible]	3	0	0
28	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	24	7	2	0
29	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	18	[illegible]	[illegible]
30	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	[illegible]	3
31	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	[illegible]	7
32	[illegible]	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	UNK	19

Figure 11: All C(*n*, *k*) data.

- A fuller view of the data in *Figure 11* may be accessed at: http://tinyurl.com/steinhaus-1
- The generator strings for all C(*n*, *k*) border cases may be accessed at: http://tinyurl.com/steinhaus-2
- The generator strings for all regular C(*n*, *k*) cases analyzed may be accessed at: http://tinyurl.com/steinhaus-3
- The complete list of analyzed disconnecting set patterns excerpted in *Figure 9* and *Figure 10* may be accessed at: http://tinyurl.com/steinhaus-4
- All of the above data may also be obtained from Professor Wayne M. Dymàček, Department of Mathematics, Washington and Lee University.