

Degree Distributions for Small Graphs

Mike French

A report submitted in partial fulfillment
of the requirements for the course:

Social Network Analysis

Coursera

December 2013

Abstract

We enumerate all non-isomorphic simple undirected graphs on n nodes. We calculate their degree distributions and find sets of graphs that have the same histogram (equivalence classes). We show that histograms uniquely determine graphs for $n=2,3,4$. We find the first ambiguous cases of non-isomorphic graphs with the same histogram for $n=5$, and show they are homeomorphic (topologically equivalent). We then find many graphs for $n=6$ that have the same histogram, but are topologically distinct.

Contents

1. Problem
 - 1.1 Graphs
 - 1.2 Degree Distributions
 - 1.3 Graph Isomorphism
 - 1.4 Graph Enumeration
 - 1.5 Topological Equivalence
2. Solution
 - 2.1 Approach
 - 2.2 Implementation
 - 2.3 Performance
3. Results
 - 3.1 Trivial Graphs ($n=2,3,4$)
 - 3.2 Case $n=5$
 - 3.3 Case $n=6$
4. Conclusions
5. References
- Appendix A – Source Code
 - A.1 Core Class
 - A.2 Utility Class
 - A.3 Construction Utilities
 - A.4 Output Utilities

1. Problem

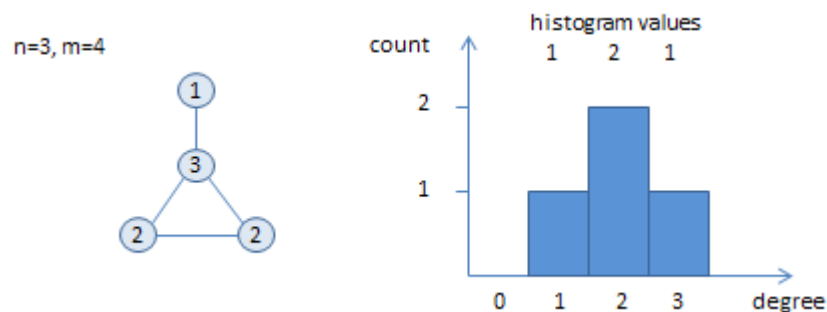
1.1 Graphs

Consider graphs with n nodes and m edges, which are:

- *Simple*: no edges with both ends at the same node (self-loops); and at most one edge between any pair of nodes.
- *Connected*: no isolated nodes that have no incident edges; so there is at least one path of edges between any pair of nodes.
- *Undirected*: edges can be traversed in either direction.
- *Unlabelled*: nodes are indistinguishable, there are no unique identifiers or subset markers (colours) associated with nodes (similarly for edges).

1.2 Degree Distributions

The *degree* of each node, d , is the number of edges incident on the node, and hence the number of other nodes to which it is connected (simple graph). The counts for all nodes in a graph can be collected in a histogram, which shows the degree distribution:



The sum of all degrees in the graph will be equal to twice the number of edges, $2m$. The graph is connected so the minimum degree is 1. The maximum degree is between 1 and $n-1$ inclusive, because there are no self-loops, and at most one edge between a node and each of the remaining nodes.

A *star* hub-and-spoke pattern will have a count of $(n-1)$ in the 1st column for the ends of the spokes, and a 1 for the hub in the $(n-1)$ th column. A *line* will have 2 in the 1st column, for the ends, then $n-2$ in the 2nd column, for the intermediate nodes. A single *ring* that includes all the nodes will just have a value of n in the 2nd column. The fully connected *clique* has leading zeroes followed by a single count of n in the $(n-1)$ th column. Here are examples for $n=5$:

Star	Line	Ring	Clique
4 0 0 1 (n-1) 0 0 1	2 3 0 0 2 (n-2) 0 0	0 5 0 0 0 n 0 0	0 0 0 5 0 0 0 n

For a relatively homogeneous *sparse* graph, the maximum degree will typically be less than $\log(n)$, so histograms will usually be small compared to the size of the graph. Histograms are cumulative data structures using simple addition, which is associative and commutative, so the node values can be collected in parallel and combined in any order (*e.g.* for large distributed graph). So for many practical graphs, the degree histograms will be small and easy to collect.

1.3 Graph Isomorphism

Two graphs are *isomorphic* if they have the same shape and structure, so that one can be completely superimposed on the other, such that all nodes and edges correspond with a single element of the other graph. More technically, we can say there must be two correspondences (1-to-1 mappings), one for the nodes and one for the edges, such that an edge connecting two nodes in one graph is mapped to an edge that connects the mappings of the original nodes in the second graph¹. Since corresponding nodes must have the same degree:

Isomorphic graphs have the same degree distribution histogram.

As an example of this principle, consider the *sub-graph isomorphism problem*, *i.e.* find one or more parts of a large target graph that match a smaller test graph. The principle says that a candidate sub-graph cannot be a match if it does not have the same degree histogram as the test graph.

1.4 Graph Enumeration

The number of non-isomorphic simply connected unlabelled graphs on n nodes is given by the following integer sequence [1]:

n nodes	2	3	4	5	6	7	8	9	10
# graphs	1	2	6	21	112	853	11,117	261,080	11,716,571

We will enumerate all the distinct graphs for $n=2,3,4,5,6$ and calculate their degree histograms. The number of edges will vary from $m=n-1$ for the minimally connected trees, up to $m=n(n-1)/2$ for the maximally connected *clique*.

We have seen that isomorphic graphs must have the same degree histogram. We will show that the inverse is not true, *i.e.* graphs with the same degree histogram are not necessarily isomorphic. We will find the smallest counter-examples.

1.5 Topological Equivalence

A graph of nodes and edges can be considered as a *1D complex* of continuous *lines* (curves) connected together at *vertices* to form various patterns. A node with $d=1$ becomes the free end of a line, called a *tail*. Nodes with degree $d>2$ become significant junctions between the line segments. The lines themselves may have intermediate nodes with $d=2$, but these are not topologically significant. Nodes can be added or removed from the linear segments without changing the overall connectivity, except that loops must not be shrunk to self-loops, or made to disappear completely.

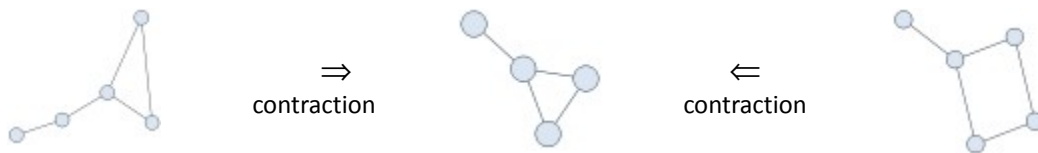
¹ There is another equivalence, rather confusingly called *homomorphism*, that relaxes the 1-to-1 restriction, allowing graphs to be folded over, so that multiple nodes and edges map to the same target element.

A *node contraction* is the removal of a $d=2$ node that does not lead to the collapse of a loop, or the creation of a self-loop. The two edges incident on the node are joined together, to form a single new edge. The process can be repeated until no more nodes can be removed. The result is a *contraction* of the original graph.

Homeomorphism: Two graphs are topologically equivalent if they share a common contraction.

The intuition here is that it doesn't matter how many intermediate edges there are on linear segments of the graph. Removing these nodes and edges does not make any difference to the overall shape and connectivity of the graph. However, the tails ($d=1$), loops and high degree junctions ($d>2$) are significant.

Here are two $n=5$ graphs, consisting of a loop and tail, which are topologically equivalent, because they have a common contraction to an $n=4$ graph with a minimal loop and minimal tail:



Contractions only modify the $d=2$ count in the histogram, so *if the contractions have the same number of steps, then the original graphs will have the same degree distribution.*

2. Solution

2.1 Approach

The objective is to automatically generate tables of unique connected undirected simple graphs on n nodes. There will be one table for each value of $n=2,3,4,5,6$. The tables will have a row for each graph, including a diagram, the list of node degrees, and the degree histogram.

The graph generation pipeline will be:

```
generate all graphs on n nodes
  filter out non-connected graphs
    filter out isomorphic graphs
```

Nodes are represented by unique integer labels. An edge is a pair of nodes. A graph is represented as a set of nodes and edges. There are no adjacency lists held in the nodes, and no explicit adjacency matrix.

An adjacency matrix approach is used during graph construction. The upper right triangle of the adjacency matrix contains all the necessary information to build the graph, because the edges are undirected (lower left triangle is just the mirror image), and there are no self-loops (no entries on the diagonal). The $n(n-1)/2$ bits of the upper right triangle can be concatenated into a single bitmask and represented as an unsigned integer. To enumerate all graphs, just count from 0 (no edges, all nodes disconnected) up to $2^{n(n-1)/2}$ (all possible edges are present, the graph is a clique), then interpret that number as a bitmask of edges.

A morphism is a mapping between sets of node labels. Two graphs are isomorphic if there is a permutation of the nodes that transforms one into the other. Graphs are not isomorphic if they have different numbers of nodes or edges, or they have different degree distributions.

The connectedness algorithm starts by assigning each node to a separate set. Then it loops over the edges, merging the two sets containing the nodes at the endpoints. Each set corresponds to a connected component, so the number of sets left at the end is the number of connected components in the graph.

For each resulting graph, a picture will be generated using GraphViz [2]. The graph will be generated in DOT format, then transformed to a PNG image using the *neato* tool.

The document generation pipeline will be:

```
for each table (n)
  build list of unique graphs on n nodes
  for each unique graph:
    generate DOT format file
    invoke neato to convert DOT to PNG
  build HTML table with column headers
  for each unique graph:
    generate HTML table row containing:
      image tag with PNG reference
      text for the graph degrees and histogram
```

The HTML tables will then be imported directly into this report (see Section 3).

2.2 Implementation

The implementation is in Scala [3], a multi-paradigm programming language (object oriented and functional), which compiles to Java bytecode and runs on the Java Virtual Machine (JVM).

The code is included in Appendix A, there are 4 sections:

1. The main `Graph` class, with methods to calculate degrees, histograms, connectedness and isomorphism.
2. A utility class `MergeSets` to help calculate the number of connected components.
3. Construction utilities to build various types of graph, including enumeration of all graphs for a given n , and filters for connectedness and isomorphism. There is also a main program to build the final tables.
4. Output utilities to generate all the HTML tables, create DOT files, invoke *neato* to generate images and write an HTML table row for a graph.

Scala is concise and most algorithms can be implemented in less space than their textual descriptions. The main `Graph` class is 2 pages of code (~77 SLOC) and the utilities are also 2 pages.

2.3 Performance

The program is designed to run once to generate the tables, so efficiency is not that important. The algorithms are brute force search through the space of permutations and isomorphisms. The most highly optimized public code for graph enumeration and analysis is probably *nauty*, written by Bill McKay [4]. My naïve code does not even attempt to walk in the shadow of *nauty*.

There is a fundamental limit of $n < 12$ in the implementation, because that is the biggest edge mask that can be represented in a `Long` (64-bit) integer. However, the exponential use of memory and cpu, mean the practical limit is much lower. For this report we will only go up to $n=6$.

A few minor optimizations have been implemented. For example, the degree histogram is cached using the Memoization Pattern [5]. The next most beneficial optimization would be to make the isomorphic check use lazy lists (streams), so that permutations and morphisms are only generated when needed.

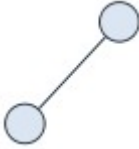
Unfortunately *neato* does not always layout planar graphs to remove all edge crossings. The main problem is with twisted quadrilaterals, where a 4-cycle that could be drawn as a square is drawn crossed over in a bow tie shape. For example, see graphs 4 and 5 for $n=4$ in the next section.

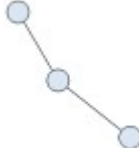
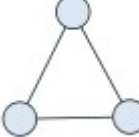
3. Results




3.1 Trivial Graphs ($n=2,3,4$)

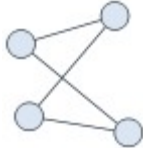
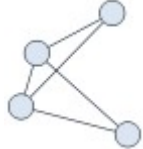
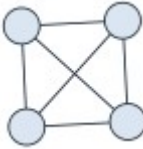
The first few graphs are easy to enumerate. The results are presented in a table with a list of node degrees, then the histogram is shown with counts for $d=1,2,\dots (n-1)$.

For each n , the graphs are listed in descending order of 'histogram number' (evaluated as a number in base $n+1$), so that graphs with the same histogram will appear next to each other. The first graph in the list will be the *star* pattern, and the last will be the *clique*.

Id	Graph	Degrees	Histogram
1		1,1	2

Id	Graph	Degrees	Histogram
1		1,1,2	21
2		2,2,2	03

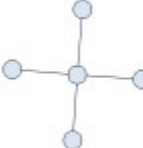


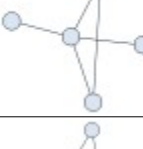
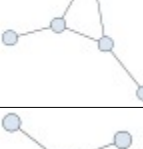

Id	Graph	Degrees	Histogram
1		1,1,1,3	301
2		1,1,2,2	220
3		1,2,2,3	121


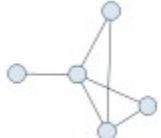
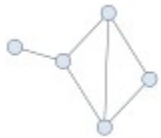
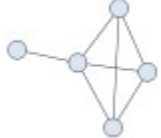

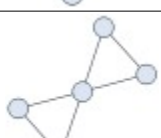
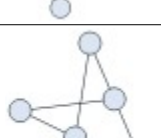

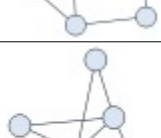
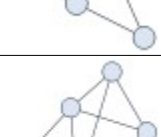
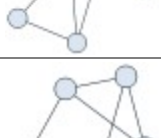
4		2,2,2,2	040
5		2,2,3,3	022
6		3,3,3,3	004

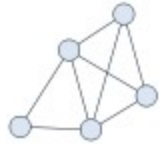
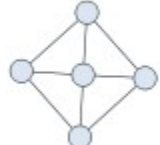
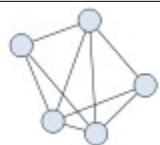

It is easy to see that all the graphs have distinct degree histograms. So we can conclude that the degree histogram uniquely identifies very small graphs ($n=2,3,4$) up to graph isomorphism.

3.2 Case $n=5$

The first non-trivial case is $n=5$. There are 21 unique graphs.

Id	Graph	Degrees	Histogram
1		1,1,1,1,4	4001
2		1,1,1,2,3	3110
3		1,1,2,2,2	2300
4		1,1,2,2,4	2201
5		1,1,2,3,3	2120
6		1,2,2,2,3	1310

7		1,2,2,2,3	1310
8		1,2,2,3,4	1211
9		1,2,3,3,3	1130
10		1,3,3,3,4	1031
11		2,2,2,2,2	0500
12		2,2,2,2,4	0401
13		2,2,2,3,3	0320
14		2,2,2,3,3	0320
15		2,2,2,4,4	0302
16		2,2,3,3,4	0221
17		2,3,3,3,3	0140



18		2,3,3,4,4	0122
19		3,3,3,3,4	0041
20		3,3,4,4,4	0023
21		4,4,4,4,4	0005

So we have two pairs of graphs that have the same degree histogram. That is 4 graphs out of 21, so about 20%. Here is the first pair for histogram 1310:

1310		
------	--	---

It is easy to see that these are both topologically equivalent to a loop and a tail, it is just that one segment (node) has been transferred between the loop and the tail.

Here is the second pair with histogram 0320, including some rearrangements to remove edge crossings and reveal the underlying structure:


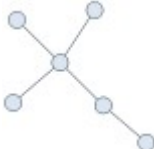



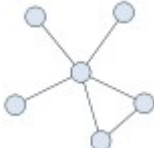



320		
-----	---	--



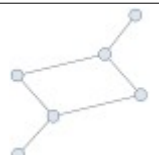
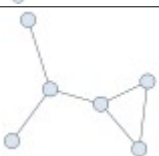
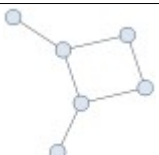
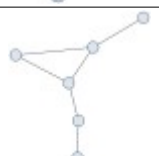
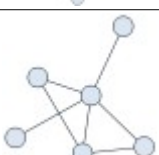
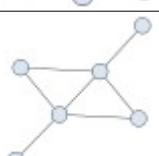
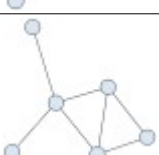
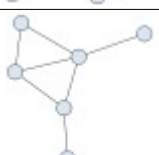
It is also easy to see that these are both topologically equivalent to two loops with shared edges. Another way to think about the graphs is as two high degree junctions, connected by 3 linear spans (right layouts). Then the difference between them is the number of edges in the spans: 1,2,3 or 2,2,2. The spans can be shortened by contraction, so they are homeomorphic.


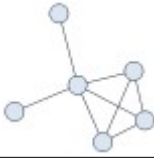
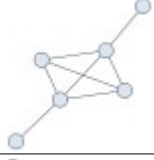
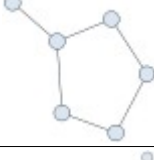

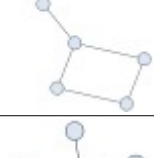
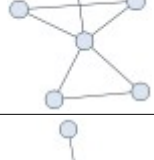
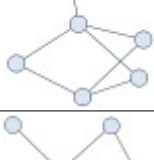
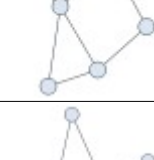

Both pairs of non-isomorphic graphs for $n=5$, which have the same degree histogram, are *homeomorphic* (topologically equivalent).

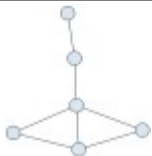
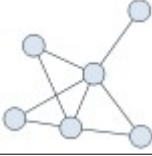
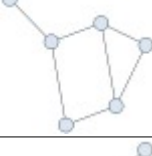
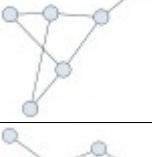
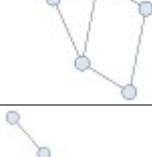
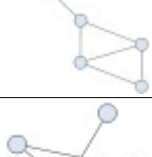
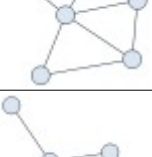
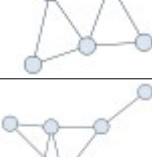
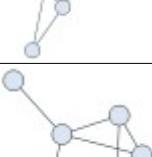
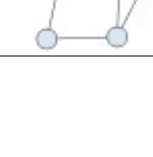
3.3 Case n=6

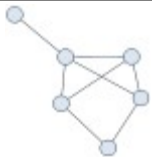
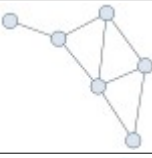
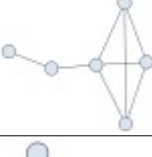
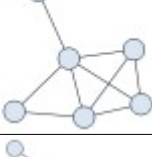
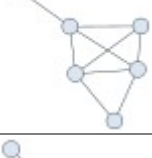
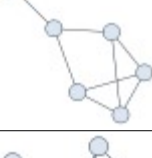
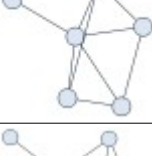
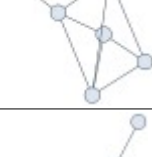
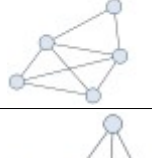
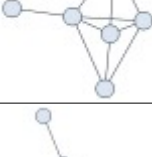
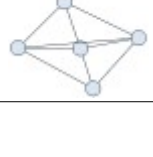
There are 112 cases for n=6 and there are many equivalence classes:



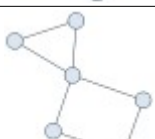
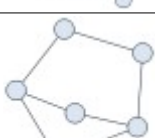
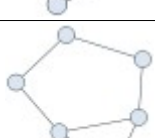
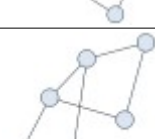
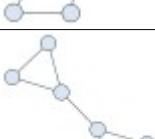
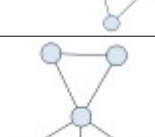
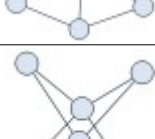
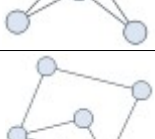
Id	Graph	Degrees	Histogram
1		1,1,1,1,1,5	50001
2		1,1,1,1,2,4	41010
3		1,1,1,1,3,3	40200
4		1,1,1,2,2,3	32100
5		1,1,1,2,2,3	32100
6		1,1,1,2,2,5	32001
7		1,1,1,2,3,4	31110
8		1,1,1,3,3,3	30300
9		1,1,2,2,2,2	24000

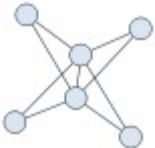

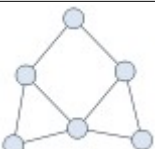
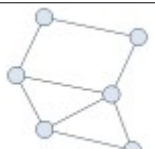
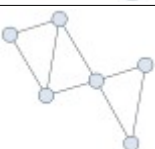

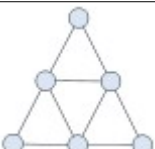
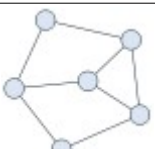
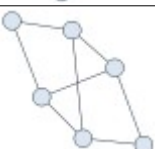
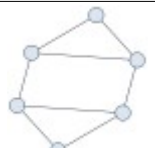
10		1,1,2,2,2,4	23010
11		1,1,2,2,2,4	23010
12		1,1,2,2,3,3	22200
13		1,1,2,2,3,3	22200
14		1,1,2,2,3,3	22200
15		1,1,2,2,3,3	22200
16		1,1,2,2,3,5	22101
17		1,1,2,2,4,4	22020
18		1,1,2,3,3,4	21210
19		1,1,2,3,3,4	21210

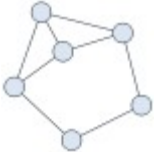


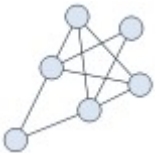



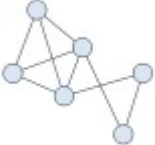
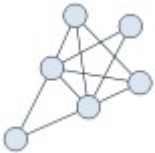
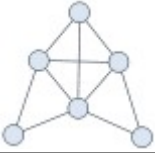

20		1,1,3,3,3,3	20400
21		1,1,3,3,3,5	20301
22		1,1,3,3,4,4	20220
23		1,2,2,2,2,3	14100
24		1,2,2,2,2,3	14100
25		1,2,2,2,2,3	14100
26		1,2,2,2,2,5	14001
27		1,2,2,2,3,4	13110
28		1,2,2,2,3,4	13110
29		1,2,2,2,3,4	13110

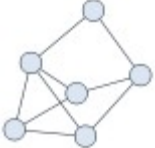
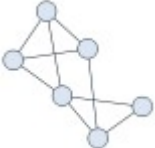


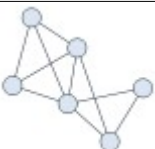


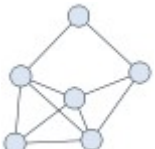
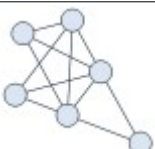
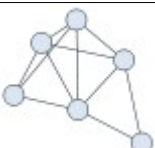
30		1,2,2,2,3,4	13110
31		1,2,2,2,4,5	13011
32		1,2,2,3,3,3	12300
33		1,2,2,3,3,3	12300
34		1,2,2,3,3,3	12300
35		1,2,2,3,3,3	12300
36		1,2,2,3,3,5	12201
37		1,2,2,3,4,4	12120
38		1,2,2,3,4,4	12120
39		1,2,3,3,3,4	11310


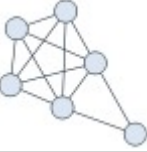
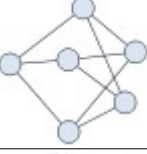

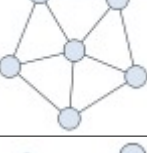
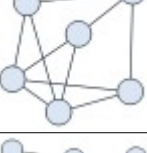
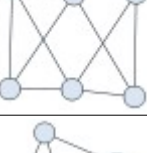
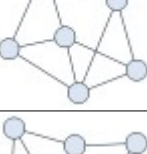
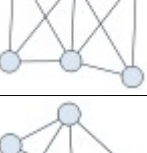
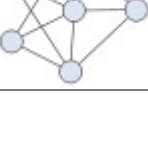
40		1,2,3,3,3,4	11310
41		1,2,3,3,3,4	11310
42		1,2,3,3,3,4	11310
43		1,2,3,3,4,5	11211
44		1,2,3,4,4,4	11130
45		1,3,3,3,3,3	10500
46		1,3,3,3,3,5	10401
47		1,3,3,3,4,4	10320
48		1,3,3,3,4,4	10320
49		1,3,3,4,4,5	10221
50		1,3,4,4,4,4	10140

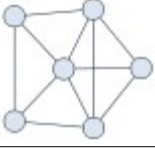
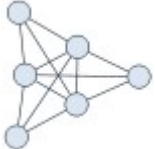
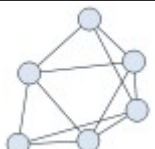
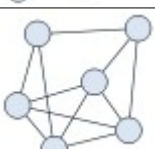
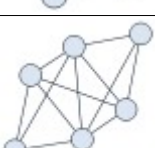
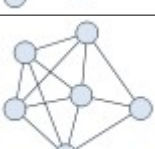
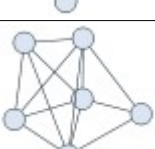
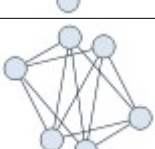
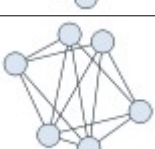
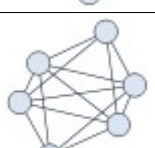
51		1,4,4,4,4,5	10041
52		2,2,2,2,2,2	06000
53		2,2,2,2,2,4	05010
54		2,2,2,2,3,3	04200
55		2,2,2,2,3,3	04200
56		2,2,2,2,3,3	04200
57		2,2,2,2,3,3	04200
58		2,2,2,2,3,5	04101
59		2,2,2,2,4,4	04020
60		2,2,2,2,4,4	04020


61		2,2,2,2,5,5	04002
62		2,2,2,3,3,4	03210
63		2,2,2,3,3,4	03210
64		2,2,2,3,3,4	03210
65		2,2,2,3,3,4	03210
66		2,2,2,3,4,5	03111
67		2,2,2,4,4,4	03030
68		2,2,3,3,3,3	02400
69		2,2,3,3,3,3	02400
70		2,2,3,3,3,3	02400

71		2,2,3,3,3,3	02400
72		2,2,3,3,3,5	02301
73		2,2,3,3,3,5	02301
74		2,2,3,3,4,4	02220
75		2,2,3,3,4,4	02220
76		2,2,3,3,4,4	02220
77		2,2,3,3,4,4	02220
78		2,2,3,3,4,4	02220
79		2,2,3,3,5,5	02202
80		2,2,3,4,4,5	02121
81		2,2,4,4,4,4	02040

82		2,3,3,3,3,4	01410
83		2,3,3,3,3,4	01410
84		2,3,3,3,3,4	01410
85		2,3,3,3,4,5	01311
86		2,3,3,3,4,5	01311
87		2,3,3,4,4,4	01230
88		2,3,3,4,4,4	01230
89		2,3,3,4,4,4	01230
90		2,3,3,4,5,5	01212
91		2,3,4,4,4,5	01131

92		2,4,4,4,4,4	01050
93		2,4,4,4,5,5	01032
94		3,3,3,3,3,3	00600
95		3,3,3,3,3,3	00600
96		3,3,3,3,3,5	00501
97		3,3,3,3,4,4	00420
98		3,3,3,3,4,4	00420
99		3,3,3,3,4,4	00420
100		3,3,3,3,5,5	00402
101		3,3,3,4,4,5	00321

102		3,3,3,4,4,5	00321
103		3,3,3,5,5,5	00303
104		3,3,4,4,4,4	00240
105		3,3,4,4,4,4	00240
106		3,3,4,4,5,5	00222
107		3,4,4,4,4,5	00141
108		3,4,4,5,5,5	00123
109		4,4,4,4,4,4	00060
110		4,4,4,4,5,5	00042
111		4,4,5,5,5,5	00024

112		5,5,5,5,5,5	00006
-----	---	-------------	-------

There are 23 equivalence classes that have more than one member. The size of the classes varies from 2 to 5. The total number of graphs that have ambiguous degree histograms is 66 out of 112 *i.e.* nearly 60%. Here are the multiple equivalence classes for $n=6$:

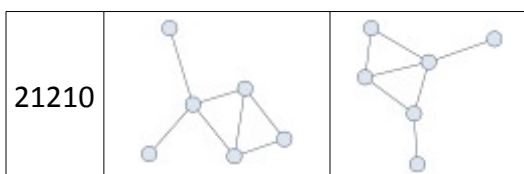
Id	Histogram	Graphs	Count	Homeomorphic
1	32100	4, 5	2	✓
2	23010	10,11	2	✓
3	22200	12,13,14,15	4	✗
4	21210	18,19	2	✗
5	14100	23,24,25	3	✓
6	13110	27,28,29,30	4	✗
7	12300	32,33,34,35	4	✓
8	12120	37,38	2	✗
9	11310	39,40,41,42	4	✗
10	10320	47,48	2	✗
11	04200	54,55,56,57	4	✗
12	04020	59,60	2	✓
13	03210	62,63,64,65	4	✗
14	02400	68,69,70,71	4	✗
15	02301	72,73	2	✗
16	02220	74,75,76,77,78	5	✗
17	01410	82,83,84	3	✗
18	01311	85,86	2	✗
19	01230	87,88,89	2	✗
20	00600	94,95	2	✗
21	00420	97,98,99	3	✗
22	00321	101,102	2	✗
23	00240	104,105	2	✗

As the density of the graph increases, there are more loops and fewer linear segments, so it is less likely for the graphs to be homeomorphic, hence the ticks are more common near the top of the table.

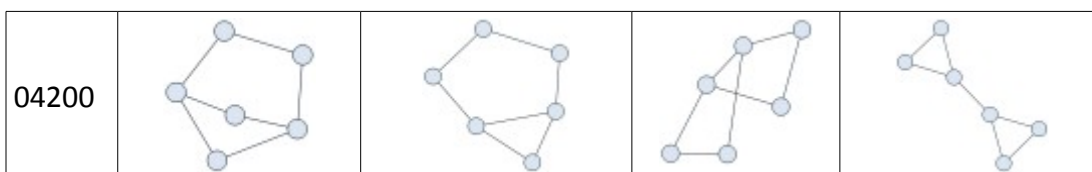
Here is an example of a class with 4 homeomorphic graphs that are all equivalent to two loops with a tail (not helped by *neato*'s layout of the last one with crossings):



Here is an example of a class with two graphs that are not homeomorphic. They both have two loops and two tails, but the tails are attached in topologically distinct ways (together and separate):



Many of the classes are mixed, with some homeomorphic graphs, but at least one that is not equivalent to the others. Here is an example with three homeomorphic variations of two adjacent loops (5+4, 5+3, 4+4), and a distinct topology for two separate loops (two triangles joined by a bar):



Unfortunately, the structure of two squares in the third diagram is obscured by *neato*'s crossed layout.

4. Conclusions

We have enumerated all the small graphs up to $n=6$, and displayed them with their degree histograms. The graphs for $n=2,3,4$ have distinct histograms and are unique up to isomorphism. The graphs for $n=5$ have two pairs of duplicate histograms, but we have shown these are topologically equivalent. The graphs for $n=6$ have 23 classes of repeated histograms (more than half of the graphs). Some of these classes contain homeomorphic graphs, but most have at least one graph that is topologically distinct.

Obviously there is a lot more that could be done to extend the code and the analysis. The first addition would be to automatically process contractions, search for the smallest contractions, and hence determine topological equivalence.

The existing program has many limitations, and there are many potential optimizations. Some of these are mentioned in Section 2.3

5. References

- [1] Online Encyclopedia of Integer Sequences [[OEIS A001349](#)]
- [2] GraphViz [[home](#)]
- [3] Scala [[home](#)]
- [4] Nauty [[home](#)]
- [5] Memoization Pattern [[wikipedia](#)]

Appendix A – Source Code

A.1 Core Class

```
type Node      = Int                // a node has an integer identifier
type Edge      = (Node,Node)        // an edge is a pair of nodes
type Morphism  = Map[Node,Node]     // a morphism is a mapping between nodes

class Graph( var id: Option[Long] = None ) { self =>

  val nodes: Set[Node] = new HashSet[Node] // a graph has a set of nodes
  val edges: Set[Edge] = new HashSet[Edge] // a graph has a set of edges

  def nNodes(): Int = nodes.size
  def nEdges(): Int = edges.size

  def isEmpty: Boolean = (nNodes == 0)

  def addNode( node: Node ): Graph = {
    nodes += node
    clearCache
  }

  def addEdge( n1: Node, n2: Node ): Graph = {
    nodes += n1 += n2
    edges += (if (n1<n2) (n1,n2) else (n2,n1)) // edges are sorted pairs
    clearCache
  }

  private var dhisto: Option[Array[Int]] = None // cache of degree histogram

  private def clearCache: Graph = { dhisto = None; this }

  def histogram(): Array[Int] = // memoization stores computed answer
    dhisto match {
      case Some(darr) => darr
      case None => { dhisto = Some(calculateHistogram); histogram }
    }

  def histogramCode(): String = histogram.mkString("")

  private def calculateHistogram(): Array[Int] =
    nodeDegrees.foldLeft( new Array[Int](nNodes-1) ){ // -1 ignores d=0
      case (arr,(node,d)) => arr(d-1) += 1; arr // -1 ignores d=0
    }

  private def nodeDegrees(): Map[Node,Int] = // map of node to degree value
    new HashMap[Node,Int] {
      for (n <- nodes) put( n, 0 )
      for ( (n1,n2) <- edges ) {
        put( n1, this(n1) + 1 )
        put( n2, this(n2) + 1 )
      }
    }
}
```

```

private def degrees(): List[Int] = // list the node degree values
  nodeDegrees.foldLeft( List[Int]() ){
    case (ds,(node,d)) => d :: ds
  }.sorted

def degreesCode(): String = degrees.mkString( "", ",", "" )

private def sameCounts( g: Graph ): Boolean = // quick nonisomorphic test
  (nNodes == g.nNodes) && (nEdges == g.nEdges) &&
  histogram.sameElements( g.histogram )

private def morph( m: Morphism ): Graph = // apply a morphism to a graph
  new Graph {
    for ( node <- self.nodes ) addNode( m(node) )
    for ((n1,n2) <- self.edges ) addEdge( m(n1), m(n2) )
  }

def isIsomorphism( m: Morphism, g: Graph ): Boolean =
  if (!sameCounts(g)) false else morph( m ).equals( g )

def isIsomorphic( gs: List[Graph] ): Boolean = gs.exists( g => isIsomorphic(g) )

def isIsomorphic( g: Graph ): Boolean =
  if (!sameCounts(g)) false
  else if (equals(g)) true
  else {
    val nodelist = nodes.toList
    for {
      perm <- nodelist.permutations.toList
      if !perm.equals( nodelist )
      if isIsomorphism( createMorphism(nodelist,perm), g )
    } return true
    false
  }

def isConnected() = if (nEdges < nNodes-1) false else (nComponents == 1)

def nComponents(): Int =
  edges.foldLeft( new MergeSets(nodes) )( (ms,e) => ms.merge(e) ).count

def generateId(): String = { // unique id for use in filenames
  "g" + nNodes + "-" + histogramCode +
  (this.id match { case Some(i) => "-" + i; case None => "" })
}

override def toString(): String = s"Graph $nodes $edges"

override def equals( any: Any ): Boolean =
  any match {
    case g: Graph => sameCounts(g) && nodes.equals(g.nodes) &&
      edges.equals(g.edges)
    case _ => false
  }
}

```

A.2 Utility Class

```
// a utility class holding a set of sets used to calculate connectedness
private class MergeSets( val nodes: Set[Node] ) {

    private var sets = new HashSet[Set[Node]] {
        for (node <- nodes) add( HashSet(node) )      // each node has its own set
    }

    def count(): Int = sets.size

    def merge( edge: Edge ): MergeSets = {
        val s1 = sets.find( s => s.contains(edge._1) ) match { case Some(s) => s }
        val s2 = sets.find( s => s.contains(edge._2) ) match { case Some(s) => s }
        if (s1 != s2) sets -= s1 -= s2 += (s1 ++ s2)
        this
    }
}
```

A.3 Construction Utilities

```
// build some useful test graphs

def createStar( n: Int ) = new Graph { for (i <- 2 to n) addEdge(1,i) }

def createLine( n: Int ) = new Graph { for (i <- 1 until n) addEdge(i,i+1) }

def createClique( n: Int ) =
    new Graph { for (i <- 1 until n; j <- (i+1) to n) addEdge(i,j) }

// main graph construction
def createGraph( n: Int, mask: Long ): Graph =
    new Graph( Some(mask) ) {
        for (i <- 1 to n) addNode( i )
        var k = 0
        for (i <- 1 until n; j <- (i+1) to n) {
            if (((mask >> k) & 0x1) == 0x1) addEdge(i,j)
            k += 1 } }

def createAllGraphs( n: Int ): List[Graph] =
    for (mask <- (0L until (1L << (n*(n-1)/2))) .toList) yield createGraph(n,mask)

def createConnectedGraphs( n: Int ): List[Graph] =
    createAllGraphs( n ).filter( g => g.isConnected )

def createUniqueGraphs( n: Int ): List[Graph] = {
    val ghead :: gtail = createConnectedGraphs( n )
    gtail.foldLeft( List(ghead) ){
        (gs,g) => if (g.isIsomorphic(gs)) gs else g::gs
    }.sortBy( _.histogramCode ).reverse
}
```

A.4 Output Utilities

```
def createAllTables =                                     // main entry point
  for (n <- 1 to 6) generateTable( s"g${n}table", createUniqueGraphs(n) )

def generateTable( title: String, graphs: List[Graph] ) = {
  for (g <- graphs) generateImage( g )
  generateHTML( title, graphs )
}

def generateImage( g: Graph ) = {
  val png = new File( "dot", g.generateId + ".png" )
  val dot = new File( "dot", g.generateId + ".dot" )
  toDot( new PrintWriter( dot ), g )
  val exitCode = Process( s"neato -Tpng -o${png} ${dot}" ).! // command line
  if (exitCode != 0) println( "Process exit code: " + exitCode )
}

def toDot( out: PrintWriter, g: Graph ) {                // GraphViz DOT format
  out.write( "graph G {\n" )
  out.write( "  size=\"0.8,0.8\";\n" )
  out.write( "  ratio=1;\n" )
  out.write( "  node [label=\"\",height=0.3,width=0.3,fixedsize=true,\n" )
  out.write( "    shape=circle,style=filled,\n" )
  out.write( "    fillcolor=\"#DCE6F2\",color=\"#142335\";]\n" )
  out.write( "  edge [style=solid,color=\"#142335\";]\n" )
  for (node <- g.nodes) out.write( s"n${node}; " )
  out.write( "\n" )
  for (edge <- g.edges) out.write( s"  n${edge._1} -- n${edge._2};\n" )
  out.write( "}" )
  out.close()
}

def generateHTML( filename: String, graphs: List[Graph] ) = {
  val html = new File( "dot", filename + ".html" )
  val out = new PrintWriter( html )
  out.write( "<html>\n<body>\n<table>\n<tr><th>Id</th>" )
  out.write( "<th>Graph</th><th>Degrees</th><th>Histogram</th></tr>\n" )
  var i: Int = 1
  for ( g <- graphs ) { toHTML( out, g, i ); i += 1 }
  out.write( "</table>\n</body>\n</html>" )
  out.close()
}

def toHTML( out: PrintWriter, g: Graph, i: Int ) {
  out.write( "<tr>\n" )
  out.write( "  <td>" + i + "</td>" )
  out.write( "  <td><img src=\"\" + g.generateId + ".png\"/></td>\n" )
  out.write( "  <td>" + g.degreesCode + "</td>\n" )
  out.write( "  <td>" + g.histogramCode + "</td>\n" )
  out.write( "</tr>\n" )
}
```