A CLASSIFICATION OF ALL CONNECTED GRAPHS ON SEVEN, EIGHT, AND NINE VERTICES WITH RESPECT TO THE PROPERTY OF INTRINSIC KNOTTING

_____

A Project

Presented

to the Faculty of

California State University, Chico

_____

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

_____

by

Christopher H. Morris

Fall 2008

A CLASSIFICATION OF ALL CONNECTED GRAPHS ON SEVEN, EIGHT, AND NINE VERTICES WITH RESPECT TO THE PROPERTY OF INTRINSIC KNOTTING

A Project

by

Christopher H. Morris

Fall 2008

APPROVED BY THE DEAN OF THE SCHOOL OF GRADUATE, INTERNATIONAL, AND INTERDISCPLINARY STUDIES:

_____
Susan E. Place, Ph.D.

APPROVED BY THE GRADUATE ADVISORY COMMITTEE:

_____
Abdel-Moaty Fayek, MS
Graduate Coordinator

_____
Tyson Henry, Ph.D., Chair

_____
Thomas Mattman, Ph.D.

ACKNOWLEDGEMENTS

I want to sincerely thank Dr. Tyson Henry and Dr. Thomas Mattman for their constant support, encouragement, and endless patience throughout this process. Thank you to Dr. Mattman for the idea behind this project and for his expertise in knot theory. Before talking with him, I had never heard of knot theory. Thank you to Dr. Henry for his expertise in computer science and for helping me to keep focused on my goal. In and out of the classroom, he has been an incredible teacher to me for many things technical and not so technical. If it were not for these two, I am confident that I would have given up on this long ago.

A special thank you to my wife, Michelle. If it were not for her, I would have never moved to Chico and consequently never pursued this degree. Thank you to her for her unconditional encouragement to complete my project, for putting up with my stress and perfectionism, and for humoring me by listening to me explain the technical aspects of my project which were completely foreign and uninteresting to her. I love you.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

ABSTRACT

A CLASSIFICATION OF ALL CONNECTED GRAPHS ON
SEVEN, EIGHT, AND NINE VERTICES WITH RESPECT
TO THE PROPERTY OF INTRINSIC KNOTTING

by

Christopher H. Morris

Master of Science in Computer Science

California State University, Chico

Fall 2008

Robertson and Seymour proved that there exists a finite set of minor minimal intrinsically knotted graphs, yet this set of graphs is unknown. The single goal of this project is to aid the mathematics community in advancing one step closer to determining this finite set of minor minimal intrinsically knotted graphs. The project achieves its goal in two distinct ways.

First, the project included the creation of an original, software-based toolset that is capable of classifying any graph into one of three distinct states—intrinsically knotted, *not* intrinsically knotted, or indeterminate. The tools use classification techniques that are based on the encoding of proved findings in the mathematics literature with regards to the property of intrinsic knotting. According to the literature search, this is the first example of such software-based tools.

The project offers a second contribution to the mathematics community, by applying this toolset to all the connected graphs on seven, eight, and nine vertices. As expected, the tools classified all of the seven and eight-vertex graphs as either intrinsically knotted or *not* intrinsically knotted, with the specific classifications matching previous results from the mathematics literature. The classification efforts of nine-vertex graphs, which have never been attempted before, revealed 32 indeterminate graphs. This set of 32 graphs likely includes new, previously undiscovered, minor minimal intrinsically knotted graphs. These 32 graphs are offered to the mathematics community as a starting point for the discovery of new minor minimal intrinsically knotted graphs.

It is with its original, software-based toolset for classifying graphs as intrinsically knotted and its presentation of 32 indeterminate graphs on nine vertices that this project hopes to aid the mathematics community in determining which graphs are in the finite set of minor minimal intrinsically knotted graphs.

CHAPTER I

INTRODUCTION

Overview

In 2004, mathematicians Robertson and Seymour proved that there exists a finite set of minor minimal intrinsically knotted graphs [1]. While this discovery proved a welcome addition to the fields of knot theory and graph theory, the pair left unanswered the questions of which graphs fall into this set or even more simply, how many graphs this set contains. Over the years, more graphs have been proven to belong to this finite set of graphs, but the same two questions remain open.

This project brings the mathematics community one step closer to determining which graphs fall into this finite set of minor minimal intrinsically knotted graphs. It does this in two distinct ways. First, this project presents an original software-based toolset which can classify any graph into one of three distinct categories—intrinsically knotted, *not* intrinsically knotted, or indeterminate. Traditionally, mathematicians classify graphs with respect to the property of intrinsic knotting by hand. They look for certain properties in the graph that can aid them in their classification efforts. Not only does this process potentially consume a lot of time, but also it is prone to error. While the toolset created cannot definitively classify all graphs as intrinsically knotted or *not* intrinsically knotted, as the indeterminate state remains, it does provide a first line of

testing for any new unclassified graph. Only if the toolset classifies a graph as indeterminate is extra work warranted.

As a direct result of creating a toolset to classify graphs with respect to the property of intrinsic knotting, this project applies those tools to all connected graphs on seven, eight, and nine vertices. This leads to the project's second main contribution. After iteratively evaluating all 273,050 connected graphs on seven, eight, and nine vertices, 32 graphs were classified as indeterminate. These 32 graphs, which are all on nine vertices, have an intrinsically knotted state that the toolset cannot determine. This means that within this set of graphs there may exist a new, previously undiscovered, minor minimal intrinsically knotted graph. Indeed, if even one of these is intrinsically knotted there will be a new minor minimal example among the 32 graphs. These graphs are a starting point to advancing one step closer to answering the question as to which graphs fall into the known finite set of minor minimal intrinsically knotted graphs that Robertson and Seymour left open.

This project combines the fields of knot theory and graph theory with the processing power of a computer to classify graphs with respect to the property of intrinsic knotting. The write-up that follows presents a more formal introduction to the problem domain, a review of relevant literature from within this domain, an in-depth description of the approach and algorithms created, followed by the results and an analysis of those results.

This first introductory chapter explains more clearly the problem domain. It defines knots and graphs and shows how the two relate. Furthermore, it describes the property of intrinsic knotting along with related terminology. The introduction concludes

with an explanation as to why the project focuses only on connected graphs on seven, eight, and nine vertices.

The second chapter reviews the relevant literature in the problem domain. It discusses the known intrinsically knotted graphs. It summarizes the literature that the toolset leverages to classify graphs as intrinsically knotted or *not* intrinsically knotted. The Background Chapter concludes with an exploration of algorithm designs along with valuable literature with respect to graphs.

The Methods Chapter outlines the high-level design decisions made for the toolset. It then explores in detail the two separate implementations of this high-level design—one in Java and one in Ruby—as well as why the two implementations exist at all. Finally, this chapter illustrates the execution of the software-based toolset.

The Results Chapter follows the Methods Chapter and describes in detail the results of running the toolset against all connected graphs on seven, eight, and nine vertices. It discusses the classifications of the graphs, the reason for those classifications, as well as some basic timing information. It will also diagram the 32 indeterminate graphs on nine vertices.

The Analysis Chapter follows the Results Chapter and analyzes the data presented in the results and compares them to the expectations, if any exist. In addition, it addresses the limitations.

Finally, the Conclusion Chapter summarizes the contributions of the entire project. It also looks forward by proposing future work that could leverage the foundation presented. The appendices that follow the Conclusion Chapter present a

detailed description of the toolset, formatted results, as well as all of the source code written.

What Is a Knot?

A knot is very likely what one imagines. If one visualizes holding an extension cord with an end in each hand, tangling the extension cord with itself, and finally plugging the ends in together, a knot results. The extension cord would be looped, and there would be no way, aside from cutting it, or unplugging the ends, to remove the tangling. Ultimately, this tangled and looped extension cord is a perfectly legitimate knot. Now, to a mathematician in the field of knot theory, this tangling could be classified, simplified and studied. In the classification and labeling of knots, there are a few very common ones. For example, Figure 1 illustrates the simplest knot, known as the 'unknot', which does not resemble a typical knot at all, but rather is nothing more than an untangled loop.

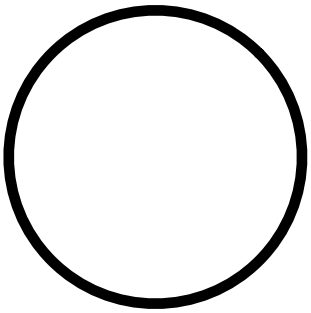


Figure 1. The Unknot

Another popular knot is the trefoil knot that illustrates a more complex tangling and can be seen in Figure 2.

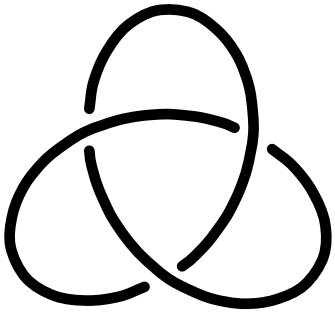


Figure 2. The Trefoil Knot

What Is a Graph?

A graph is simply a set of vertices and edges. One can visualize a vertex as a point or a dot. A graph on seven vertices has seven of these 'points'. An edge is a line that joins any two of the vertices. The line can be straight or curved, but it must be connected to a vertex at both ends. A graph will likely have multiple edges joining many different pairs of vertices, but a collection of vertices with no edges is also a valid graph.

This project only considers connected, undirected, unweighted graphs that do not contain any loops or double edges. Connected means that from any given vertex in the graph, one can reach any other vertex in the graph by traveling along some set of edges. The direction travelled does not matter since the edges are undirected. Unweighted means the edges have no value associated with them; they are simply lines connecting two vertices. Finally, no edge can exist from a vertex back to itself, and between any two vertices there is at most one edge. Figure 3 illustrates a connected, undirected, unweighted, five-vertex graph.

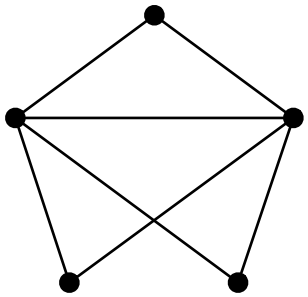


Figure 3. A Connected, Undirected, Unweighted, Five-Vertex Graph

How Do Knots and Graphs Relate?

A basic characteristic of a graph is the existence of one or more cycles. A cycle is a path in the graph beginning and ending with the same vertex that travels across edges and through any other vertex at most one time. A graph may contain zero or more cycles. One can visualize a cycle as a chain of connected edges in the graph that form a loop. Figure 4 illustrates a cycle that follows edges 0-1, 1-2, 2-4, and 4-0.

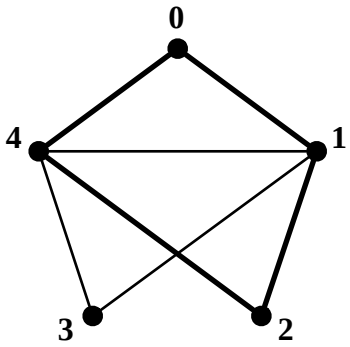


Figure 4. A Cycle Exists Using Edges 0-1, 1-2, 2-4, and 4-0

As described earlier, a knot can be visualized as a tangling in an extension cord that is plugged into itself. Just as the extension cord forms a loop, the cycle in a graph forms a loop; and, just as the loop in the extension cord can be knotted, so too can the cycle in the graph. Thus, if a graph contains a cycle that is knotted, the graph itself is

considered knotted. Figure 5 illustrates a non-trivially knotted graph as there exists a cycle which contains a knot, other than the unknot.

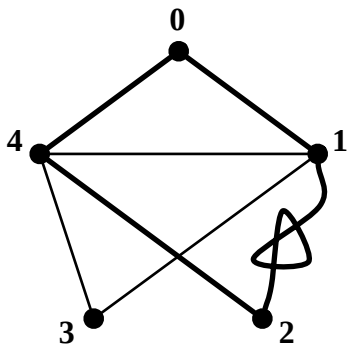


Figure 5. A Knotted Graph

What Is the Property of Intrinsic Knotting?

A realization of a graph in three-dimensional space is known as an embedding of the graph. No embedding for a graph is strictly associated with a graph, thus no embedding is more 'correct' than another. While there are an infinite number of embeddings for a single graph, a specific embedding simply serves as a convenient way of visualizing and communicating the graph's structure.

The cycles in graphs do not change as the embeddings change. A cycle in one embedding will always exist in every other embedding of that graph. As mentioned previously, it is possible for a cycle in a graph to be knotted, thus making the entire graph knotted. More precisely, it is the embedding of the graph that contains a knotted cycle, and thus the embedding of the graph that is considered knotted. If one were to embed the graph differently in space, the cycle that was once knotted may no longer be knotted. Figure 6 illustrates a different embedding of the same graph that previously contained a

knotted cycle in Figure 5. This new embedding does not have a knotted cycle and thus is not knotted.

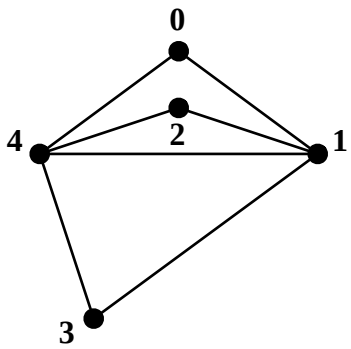


Figure 6. An Unknotted Embedding of Figure 5

A graph is defined to be intrinsically knotted if no matter how one embeds the graph in three-dimensional space, there will always exist at least one knotted cycle [2]. Different embeddings may yield different knotted cycles, but if a graph is intrinsically knotted there does not exist a single embedding, in the infinite number of possible embeddings, that does not contain a knotted cycle. When considering the property of intrinsic knotting, the embedding is irrelevant. Yet, if one embedding of a graph is found that does not contain a knotted cycle, then the entire graph is considered *not* intrinsically knotted. To determine that a particular embedding has no knotted cycles can be difficult. For example, some researchers believe that Figure 7 represents an unknotted embedding of a particular nine-vertex graph [3]. However, proving that there is no knotted cycle is no easy challenge.

An additional, relevant concept from graph theory is that of a graph minor. A graph minor is a graph G' which is created as a result of a series of vertex removals, edge removals, and/or edge contractions from a given graph G [4]. The graph G must have at

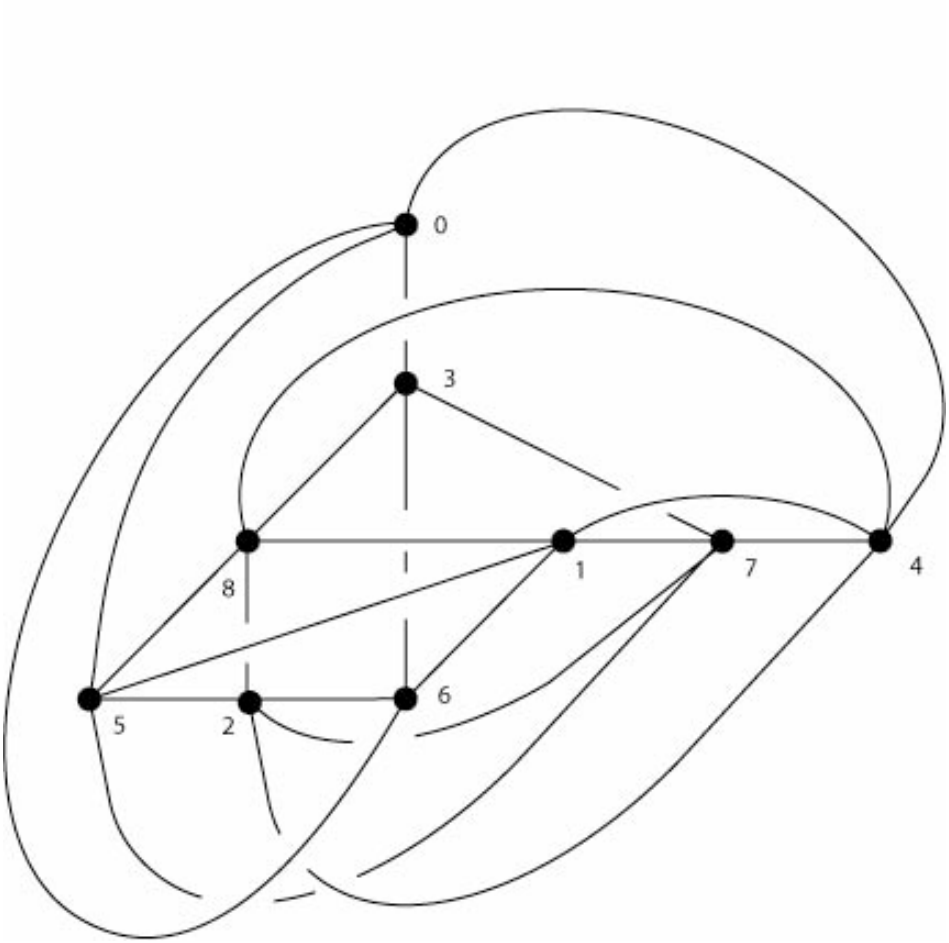
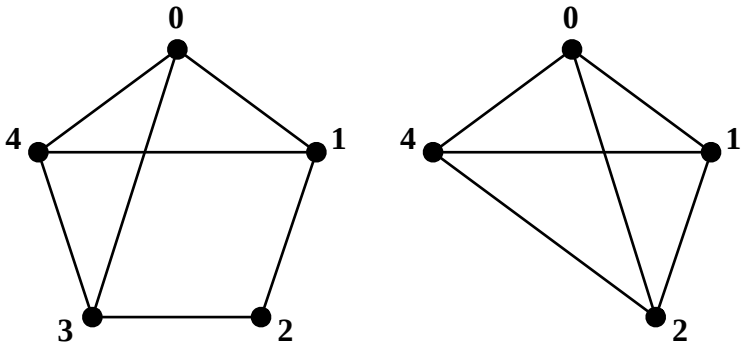


Figure 7. Is This Embedding Unknotted?

a minimum one of these actions performed on it in order to create a minor G'. In other words, G is not a minor of G. Figure 8 illustrates a graph on five vertices along with a minor of that graph which is the result of the contraction of edge 2-3.

The opposite of a minor is a graph expansion, which is created through a series of vertex additions, edge additions and/or vertex expansions. In Figure 8, graph G is an expansion of the graph G'.

An important concept that will often be used in conjunction with intrinsic knotting, is that of minor minimal with respect to the property of intrinsic knotting. A graph is minor minimal with respect to some property if the property is exhibited by the



(a) (b)

Figure 8. A Graph G and a Minor G' of G Resulting From the Contraction of Edge 2-3

graph but not by any of its minors. (For example, see Foisy's summary [4].) An intrinsically knotted graph is minor minimal if no minor of that graph is intrinsically knotted. In other words, any minor created from a minor minimal intrinsically knotted graph through vertex removals, edge removals and/or edge contractions will not exhibit the same intrinsically knotted property.

Why Graphs on Seven, Eight, and Nine Vertices?

This project focuses on graphs on seven, eight, and nine vertices for a few very simple reasons. To begin with, others have already classified the graphs on seven [5], [6] and eight [7], [8] vertices with respect to the property of intrinsic knotting. With this information available, if the software toolset were unable to classify all of the graphs on seven and eight vertices, it would be evidence that the encoded classification tests were incomplete. Furthermore, if any of the results from the toolset's classification attempts differ from those previously published, it would be evidence of inaccurate tests. This project includes the graphs on nine vertices in order to go beyond what is already known and proven. It ignores looking at graphs on six or fewer vertices because doing so

is trivial, as it is known that none are intrinsically knotted. Also, the project does not attempt to classify ten-vertex graphs because the current implementation would not complete in a reasonable amount of time as the number of connected graphs increases from just over 260,000 on nine vertices to nearly 12 million on ten vertices [9].

CHAPTER II

BACKGROUND

While reviewing literature for this project, the focus quickly became locating proved classification techniques with regards to intrinsic knotting. It is these proved techniques that provide validation of the set of classification tests in the toolset, which determine whether or not a graph is intrinsically knotted. Ultimately, each test that the tools apply to a graph has a corresponding documented proof that the test is an acceptable means for classification. Using these proved techniques requires information about which graphs are already known to be intrinsically knotted. As a result, the research also aimed to uncover these graphs.

Secondarily, the research included reviewing algorithms and techniques for proving isomorphism and planarity. It also revealed tools that could generate the graphs that were tested.

Known Intrinsically Knotted Graphs

The list of known intrinsically knotted graphs is a fundamental part of the research of the intrinsic knotting property. While over the years more minor minimal intrinsically knotted graphs have been identified, Robertson and Seymour proved that the list is finite [1]. One of the main focuses of this project is to help mathematicians advance one step closer to determining all of the graphs in this finite set.

When discussing the proved intrinsically knotted graphs, it is important to draw the distinction between which graphs have been proven to be minor minimal with respect to the property of intrinsic knotting and which have been proven to be simply intrinsically knotted. The set of minor minimal intrinsically knotted graphs is finite, while the list of intrinsically knotted graphs is infinite. Also, every intrinsically knotted graph either is or contains a minor minimal minor.

One of the first graphs proven to be intrinsically knotted was the complete graph on seven vertices, K_7, by Conway and Gordon [5]. Later, Kohara and Suzuki proved that in fact K_7 is minor minimal with respect to intrinsic knotting, which means that it is the only intrinsically knotted graph on seven or fewer vertices [6]. In this same publication, Kohara and Suzuki also proved that if the graph K_{3311} was intrinsically knotted, then it would be minor minimal with respect to that property [6]. It was Foisy who proved that K_{3311} was in fact intrinsically knotted [10] and thus minor minimal.

In their research, Motwani, Raghunathan, and Saran proved that if a graph G is intrinsically knotted and another graph G' is derived from G through a series of *triangle-Y exchanges*, then that graph G' is also intrinsically knotted [11]. The application of this research led to the identification of more intrinsically knotted graphs. Kohara and Suzuki applied the *triangle-Y exchanges* to K_7 and proved that all 13 graphs derived with these transformations were minor minimal with respect to intrinsic knotting [6]. They also showed that if K_{3311} was minor minimal with respect to intrinsic knotting, then all graphs obtained from *triangle-Y exchanges* on K_{3311} would also be minor minimal [6]. An unpublished work by Kohara, and later by Mattman, Ottman, and

Rodrigues, includes a diagram of the twenty-five graphs resulting from the *triangle-Y exchanges* of K_{3311} [12].

Still, mathematicians have discovered more intrinsically knotted graphs. In his work, Foisy proved that his graph *H* on 13 vertices is minor minimal with respect to intrinsic knotting [4]. Later, he proved that four other graphs, which he labeled G_{15}, H_{15}, J_{14}, and J'_{14}, are all intrinsically knotted [13]. He did not however prove that they were minor minimal, although he did conjecture that they were. Furthermore, he did not perform the *triangle-Y exchanges* on these graphs as he left that as an exercise for those interested.

While not revealing any new intrinsically knotted graphs Blain, Bowlin, Fleming et al. and Campbell, Mattman, Ottman et al. independently classified all intrinsically knotted graphs on eight vertices [7] [8].

Discovering Intrinsic Knotting

The research next focused on a single idea—collecting information that could help to classify a graph as intrinsically knotted or *not* intrinsically knotted. This research was encoded as a series of classification tests, which were then bundled together as a toolset for determining if a graph could be classified as intrinsically knotted or *not* intrinsically knotted. As previously stated, as a result of Kohara and Suzuki's proof that K_7 is minor minimal with respect to intrinsic knotting [6], based on the definition of minor minimal, it follows that K_7 is the only graph on seven vertices that is intrinsically knotted. Furthermore, it is the smallest graph that is intrinsically knotted, because any minor does not exhibit the intrinsically knotted property. For example, Mattman,

Ottman, and Rodriques state that a minimum of seven vertices is necessary in order for a graph to be classified as intrinsically knotted since K_7 was proven to be minor minimal with respect to intrinsic knotting [12]. Thus, the toolset can test for the order of the graph, that is, the number of vertices, knowing that any graph with less than seven vertices is *not* intrinsically knotted.

In their work, Robertson, Seymour, and Thomas showed that a graph is intrinsically linked if and only if it contains one of the seven Petersen graphs as a minor, and a graph that is intrinsically knotted is also intrinsically linked [14]. Since each of the Petersen graphs has 15 edges, it follows that all intrinsically knotted graphs have at least 15 edges. This discovery allows the toolset to classify any graph with less than 15 edges as *not* intrinsically knotted.

Campbell, Mattman, Ottman et al. proved that a graph G with n vertices, where n is greater than or equal to seven, was guaranteed to be intrinsically knotted if it had $5n-14$ or more edges because they observed that such a graph would contain K_7 as a minor [8]. In their work, they deduced that these graphs were intrinsically knotted by leveraging Mader's proof that a graph with $5n-14$ or more edges has a K_7 minor [15]. This presents the simple statement that a graph that has $5n-14$ or more edges where n represents the number of vertices is intrinsically knotted.

Nešetřil and Thomas proved the fact that if a graph G contains as a minor, a known intrinsically knotted graph, then that graph G is also intrinsically knotted [16]. This fact allows the toolset to use all of the known intrinsically knotted graphs that were outlined in the earlier section. Since there is a known set of intrinsically knotted graphs,

the toolset can test to see if any of those graphs is a minor of a given graph *G*. If so, then the graph in question is also intrinsically knotted.

Simply applying the definition of minor minimal, leads to a test similar to the previous one. The toolset can test to see if the graph *G* is a minor of any known minor minimal intrinsically knotted graph. If it determines that G is a minor, then that is proof that G is *not* intrinsically knotted because the graph it is a minor of was minor minimal with respect to the intrinsically knotted property.

A final, yet very powerful piece of research proved by Blain, Bowlin, Fleming et al. states that if from a given graph *G*, two vertices are removed (with all of the adjoining edges), and if the resulting graph with two less vertices is planar, then the original graph *G* is definitely *not* intrinsically knotted [7]. A graph is planar if it can be embedded in a two-dimensional plane without any edges crossing each other. The reason this is true is because if one can embed a graph in a plane, then adding back the two removed vertices by placing one below the plane and one above the plane, will not introduce any cross edges. Without cross edges, there can be no knot. Thus, the toolset can use this research and take a given graph *G* and remove pairs of vertices, looking for a planar subgraph. If it discovers one planar subgraph, then it can conclude that the original graph is definitely *not* intrinsically knotted.

Algorithms

Based on the tests that the toolset performs on graphs, a couple of key algorithms are needed. One key algorithm is a test for graph isomorphism. This means that given two graphs, *G* and *H*, even though their vertices may be labeled differently, are

they essentially the same graph? This is of particular importance when trying to determine if a graph contains a given minor. The research included the review of two published algorithms for the Java implementation of a graph isomorphism test, one by Faizullin and Prolubnikov [17], and the other by McKay [18]. Neither solution was chosen. Instead, an original, less efficient, yet simpler, algorithm was designed to solve this problem in Java. To avoid the riskiness of an original algorithm in such a key position, the Ruby implementation harnesses the power of McKay's library of tools known as *nauty*, which can test isomorphism directly [19].

A similarly important algorithm is one that can determine if a given graph is planar or not. The Java implementation of the toolset ultimately reduces this algorithm to an isomorphism test between two graphs. While this is fast enough, the research by Hopcroft and Tarjan that proved a linear time algorithm for planarity testing could have been used [20]. Again, due to the riskiness of this implementation, the Ruby version of the toolset instead invokes one of McKay's tools for testing planarity [19].

Graphs

If there were no known set of graphs with seven, eight, and nine vertices, this project could not be continued, as generating close to 300,000 graphs would be a huge project in itself. Fortunately, in McKay's *nauty* suite of tools, there is a tool that can generate every connected graph on *n* vertices [19]. This tool is used as the source of all of the graphs tested. Furthermore, his chart of the distribution of graphs [9] aided in the decision to limit the project to graphs on nine vertices and not test ten vertices because the number of graphs increases by a factor of nearly fifty.

CHAPTER III

METHODS

The project is centered on a software-based toolset created to classify all of the connected graphs on seven, eight, and nine vertices with respect to the property of intrinsic knotting. After much work, two similar classification tools resulted. One tool was implemented in Java, the other in Ruby. While the tools were implemented in different languages, the logic for both tools is nearly identical. Both tools take a brute force approach to classifying all of the graphs, processing one graph at a time. Both tools apply the same series of classification tests to each graph, in the same order. The only substantial, logical difference between the tools is their implementation of the minor detection and planarity algorithms. The following chapter details the design, implementation, and execution of these tools. It also discusses the rationale for the two similar tools. Because the implementation languages differ between the classifier tools, all logic is expressed in a language neutral pseudo-code.

Design

Main Algorithm

Both the Java and the Ruby implementation of the classification tool follow the same high-level design. Both implementations read a graph from some source, attempt to classify that graph by applying a series of classification tests to it, output the

result, and proceed with the next graph. When the tools apply the series of classification tests to the current graph, they only try the next test if the current test yielded an indeterminate result. If all classification tests yield an indeterminate result, then the result of the classification for the graph is indeterminate and the tools move on to the next graph. In the following illustration of the logic for this algorithm, intrinsically knotted is referred to as IK.

```
for each graph in the set of graphs to test
  for each test in the set of tests
    apply classification test to graph
    done if graph is IK or not IK
  end

  graph is indeterminate
end
```

Graph Data Structure

It was crucial for this project to contain a comprehensive data structure that encapsulated the properties of a graph. This data structure represents a graph in memory using a common strategy known as an adjacency matrix. This is nothing more than a two-dimensional array, with each element representing the existence or absence of an edge between two vertices. The data structure includes basic access methods to get the order and size of the graph as well as the edges in the graph. It has methods to remove vertices and edges as well as contract edges. In addition, the data structure includes numerous other, trivial methods.

Perhaps the most important aspect of the graph data structure is the method that determines if one graph is a minor of another graph. While the high-level logic in this method is similar in both the Java and the Ruby designs, the methods that it depends

on, differ. In both designs, the method which determines if the graph G is a minor of the graph G' has the following general logic. (Recall that the order of a graph is the number of vertices, while the size is the number of edges.)

```
if order of G > order or G'
  return is not a minor
end

if size of G > size of G'
  return is not a minor
end

if G is a subgraph of G'
  return is a minor
end

for each edge in G'
  create G" by contracting the edge in G'

  if G is a minor of G"
    return is a minor
  end
end

return is not a minor
```

The designs diverge in the method which determines if the graph G is a subgraph of the graph G'. In the Java design, subgraph detection is determined by trying to find a mapping from the vertices in the potential subgraph G to the vertices in the larger graph G'. All of the possible permutations are attempted, and tested to see if they represent a legitimate mapping of the vertices in one graph to the vertices in the other graph. A legitimate mapping means that for each edge in G, there exists a corresponding edge in G', after the mapping from G to G' is applied to the vertices. If a complete mapping of vertices is detected, then it can be concluded that the graph G is in fact a

subgraph of the graph G'. The following illustrates this high-level design in the Java version where G is the graph that is being tested to see if it is a subgraph of the graph G'.

```
if order of G > order or G'
  return is not a subgraph
end

if size of graph G > size of graph G'
  return is not a subgraph
end

for each permutation of set of n vertices
         from G' where n is order of G
  if the permutation is a valid mapping of
     vertices from G to G'
    return is a subgraph
  end
end

return is not a subgraph
```

In the Ruby design, the ability to check for isomorphism is leveraged in the subgraph detection algorithm. In this algorithm, all combinations of edge and vertex removals are performed on graph G' to create G". The algorithm tests the resulting graph G" to see if it is isomorphic to the original graph G. If the graphs are isomorphic, it concludes that the graph G is a subgraph of the graph G'. The following illustrates the design of this logic for the Ruby version.

```
if order of G > order or G'
  return is not a subgraph
end

if size of graph G > size of graph G'
  return is not a subgraph
end

if order of G < order of G'
  for each combination of set of n vertices
           from G' where n is (order of G' -
```

```
            order of G)
        create G″ by removing current set of
            vertices from G′

        if G is subgraph of G″
          return is a subgraph
        end
      end
    else if size of G < size of G′
      for each combination of set of n edges
            from G′ where n is (size of G′ -
            size of G)
        create G″ by removing current set of
            edges from G′

        if G is subgraph of G″
          return is a subgraph
        end
      end
    else
      if G is isomorphic to G′
        return is a subgraph
      else
        return is not a subgraph
      end
    end

    return is not a subgraph
```

The reason the Ruby design of this subgraph detection algorithm leverages the isomorphism method is because the isomorphic method is trivial in Ruby. In the Ruby design, the isomorphic test is performed by comparing the canonical labeling of each graph to each other. If the canonical labelings of the graphs are identical, then the graphs are isomorphic. If they are not identical, then the graphs are not isomorphic. The canonical labeling is generated from the *labelg* tool in the *nauty* suite of tools.

Another integral algorithm in the graph data structure that differs between the Java and the Ruby designs is the algorithm that determines if a graph is planar or not. The Java design leverages a fact about planarity that states that a graph is non-planar if

and only if it contains either the graph K_5 or the graph K_{33} as a minor. Since the Java design already has a minor detection algorithm, this algorithm can be used to determine if K_5 or K_{33} are contained as minors of the graph in question. The algorithm also uses another known fact about planar graphs that states that a graph is not planar if it has three or more vertices and the size of the graph is greater than three times the order less six, as that is the number of edges in a triangulation. The following illustrates the design for this method in the Java version. (The Java implementation of this logic, which can be found in Appendix E, is located in the planarity classification rather than the graph class due to a language limitation.)

```
if order of graph ≥ 3 and
   size of graph > 3 x order of graph - 6
  return is not planar
end

if K5 is a minor of graph or
   K33 is a minor of graph
  return is not planar
end

return is planar
```

The Ruby design for this same algorithm leverages the *planarg* tool, which comes bundled with the *nauty* suite of tools. Instead of looking for the minors K_5 or K_{33}, the algorithm simply invokes the tool. The following illustrates the algorithm design for the Ruby version.

```
if order of graph ≥ 3 and
   size of graph > 3 x order of graph - 6
  return is not planar
end

return result from planarg tool
```

The remaining algorithms and details in the graph data structure are rather trivial and not worthy of discussion. For a look at the source code to the Java and Ruby implementations of this data structure, see Appendix E and Appendix F respectively.

Classification Tests

The ability of the tools to classify a single graph with respect to the property of intrinsic knotting relies on the existence of a series of classification tests. These classification tests accept a graph as a parameter, perform some operations on that graph, and return one of three classification results—intrinsically knotted, *not* intrinsically knotted, or indeterminate. While each test is rather simplistic by itself, the collection of all of the tests proves to be quite powerful in classifying most graphs. The following section describes each classification test along with the research that justifies its inclusion. As mentioned, while the implementations in Java and Ruby differ slightly with respect to these classification tests, the logic in both implementations is functionally identical.

Order Classification

Even though this project only tests graphs of seven or more vertices, it includes this test for completeness. Kohara and Suzuki proved that a graph that has six or fewer vertices is *not* intrinsically knotted [6]. As a result, this classification test simply tests the number of vertices in the graph, and if the number is less than or equal to six, determines that the graph is *not* intrinsically knotted. If the order of the graph is greater than six, this classification test cannot reach a definitive result, and thus returns an indeterminate result. The logic for this classification is as follows:

```
if order of graph ≤ 6
  return not intrinsically knotted
else
  return indeterminate
end
```

Absolute Size Classification

Similar to the Order Classification, the Absolute Size Classification looks at the number of edges in the graph, which is also known as the size of the graph. This classification is an encoding of Robertson, Seymour, and Thomas' work that showed that a graph with less than fifteen edges is *not* intrinsically knotted [14]. A graph with fifteen edges or more cannot be determined by this classification test. The logic is illustrated below.

```
if size of graph < 15
  return not intrinsically knotted
else
  return indeterminate
end
```

Relative Size Classification

The Relative Size Classification is an encoding of the results proved by Campbell, Mattman, Ottman et al. that a graph on n vertices is guaranteed to be intrinsically knotted if it has $5n-14$ or more edges, where n is greater than or equal to seven [8]. As a result, this classification checks to see if there are $5n-14$ edges or more in the graph. If there are, then it determines that the graph is intrinsically knotted, otherwise the result cannot be determined by this classification. The logic for this classification test is as follows:

```
if graph order ≥ 7 and
   graph size ≥ ((5 x graph order) – 14)
  return intrinsically knotted
else
  return indeterminate
end
```

Minor Of Classification

This classification test is a bit more robust than the previous few. To begin, it relies on the encoding of all of the known minor minimal intrinsically knotted graphs. An instance of this classification is initialized for each one of these known minor minimal intrinsically knotted graphs. Each instance determines if the graph it is testing is a minor of the known minor minimal intrinsically knotted graph with which it was initialized. If the tested graph is a minor of a known minor minimal intrinsically knotted graph, then the classification concludes that the graph is *not* intrinsically knotted, otherwise the result is indeterminate. This classification is supported by the definition of minor minimal, which states that the property is exhibited by the graph, but not by any of its minors. (For example, see [4].) The following illustration assumes that the classification was initialized with a single known minor minimal intrinsically knotted graph (MMIK).

```
if graph is a minor of the known MMIK graph
  return not intrinsically knotted
else
  return indeterminate
end
```

Contains Minor Classification

Similar to the Minor Of Classification, the Contains Minor Classification leverages previously encoded graphs. The graphs it uses are any known intrinsically knotted graphs, not simply the ones that are minor minimal with respect to the property of

intrinsic knotting. An instance of this classification is created for each one of these known graphs. Each instance of the Contains Minor Classification tests to see if the given graph is isomorphic to, or contains as a minor, the known intrinsically knotted graph, with which it was initialized. If the graph is isomorphic to, or contains as a minor the intrinsically knotted graph, then it concludes that the graph that is being tested is in fact intrinsically knotted, otherwise the result cannot be determined by this classification. Nešetřil and Thomas provide the validation for this classification, for they proved the fact that if a graph *G* contains as a minor, a known intrinsically knotted graph, then that graph *G* is also intrinsically knotted [16]. The following illustration assumes that the classification was initialized with a single known intrinsically knotted graph (IK).

```
if graph is isomorphic to known IK graph or
   graph contains as a minor known IK graph
  return intrinsically knotted
else
  return indeterminate
end
```

Planarity Classification

Blain, Bowlin, Fleming et al. proved that if a graph is formed from a planar graph plus two vertices, then the graph is *not* intrinsically knotted [7]. This research leads to the final classification test—the Planarity Classification. This classification tests if after removing any two vertices from the given graph a planar graph remains. If it finds one set of two vertices where this is true, then it concludes that an embedding of the graph exists without any knots present, and therefore it is *not* intrinsically knotted. If this test cannot find a planar subgraph after removing every possible combination of two vertices from the original graph, then it concludes that it cannot determine a result. The

actual test for planarity is described in the discussion of the Graph data structure; this description simply assumes that the Graph data structure has the ability to determine if the graph it represents is planar or not. The following illustrates the logic for the Planarity Classification. For simplicity, it assumes that the graph that is being tested is called *G*.

```
for each combination of two vertices in G
  create G' by removing current pair from G

  if G' is planar
    return not intrinsically knotted
  end
end

return indeterminate
```

Implementation

Java Versus Ruby

As has been mentioned, the toolset created for this project contains two tools that can classify graphs with respect to the property of intrinsic knotting. One tool, *java_ik_classifier*, is implemented in Java, while the other, *ik_classifier*, is implemented in Ruby. The reasons for two tools, which perform the same basic functionality, are rather simple.

When the project began, the Java tool was the first implementation of a tool to classify graphs with respect to the property of intrinsic knotting. The tools was rather efficient, and seemingly accurate, but it became very obvious that the original algorithms for minor detection and planarity were by far the riskiest parts of the codebase. Most everything in the codebase was rather simplistic, with the exception of these two

algorithms. Realizing this, it was decided that leveraging McKay's *nauty* suite of tools would be a good way to mitigate some of the risk. The planarity detection algorithm proved to be easily replaceable by McKay's tool, *planarg*, which can determine if a given graph is planar or not. Although McKay does not offer a tool for minor detection, he does offer an easy way to determine if two graphs are isomorphic to each other by comparing their canonical labeling. Fortunately, it is simplistic to reduce the minor detection algorithm to a series of isomorphic tests. Thus, the minor detection algorithm was converted to utilize McKay's canonical labeling tool, *labelg*.

After changing the implementations of the minor detection and planarity algorithms to leverage McKay's tools, efficiency became a huge problem in Java. Making a single call to a Unix utility from Java consumed nearly one tenth of a second. While not slow by itself, millions of calls out to Unix proved to be a huge bottleneck. This inefficiency led to the second implementation of the toolset in the Ruby language. In Ruby, a call out to Unix took less than a millisecond. Even though Ruby is slower, in general, than Java, the efficiency gain in making calls to the shell proved worthy of the transition. In short, the main driving factor to the Ruby implementation was more efficient calls to the Unix shell than could be obtained in Java.

A secondary benefit of moving to Ruby was simply that the language is much more concise and simpler to read. This allows the code to be very close to the definitions in the problem domain. As an example, the minor detection algorithm in Ruby reads very clearly as the definition of a graph minor. This not only makes the programming easier, but it allows one to read the code easily and learn from it. While this is certainly possible in Java, the Ruby programming language makes this much easier.

With two separate implementations of the same tool, there were considerations of abandoning the original Java tool. It was decided that keeping the Java tool was actually valuable because it allowed for two independent classifications of a set of graphs, and thus the results from both classifications could be compared for correctness. With the exception of timing, the results of both tools must be identical, which they are. Thus, both tools remained in the final implementation. Appendix E gives the Java source code, while the Ruby source code is in Appendix F.

Statically Encoded Graphs

As mentioned, a handful of graphs were encoded for use with the Minor Of and Contains Minor Classifications. The encoding of these graphs was done by hand from printed diagrams. To ensure correctness, each graph was independently encoded a minimum of three times, and the three encodings compared with one another. The encodings are in Appendix E and Appendix F with the source code.

Classification Test Order

While efficiency was not the primary emphasis, some thought was put into the ordering of the classification tests. The goal of the ordering was to do the fastest classification tests first, hoping that a result could be found, before trying the more complex classifications. As a result, the final ordering of the classifications was as follows—Order, Absolute Size, Relative Size, Planarity, Contains Minor K_7, Contains Minor H_8, Contains Minor H_9, Contains Minor F_9, Contains Minor K_{3311}, Contains Minor A_9, Contains Minor B_9, Minor Of K_7, Minor Of H_8, Minor Of H_9, Minor Of F_9, Minor Of K_{3311}, Minor Of A_9, Minor Of B_9.

The Intrinsic Knotting Toolset

This project includes an original suite of tools, which proved useful during its development. All of these tools are found in the *tools* directory of the *ik_toolset* package.

The first tool is the *installer*. This tool prepares the environment, making it easy to get started with the toolset. It downloads the *nauty* suite of tools, compiles them, and moves the binaries into the *nauty* directory for use by the other tools. It also compiles the Java source and builds a distribution *jar* file. Furthermore, the *installer* leverages the *graph_generator* tool and generates files for all of the connected graphs on one through nine vertices in the *graphs* directory. It creates the *docs* directory and generates the Java documentation as well as the Ruby documentation. Finally, the *results* directory is made as a location to store results from classification runs.

As mentioned, the *graph_generator* tool is capable of creating files with all of the connected graphs on the provided number of vertices. This tool is used as the source for all graphs used in this project.

A tool that complements the *graph_generator* is the *graph_finder* tool. This tool is able to look at the file generated by the *graph_generator* and select supplied lists of graphs from it. If one wants to know what certain graphs look like, this is the tool to use.

The *graph_complementor* tool is identical to the *graph_finder* tool except that it will show the complement of the graphs of interest instead of the graph itself. The complement is included because with large, dense graphs, it is often easier to look at the edges that do not exist than analyzing those that do exist.

There are two tools in the toolset that are capable of classifying a given graph into one of three states—intrinsically knotted, *not* intrinsically knotted, or indeterminate. These tools are the *java_ik_classifier* and the *ik_classifier*. The *java_ik_classifier* tool is written in Java, while the *ik_classifier* tool is written in Ruby. Both tools have similar interfaces, allowing for a graph file as input and a destination file for the output.

The *ik_summarizer* tool summarizes the output of the *java_ik_classifier* and *ik_classifier* tools. It determines how many graphs the tools processed, how they were classified, why they were classified the way they were, as well as timing related information. This tool was integral in preparing the Results Chapter.

The final tool in the toolset is the *expansion_mapper* tool. This tool takes a list of graphs as an argument and generates a mapping of each graph to each other. The relationships between the graphs in this mapping are an expansion/minor relationship. For example, if graph one is shown as related to graph three, it means that graph three is an expansion of graph one, and graph one is a minor of graph three. This tool proved very useful in the Analysis Chapter when looking at the 32 indeterminate graphs on nine vertices.

Appendix A gives further details for each of these tools, such as usage information.

Execution

Upon the completion of the implementation of the toolset, it was time to execute the tools against all of the connected graphs on seven, eight, and nine vertices.

Before detailing the execution steps, it is valuable to know the project's execution environment. Table 1 details the hardware and operating system used.

Table 1. Hardware and Operating System Execution Environment

Computer Model	Apple iMac (iMac8,1)
RAM	4 GB 800 MHz DDR2 SDRAM
CPU	2.8 GHz Intel Core 2 Duo
Level 2 Cache	6 MB
Bus Speed	1.07 GHz
Operating System	Mac OS 10.5.4

In addition to the hardware, the details of the dependent software are equally important. Table 2 details the software tools, their versions and any installation notes that the toolset utilized.

Table 2. Software Execution Environment

Tool	Version	Installation Notes
ant	1.7.0	bundled with MacOS
dretog	2.4 Beta 7	bundled with nauty
gcc	4.0.1	bundled with MacOS
geng	2.4 Beta 7	bundled with nauty
jar	1.5.0_13	bundled with MacOS
java	1.5.0_13	bundled with MacOS
javac	1.5.0_13	bundled with MacOS
javadoc	1.5.0_13	bundled with MacOS
labelg	2.4 Beta 7	bundled with nauty
make	3.8.1	bundled with MacOS
planarg	2.4 Beta 7	bundled with nauty
rdoc	1.0.1	installed with macports
ruby	1.8.6	installed with macports
showg	2.4 Beta 7	bundled with nauty

When it came to finally utilizing the toolset, all commands were executed in the Bash Shell from the *ik_toolset* directory. First, the toolset needed to be properly installed. This process downloaded and compiled the *nauty* tools. It also built the Java packages, documentation, and generated all of the necessary graphs. The following command initiated this process.

```
tools/installer
```

Next, the Java classifier was run against the connected graphs on seven, eight, and nine vertices, which were created by the *installer*. The following commands performed these classifications.

```
tools/java_ik_classifier \
  –f graphs/connected_graphs_7.txt \
  –o results/java_7.txt

tools/java_ik_classifier \
  –f graphs/connected_graphs_8.txt \
  –o results/java_8.txt

tools/java_ik_classifier \
  –f graphs/connected_graphs_9.txt \
  –o results/java_9.txt
```

Upon the completion of the Java classifications, the same classifications were repeated using the Ruby classification tool. The following commands performed the Ruby classifications.

```
tools/ik_classifier \
  –f graphs/connected_graphs_7.txt \
  –o results/ruby_7.txt

tools/ik_classifier \
  –f graphs/connected_graphs_8.txt \
  –o results/ruby_8.txt
```

```
tools/ik_classifier \
  –f graphs/connected_graphs_9.txt \
  –o results/ruby_9.txt
```

Finally, when all of the classification runs had completed, the *ik_summarizer* tool was used to summarize the classification results. The following illustrates the use of this tool.

```
tools/ik_summarizer \
  -f results/java_7.txt \
  -o results/java_7_summarized.txt

tools/ik_summarizer \
  -f results/java_8.txt \
  -o results/java_8_summarized.txt

tools/ik_summarizer \
  -f results/java_9.txt \
  -o results/java_9_summarized.txt

tools/ik_summarizer \
  -f results/ruby_7.txt \
  -o results/ruby_7_summarized.txt

tools/ik_summarizer \
  -f results/ruby_8.txt \
  -o results/ruby_8_summarized.txt

tools/ik_summarizer \
  -f results/ruby_9.txt \
  -o results/ruby_9_summarized.txt
```

The Results Chapter details the summaries from the classification runs.

CHAPTER IV

RESULTS

As previously mentioned, upon the completion of the implementation of the tools, the project included the running of the *java_ik_classifier* and *ik_classifier* programs against all connected graphs on seven, eight and nine vertices. The *ik_summarizer* tool then summarized the raw results into a simple presentation of statistics along with a list of any graphs that could not be classified with respect to the property of intrinsic knotting. Both the Java and the Ruby versions of the classification tool yielded identical results when it came to classifications, although the two versions did differ when it came to timing. While the summarized results from the *ik_summarizer* tool can be found in Appendix B, the following is a description of those results.

Seven-Vertex Graphs

The classification results for seven-vertex graphs were the same using both the Java and the Ruby tools. Each tool processed all 853 distinct graphs. The tools classified 852 of those graphs, or 99.88% of the total, as *not* intrinsically knotted, and one graph, or 0.12% of the total, as intrinsically knotted. The one intrinsically knotted graph, graph id 853, is the complete graph on seven vertices, which is also known as K_7. Of the 853 graphs tested, zero had an intrinsically knotted state that could not be determined.

Since the tools classified all of the 853 graphs as either intrinsically knotted or *not* intrinsically knotted, one of the classification tests was responsible for each graph's classification. Of the graphs that were *not* intrinsically knotted, the Absolute Size Classification classified 773 of them, or 90.62%. The Planarity Classification classified the remaining 79 non-intrinsically knotted graphs, 9.26% of the total. The Relative Size Classification classified the one intrinsically knotted graph, 0.12% of the total. This distribution is illustrated in Figure 9.

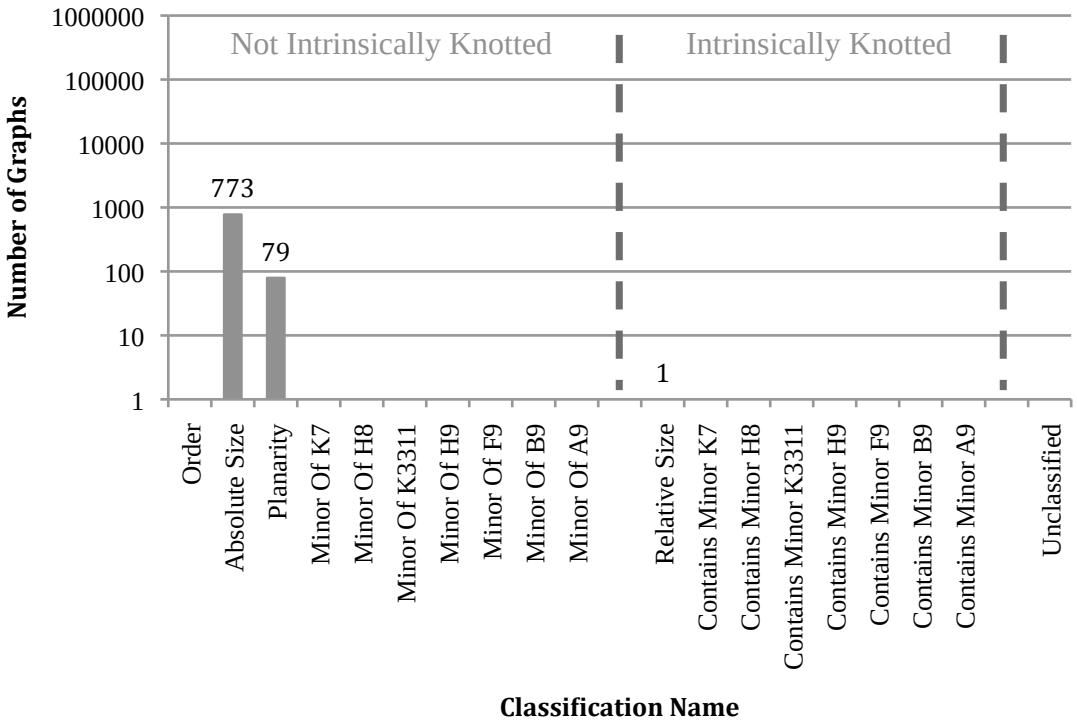


Figure 9. Classification Tests for Seven-Vertex Graphs

The results of the Java and Ruby classification runs on seven-vertex graphs differed when it came to timing. The total running time for the Java version was 79 milliseconds while the Ruby version completed in 505 milliseconds. This time encompasses everything from start to finish which not only includes the actual time spent

processing each graph, but also overhead tasks such as opening files, parsing files, as well as writing and formatting output. The total time spent actually classifying the graphs was five milliseconds in the Java version and 388 milliseconds in Ruby.

The tools also collected per graph processing time. Of the 853 graphs processed, the mean time spent on a single graph was less than one millisecond in both the Java and Ruby versions. Similarly, the median per graph time was also less than one millisecond in both versions. In the Java version, the fastest graph was processed in less than one millisecond, while the slowest graph took one millisecond. The Ruby version also had the fastest graph completing in less than one millisecond, but the slowest graph took six milliseconds to complete. Figure 10 illustrates the per graph time differences between the Java and Ruby versions.

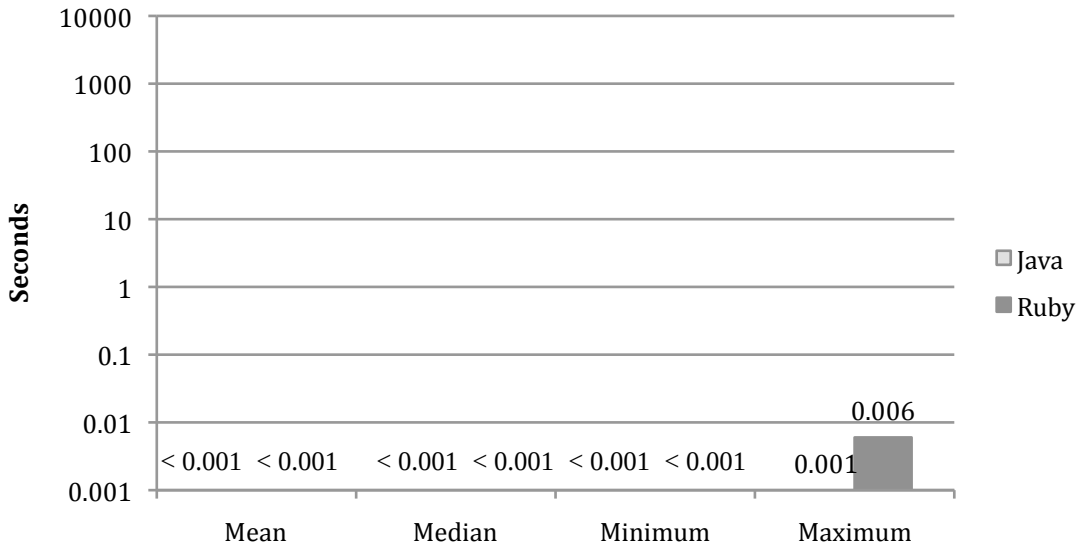


Figure 10. Per Graph Classification Times for Seven-Vertex Graphs

All 853 graphs on seven vertices were classified as either intrinsically knotted or *not* intrinsically knotted; thus, there are no indeterminate seven-vertex graphs.

Eight-Vertex Graphs

The classification results for eight-vertex graphs were the same using both the Java and the Ruby tools. Each tool processed all 11,117 distinct graphs. The tools classified 11,095 of those graphs, or 99.80% of the total, as *not* intrinsically knotted, and 22 graphs, or 0.20% of the total, as intrinsically knotted. Of the 11,117 graphs tested, zero had an intrinsically knotted state that the tools could not determine.

Since the tools classified all of the 11,117 graphs as either intrinsically knotted or *not* intrinsically knotted, one of the tests was responsible for each graph's classification. Of the graphs that were *not* intrinsically knotted, the Absolute Size Classification classified 5,850, or 52.62% of the total. The Planarity Classification classified the remaining 5,245 non-intrinsically knotted graphs, 47.18% of the total. Of the 22 graphs classified as intrinsically knotted, 12, 0.11% of the total, were classified by the Contains Minor K_7 Classification. The Contains Minor H_8 classified five graphs, 0.04% of the total, while the Relative Size Classification classified four graphs, 0.04% of the total. The Contains Minor K_{3311} Classification classified the remaining one graph, 0.01%, as intrinsically knotted. This distribution is illustrated in Figure 11.

The results of the Java and Ruby classification runs on eight-vertex graphs differed when it came to timing. The total running time for the Java version was 1.916 seconds, while the Ruby version completed in 36.151 seconds. This time encompasses everything from start to finish which not only includes the actual time spent processing each graph, but also overhead tasks such as opening files, parsing files, as well as writing and formatting output. The total time spent classifying the graphs was 1.633 seconds in the Java version and 33.887 seconds in Ruby.

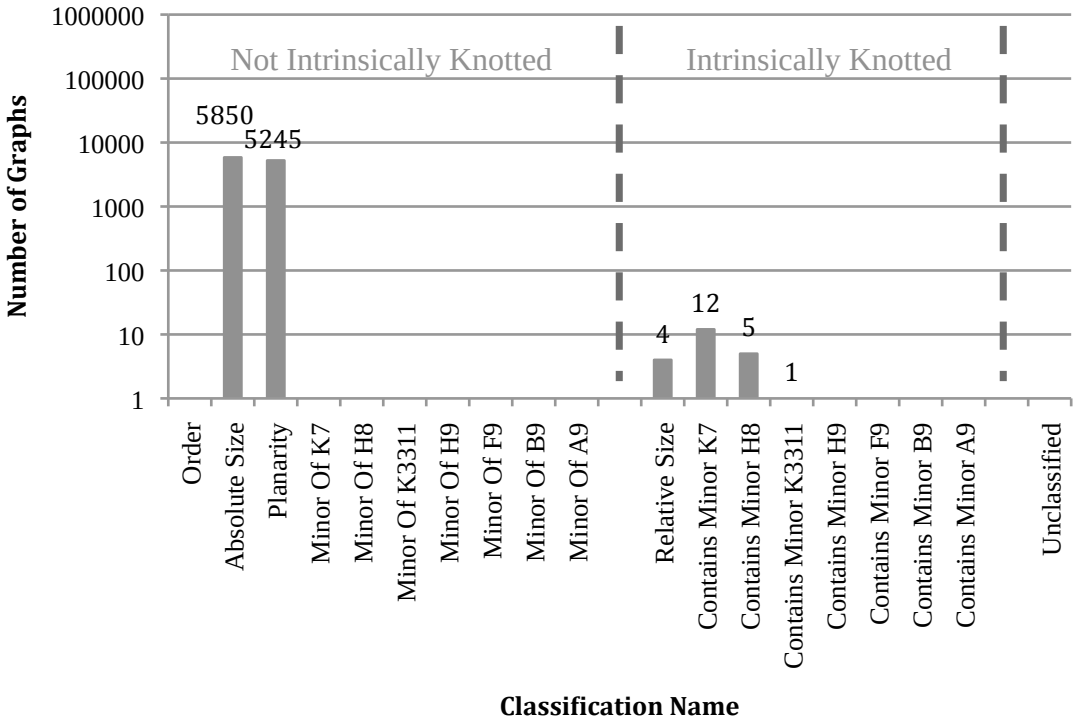


Figure 11. Classification Tests for Eight-Vertex Graphs

The tools also collected per graph processing time. Of the 11,117 graphs processed, the mean time spent on a single graph was less than one millisecond in Java and three milliseconds in Ruby. The median per graph time was less than one millisecond in both the Java and Ruby versions. In the Java version, the fastest graph was processed in less than one millisecond while the slowest graph took 17 milliseconds. The Ruby version also had the fastest graph completing in less than one millisecond, but the slowest graph took 2.152 seconds to complete. Figure 12 illustrates the per graph time differences between the Java and Ruby versions.

All 11,117 graphs on eight vertices were classified as either intrinsically knotted or *not* intrinsically knotted; thus, there are no indeterminate eight-vertex graphs.

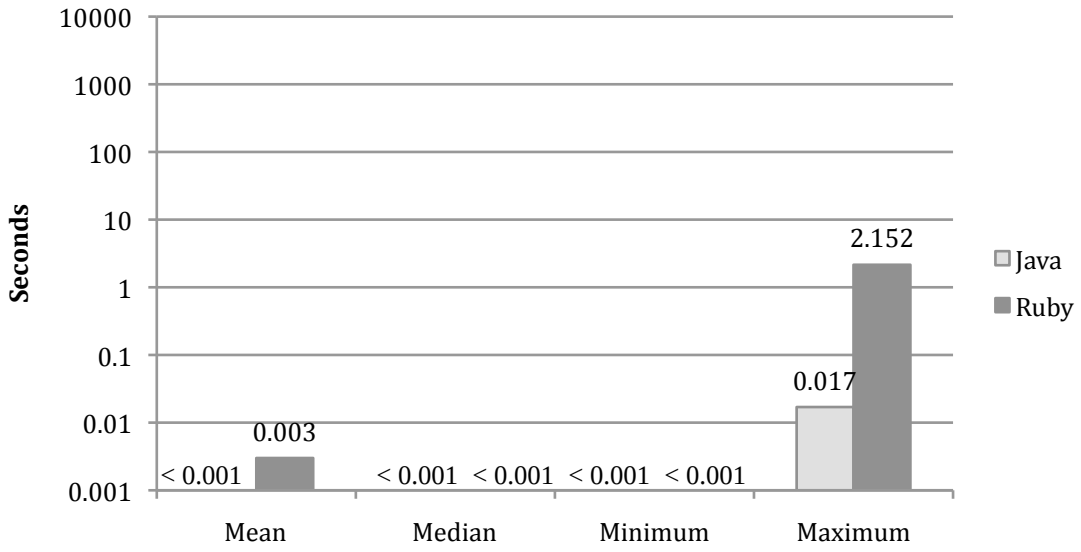


Figure 12. Per Graph Classification Times for Eight-Vertex Graphs

Nine-Vertex Graphs

The classification results for nine-vertex graphs were the same using both the Java and the Ruby tools. Each tool processed 261,080 graphs. The tools classified 259,055 of those graphs, or 99.22% of the total, as *not* intrinsically knotted, and 1,993 graphs, or 0.76% of the total, as intrinsically knotted. The remaining 32 graphs, or 0.01% of the total, had an intrinsically knotted property that the tools could not determine.

The tools classified all but 32 of the 261,080 as either intrinsically knotted or *not* intrinsically knotted. As a result, one of the tests was responsible for each of these graph's classification. Of the graphs that were *not* intrinsically knotted, the Absolute Size Classification classified 29,913, or 11.46% of the total. The Planarity Classification classified the remaining 229,142 non-intrinsically knotted graphs, 87.77% of the total. Of the 1,993 intrinsically knotted graphs, the Contains Minor K_7 Classification classified 1,272, 0.49% of the total. The Contains Minor B_9 Classification classified seven graphs, 0.00% of the total, while the Contains Minor A_9 Classification classified six graphs,

0.00% of the total. The Contains Minor H_8 Classification classified 543 graphs, 0.21% of the total, as intrinsically knotted, the Contains Minor K_{3311} Classification classified 39 graphs, 0.01% of the total, and the Contains Minor H_9 Classification classified 32 graphs, 0.01% of the total. Of the remaining intrinsically knotted graphs, the Contains Minor F_9 Classification classified 49, 0.02% of the total, and the Relative Size Classification classified 45, 0.02%. As mentioned, the tools were unable to classify 32 graphs, 0.01% of the total. This distribution is illustrated in Figure 13.

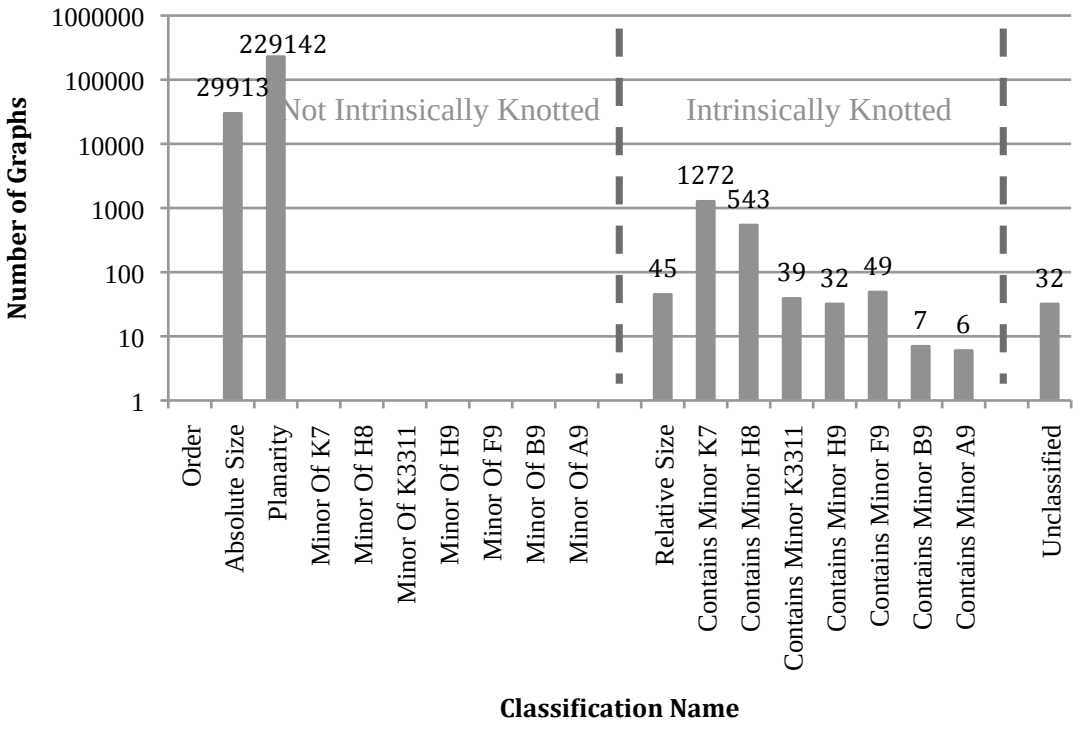


Figure 13. Classification Tests for Nine-Vertex Graphs

The results of the Java and Ruby classification runs on nine-vertex graphs differed when it came to timing. The total running time for the Java version was 17 minutes 53.302 seconds, while the Ruby version completed in three hours eight minutes

49.326 seconds. This time encompasses everything from start to finish which not only includes the actual time spent processing each graph, but also overhead tasks such as opening files, parsing files, as well as writing and formatting output. The total time spent classifying the graphs was 17 minutes 46.771 seconds in the Java version and three hours seven minutes 31.264 seconds in Ruby.

The tools also collected per graph processing time. Of the 261,080 graphs processed, the mean time spent on a single graph was four milliseconds in Java and 43 milliseconds in Ruby. The median per graph time was two milliseconds in Java and five milliseconds in Ruby. In the Java version, the fastest graph was processed in less than one millisecond while the slowest graph took 692 milliseconds. The Ruby version also had the fastest graph completing in less than one millisecond, but the slowest graph took 55 minutes 8.123 seconds to complete. Figure 14 illustrates the per graph time differences between the Java and Ruby versions.

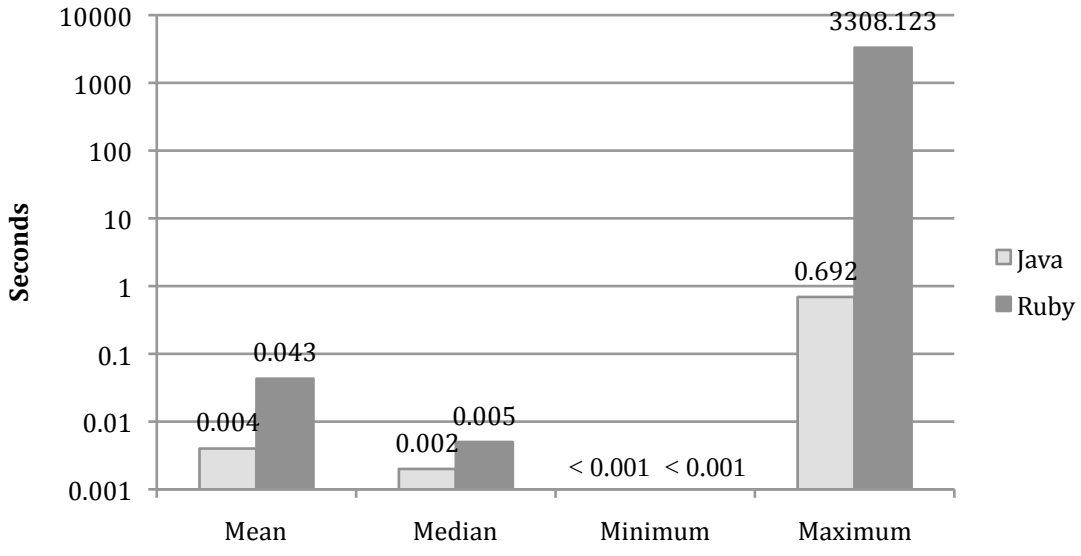


Figure 14. Per Graph Classification Times for Nine-Vertex Graphs

Of the 261,080 nine-vertex graphs, 32 of them were classified as indeterminate with respect to the property of intrinsic knotting. These 32 graphs ranged in size from 21 edges to 29 edges. The distribution of these sizes is illustrated in Figure 15.

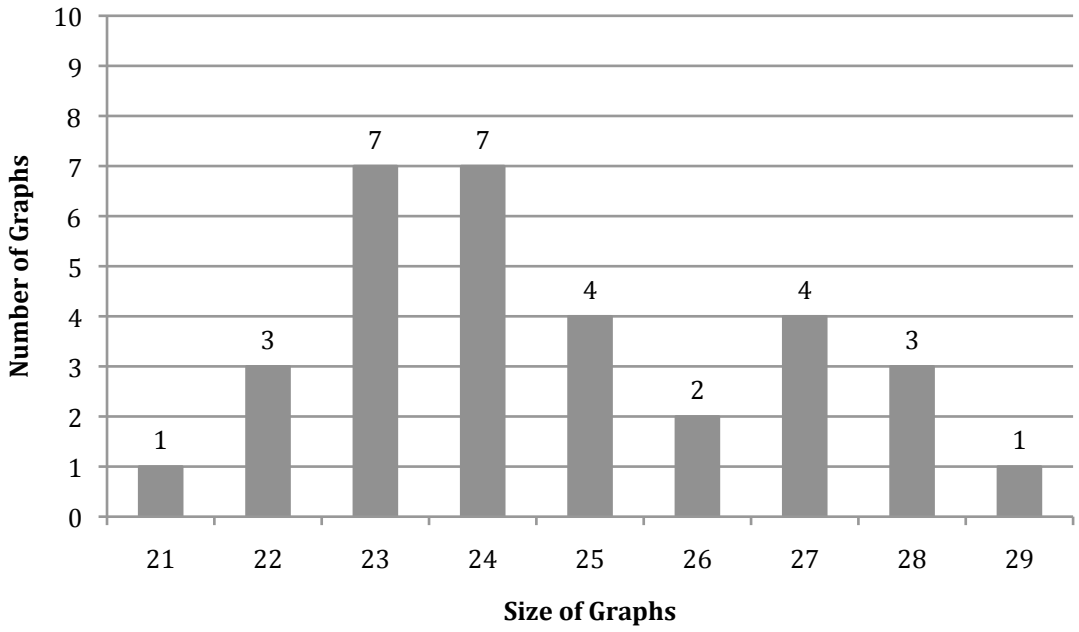


Figure 15. Size Distribution of Indeterminate Nine-Vertex Graphs

The edge listings for all 32 indeterminate graphs along with their complements can be found in Appendix C. The complements have been included because when studying a graph, it is often easier to study the edges that do not exist than studying those that do. Figure 16 through Figure 47 provide visual representations of each of these 32 indeterminate graphs along with their complements. The figures include the identification numbers associated with each graph; this is a part of the raw graph data, and each graph has a unique number.

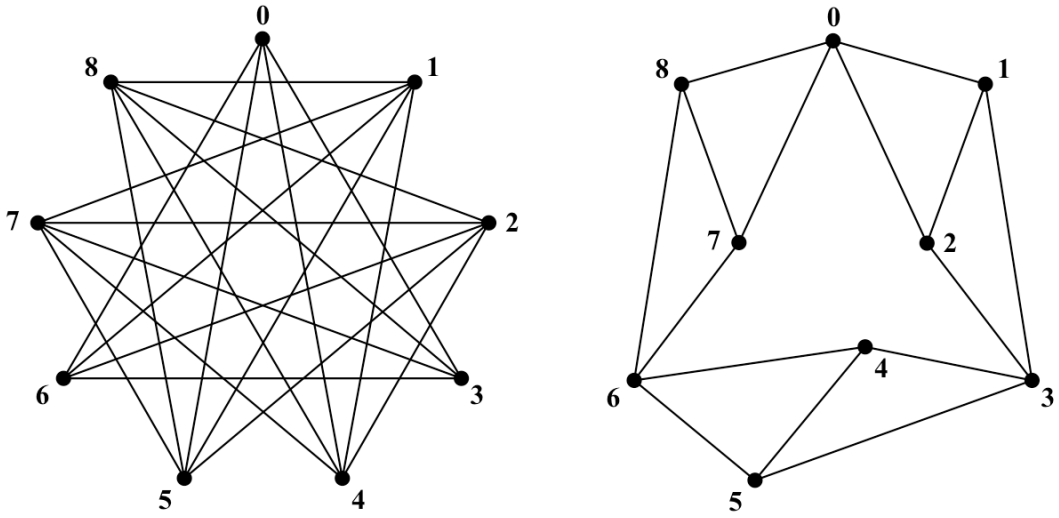


Figure 16. Indeterminate Nine-Vertex Graph #243680 and Complement

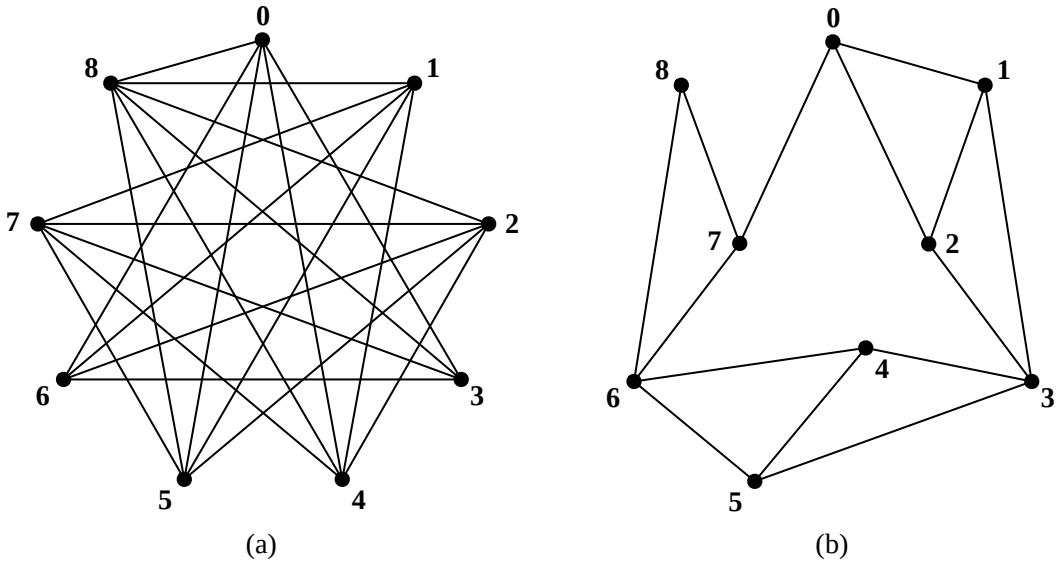


Figure 17. Indeterminate Nine-Vertex Graph #243683 and Complement

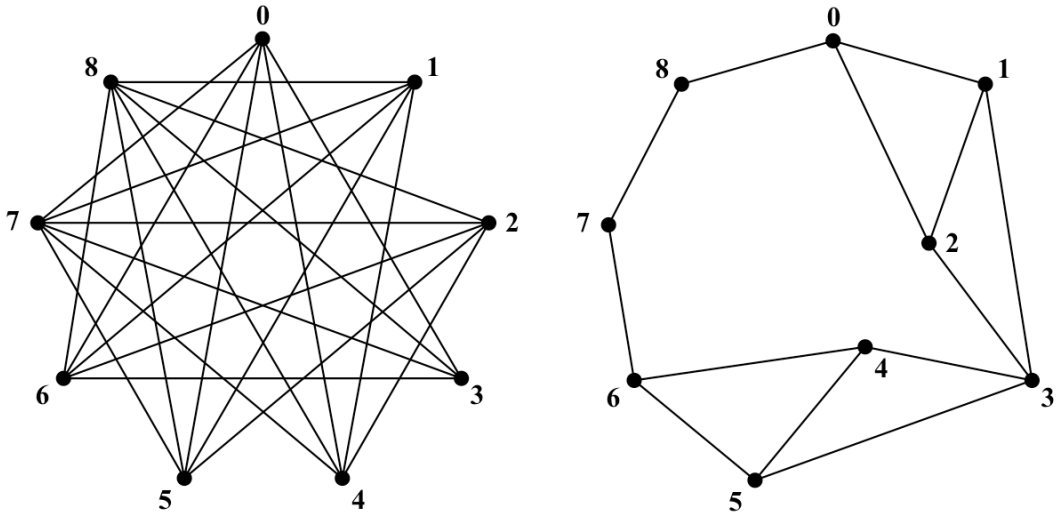


Figure 18. Indeterminate Nine-Vertex Graph #243745 and Complement

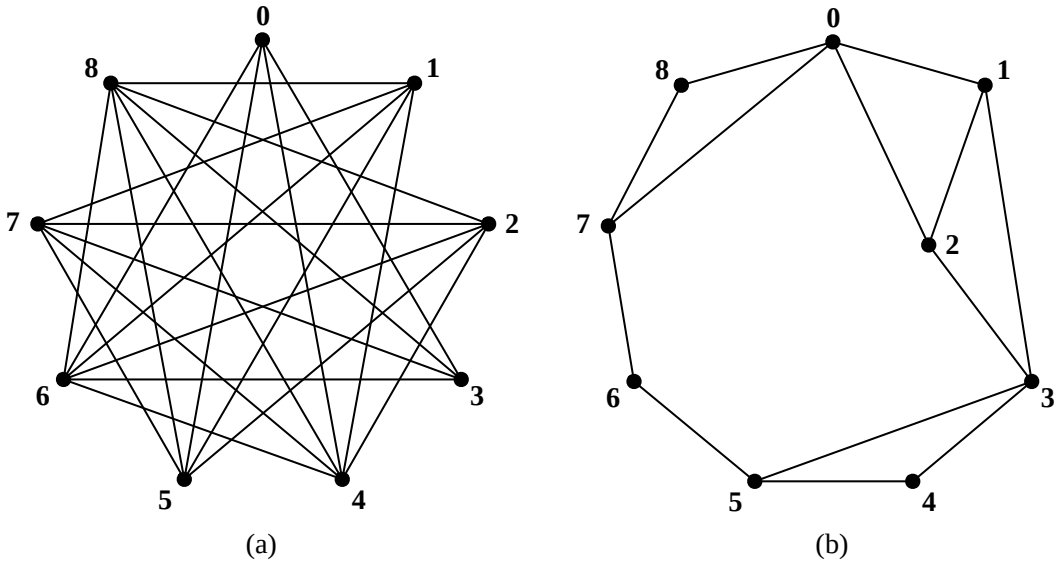


Figure 19. Indeterminate Nine-Vertex Graph #244064 and Complement

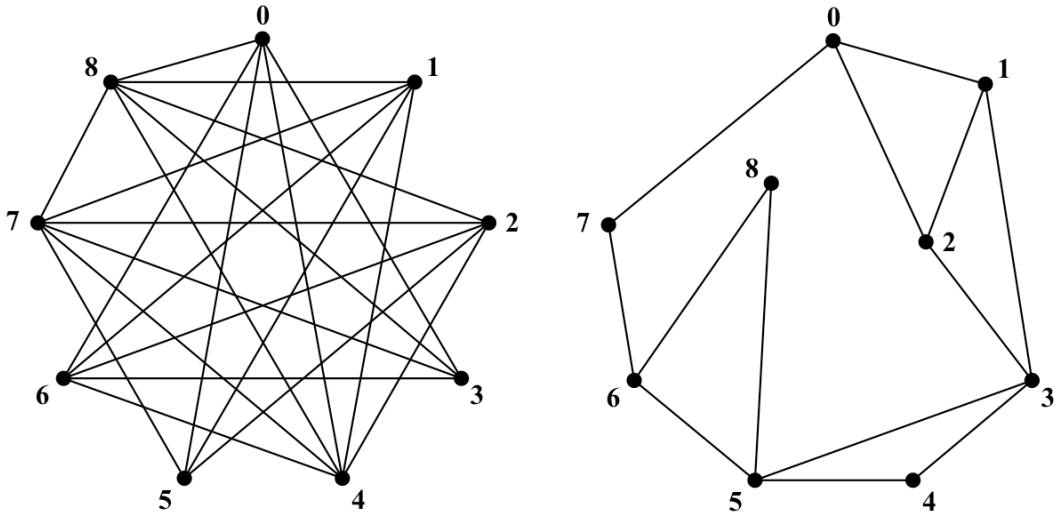


Figure 20. Indeterminate Nine-Vertex Graph #244065 and Complement

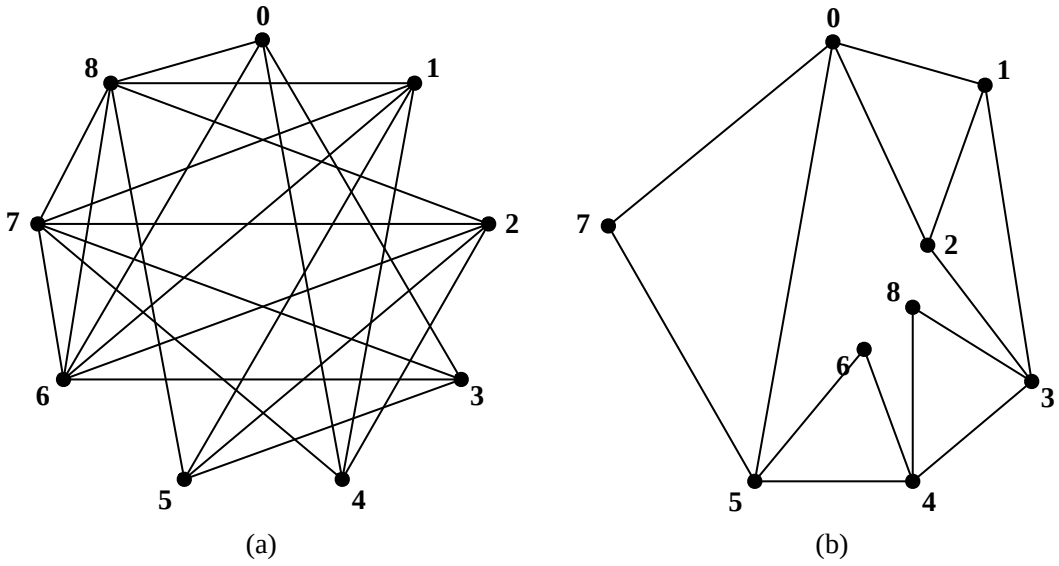


Figure 21. Indeterminate Nine-Vertex Graph #244632 and Complement

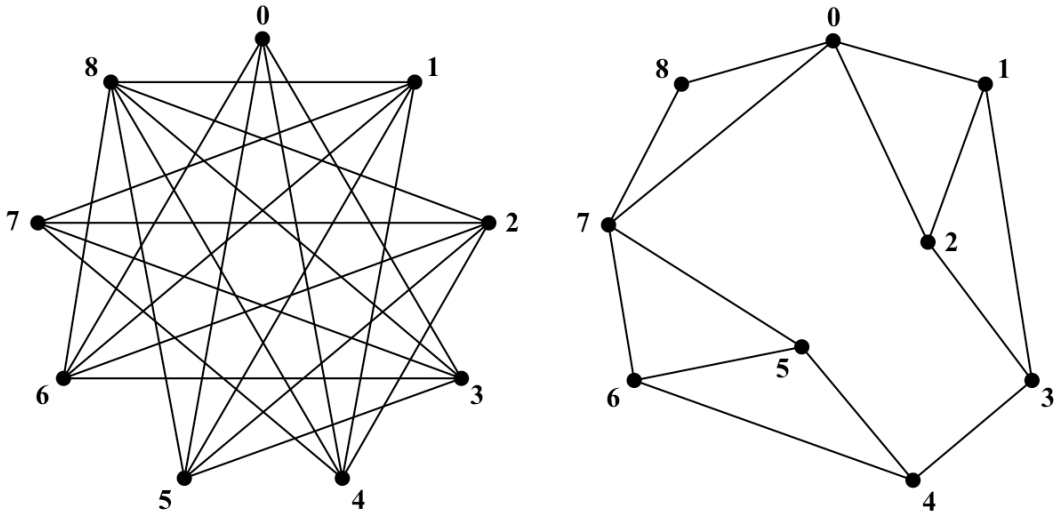


Figure 22. Indeterminate Nine-Vertex Graph #245103 and Complement

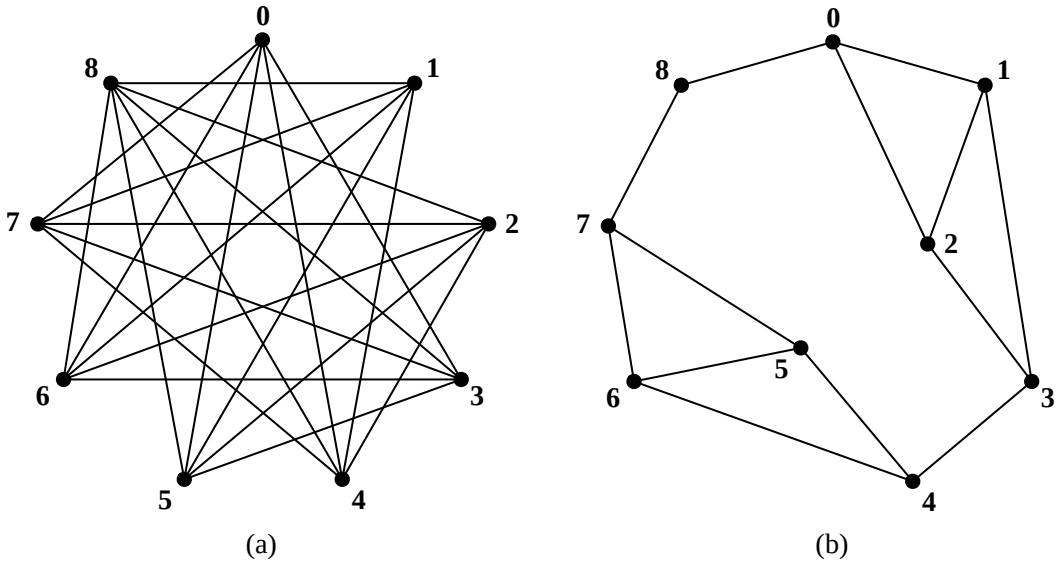


Figure 23. Indeterminate Nine-Vertex Graph #245113 and Complement

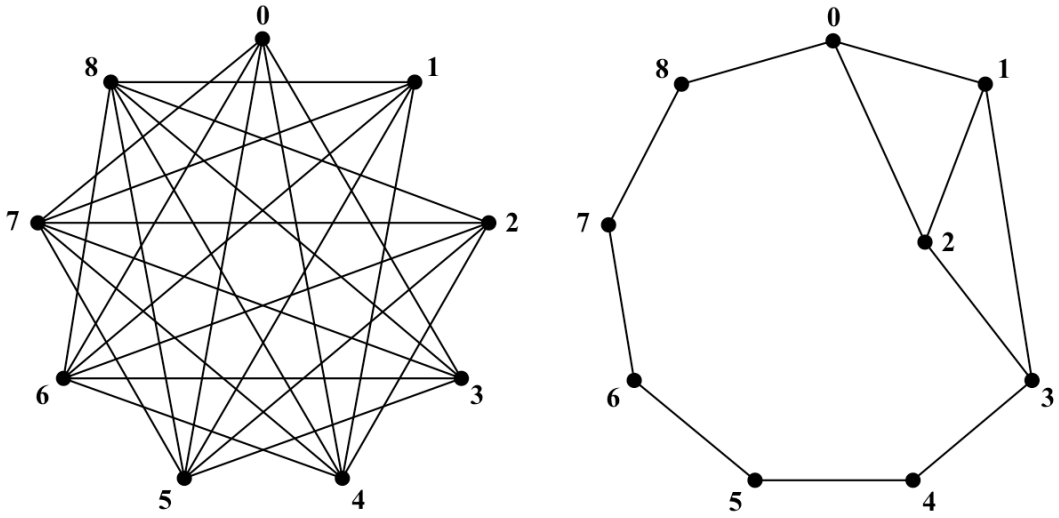


Figure 24. Indeterminate Nine-Vertex Graph #245195 and Complement

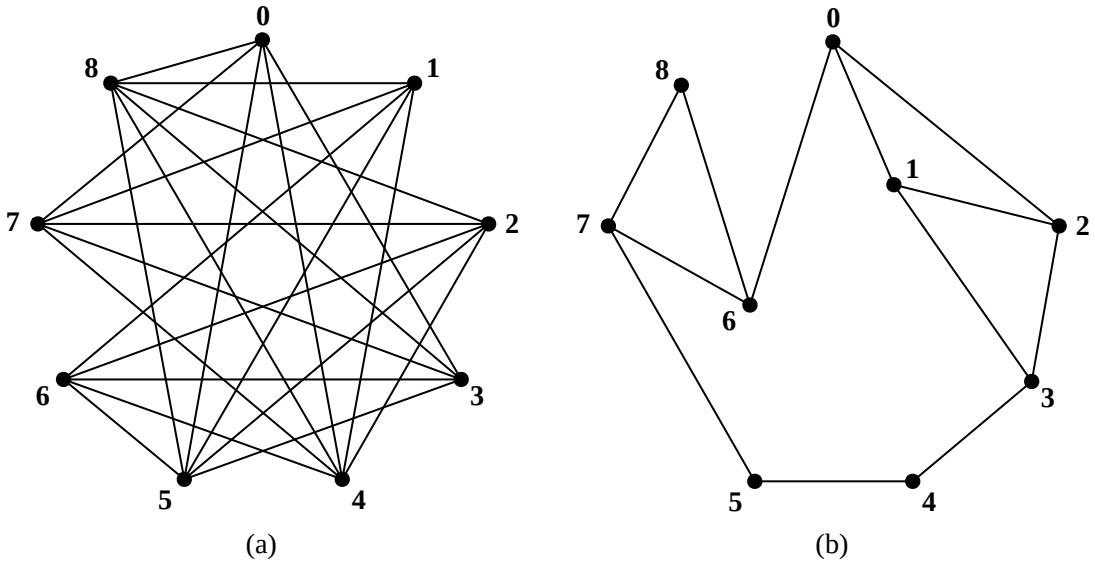


Figure 25. Indeterminate Nine-Vertex Graph #245238 and Complement

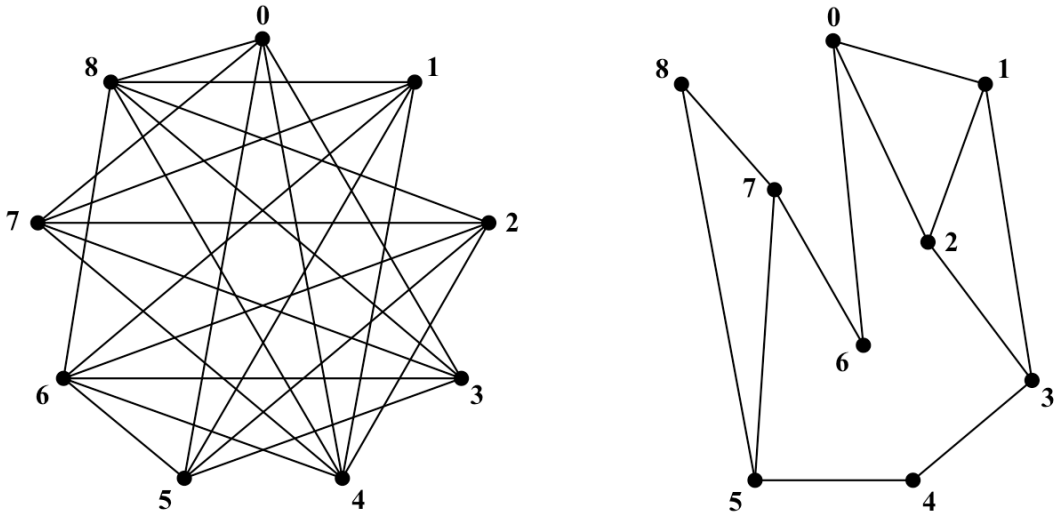


Figure 26. Indeterminate Nine-Vertex Graph #245239 and Complement

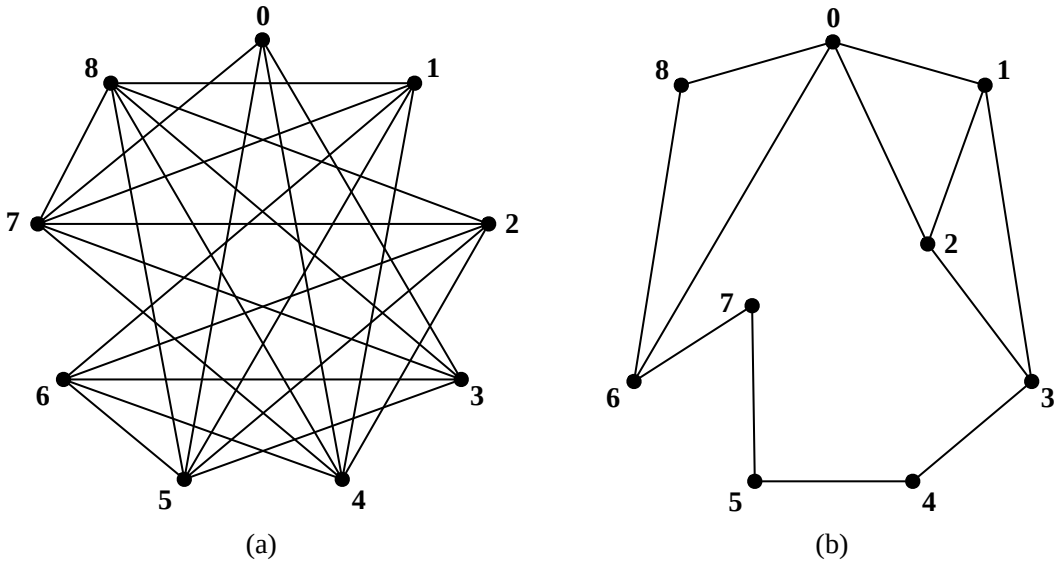


Figure 27. Indeterminate Nine-Vertex Graph #245246 and Complement

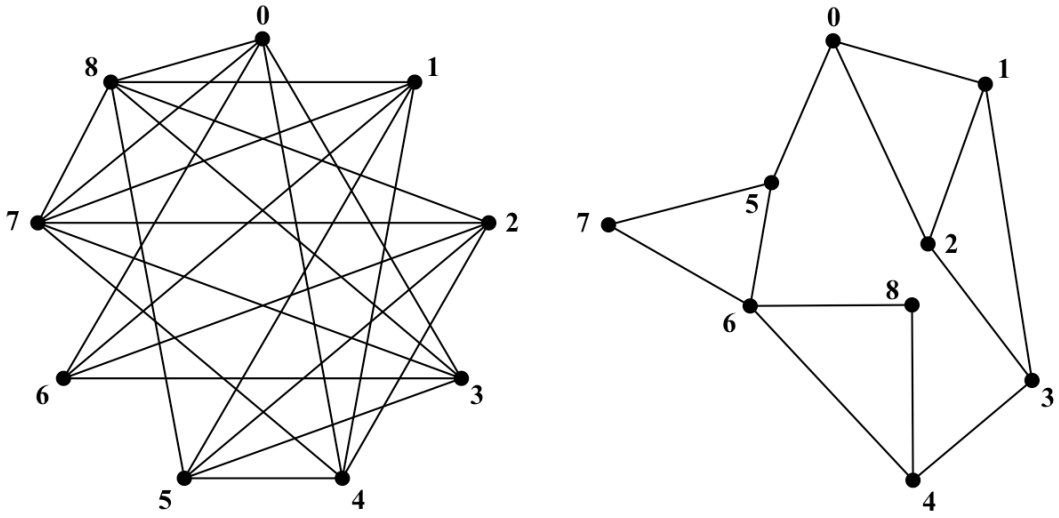


Figure 28. Indeterminate Nine-Vertex Graph #245605 and Complement

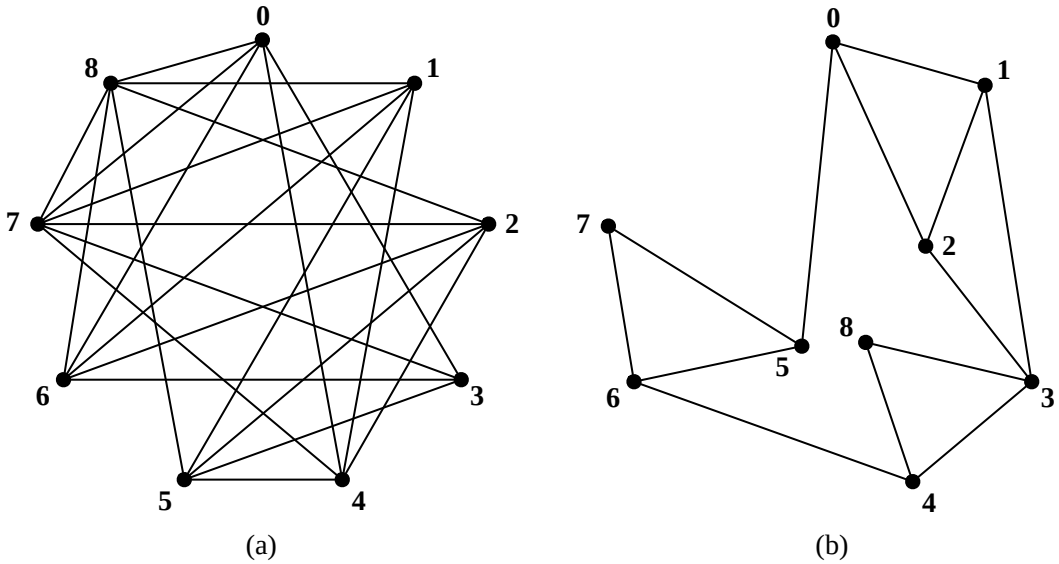


Figure 29. Indeterminate Nine-Vertex Graph #245608 and Complement

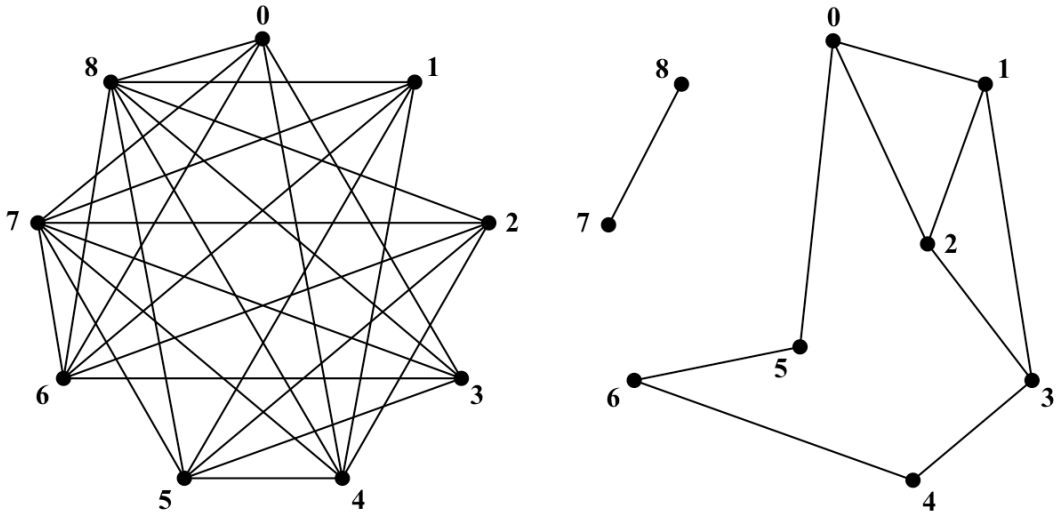


Figure 30. Indeterminate Nine-Vertex Graph #245677 and Complement

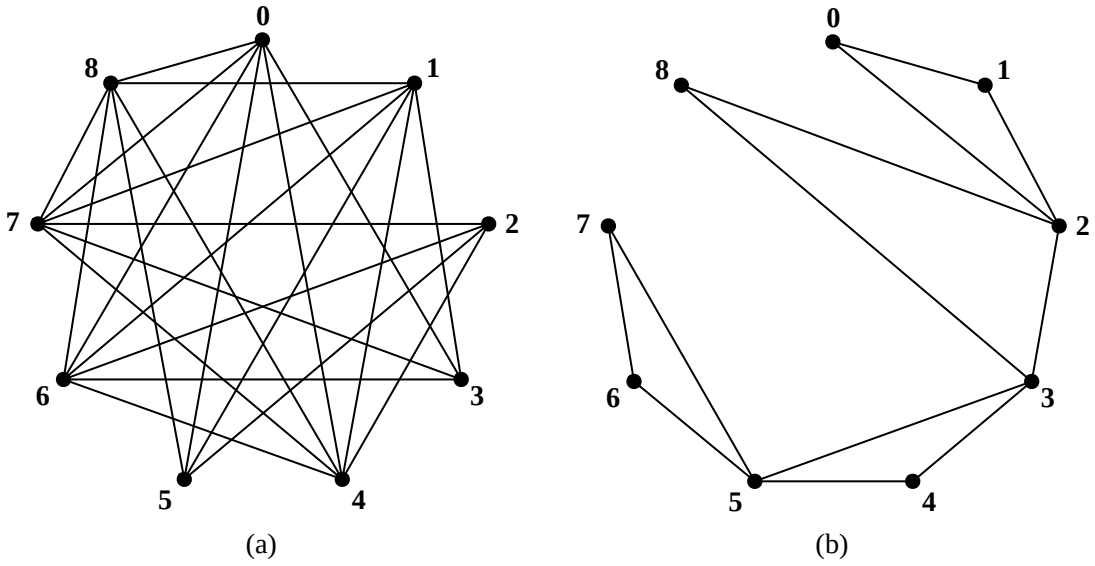


Figure 31. Indeterminate Nine-Vertex Graph #255220 and Complement

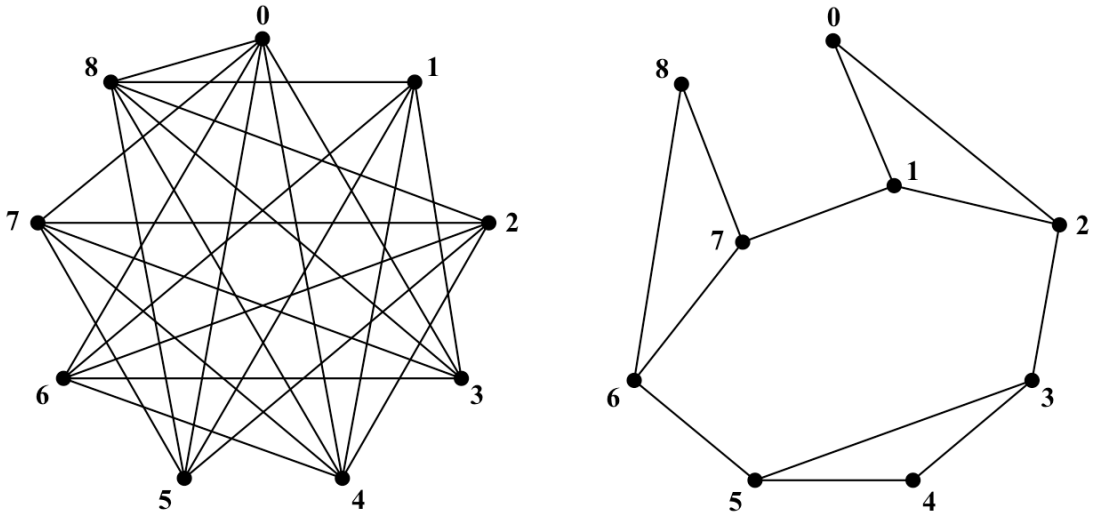


Figure 32. Indeterminate Nine-Vertex Graph #255244 and Complement

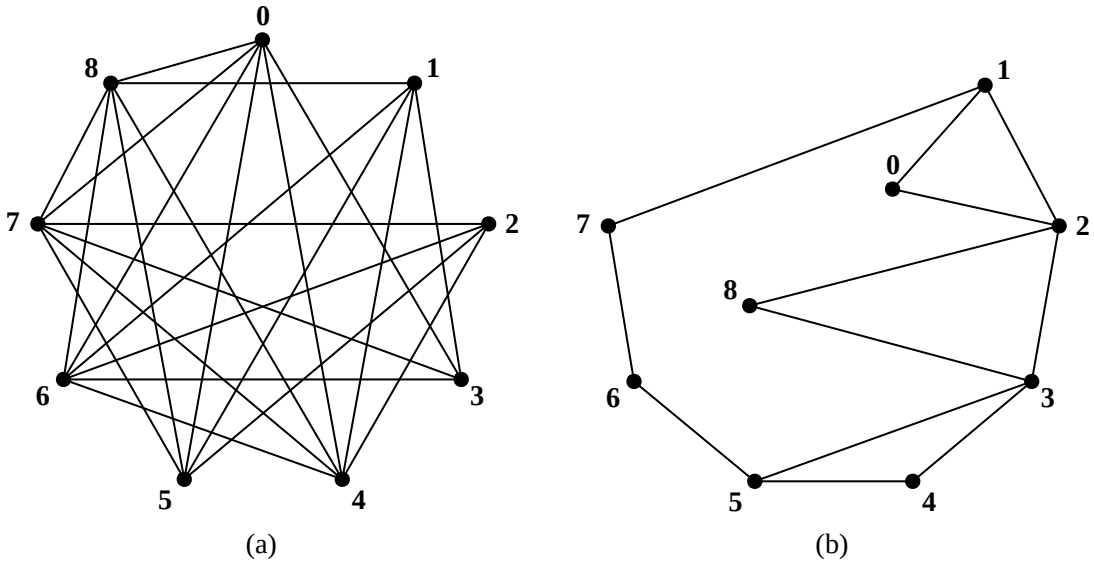


Figure 33. Indeterminate Nine-Vertex Graph #255247 and Complement

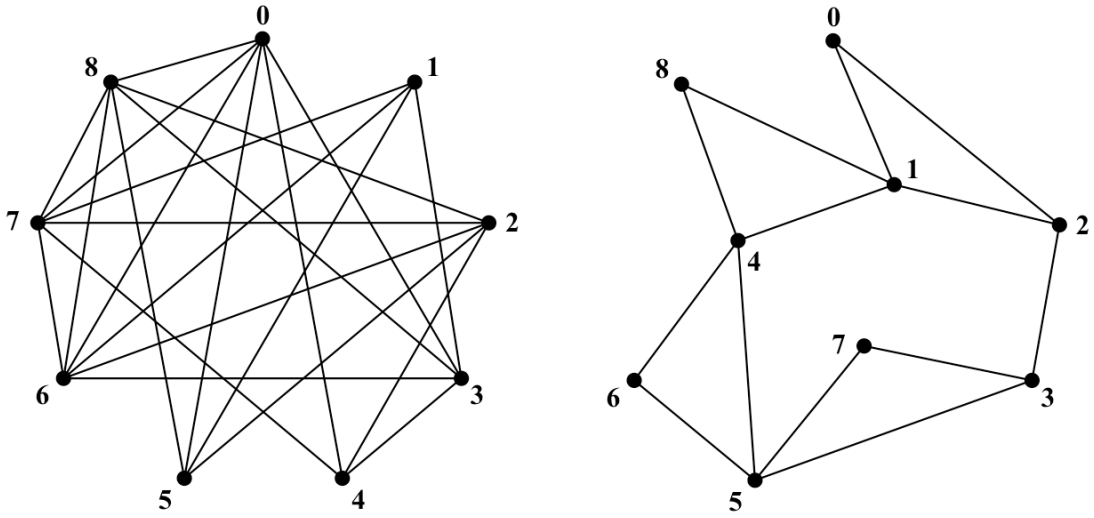


Figure 34. Indeterminate Nine-Vertex Graph #255925 and Complement

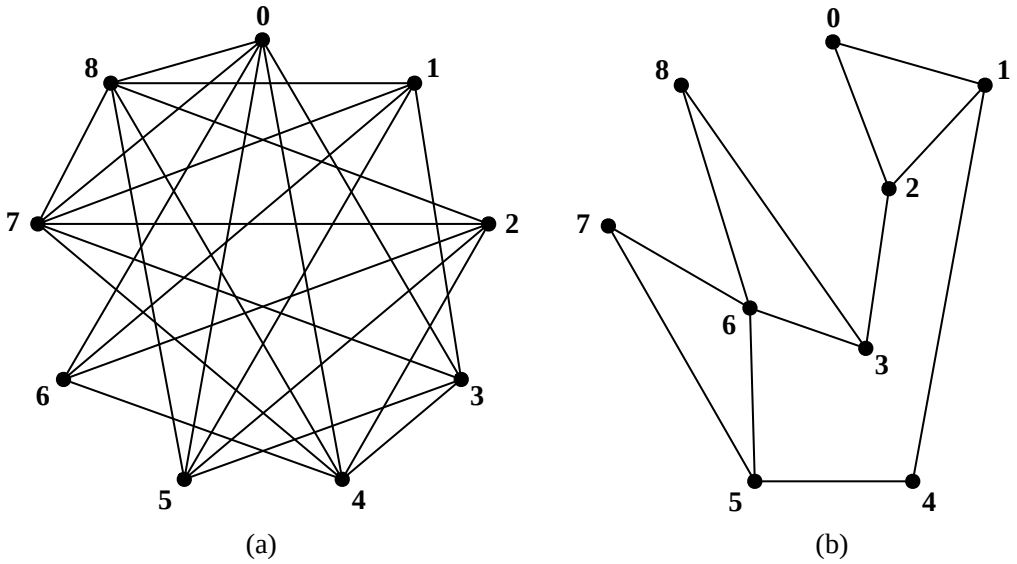


Figure 35. Indeterminate Nine-Vertex Graph #256305 and Complement

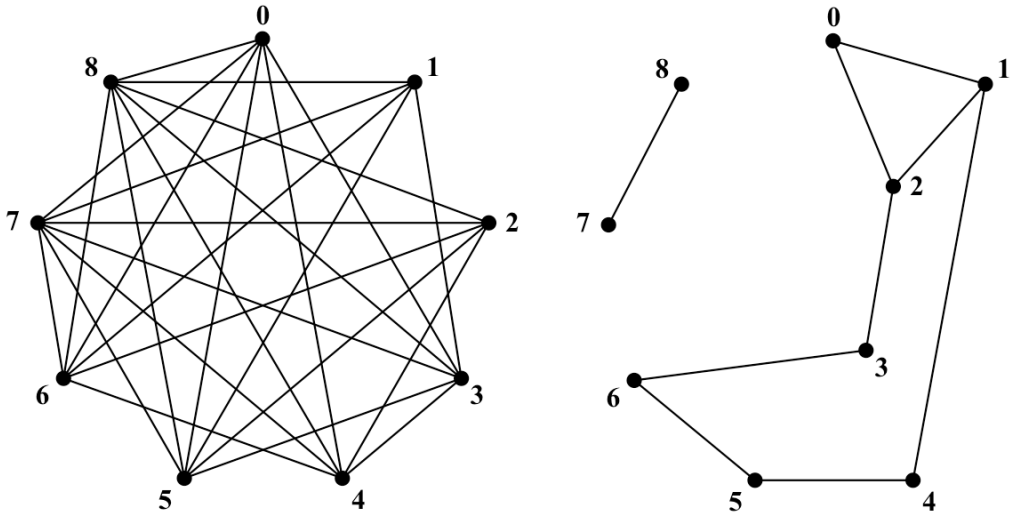


Figure 36. Indeterminate Nine-Vertex Graph #256338 and Complement

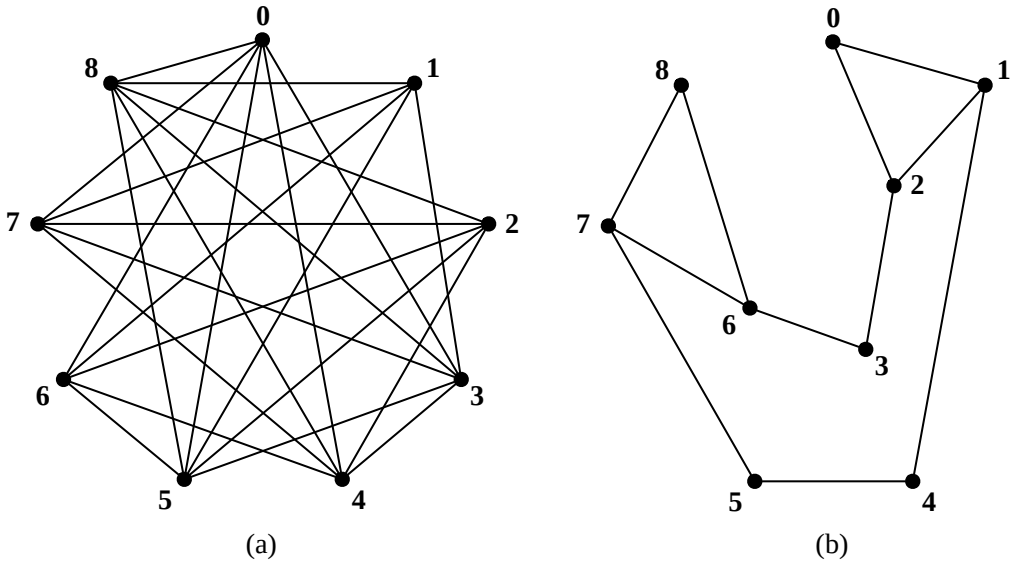


Figure 37. Indeterminate Nine-Vertex Graph #256363 and Complement

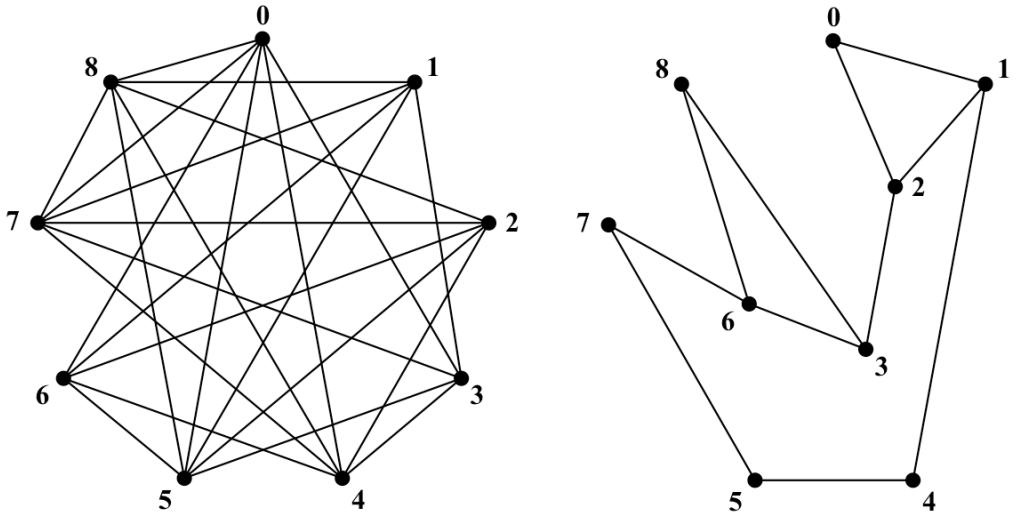


Figure 38. Indeterminate Nine-Vertex Graph #256368 and Complement

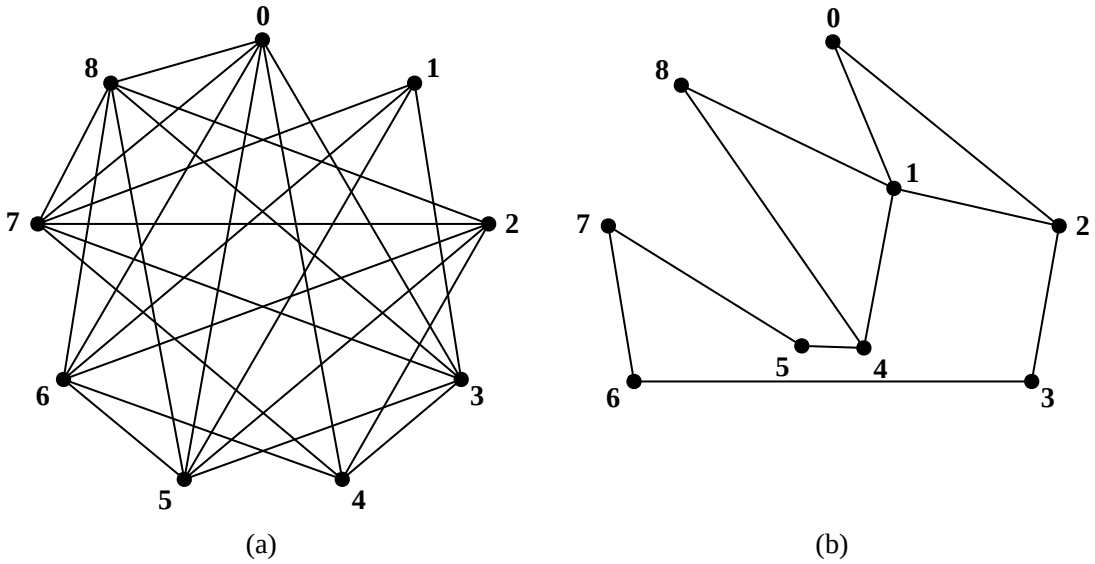


Figure 39. Indeterminate Nine-Vertex Graph #256372 and Complement

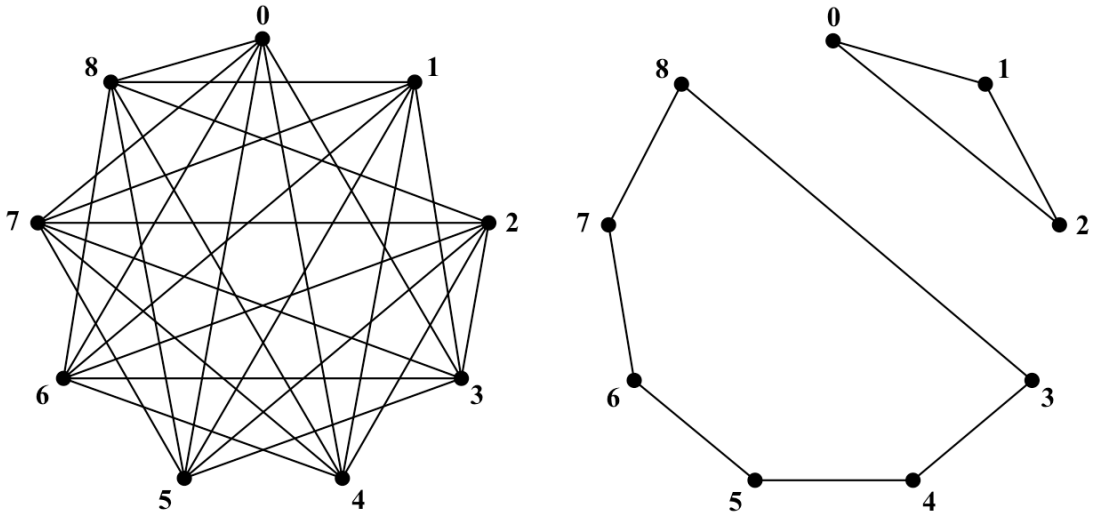


Figure 40. Indeterminate Nine-Vertex Graph #256510 and Complement

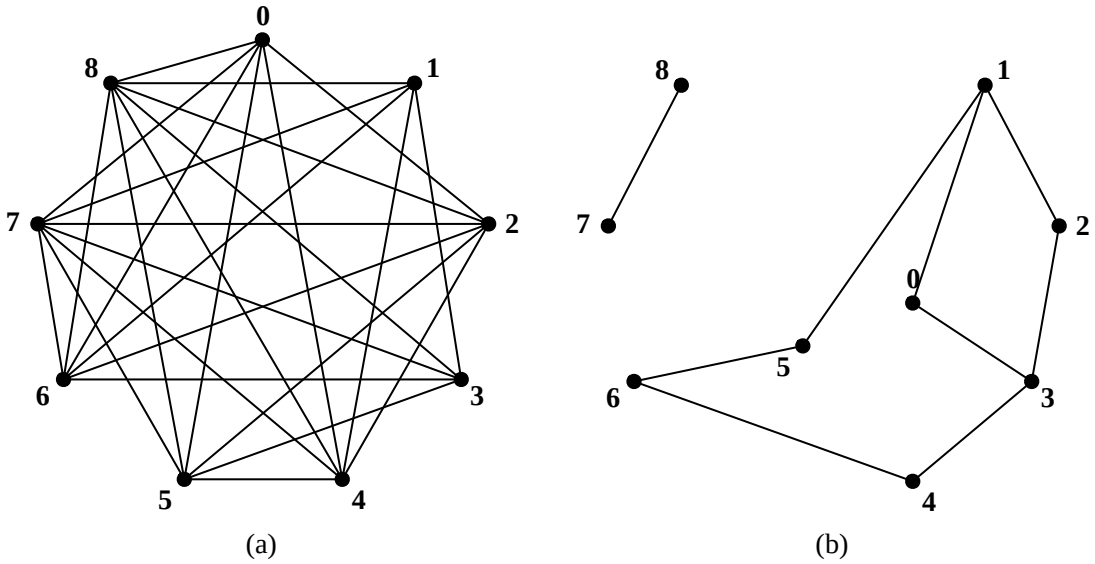


Figure 41. Indeterminate Nine-Vertex Graph #260624 and Complement

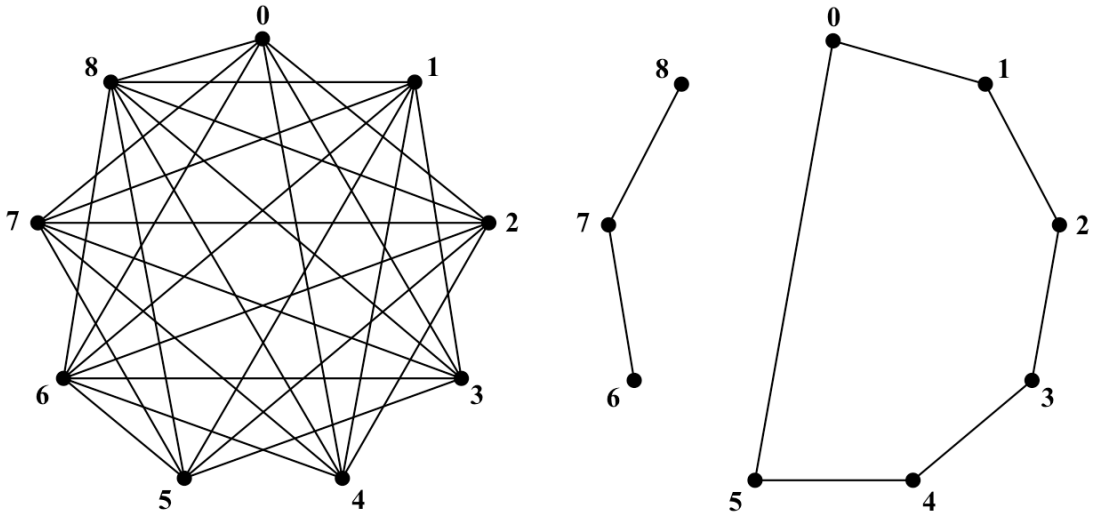


Figure 42. Indeterminate Nine-Vertex Graph #260908 and Complement

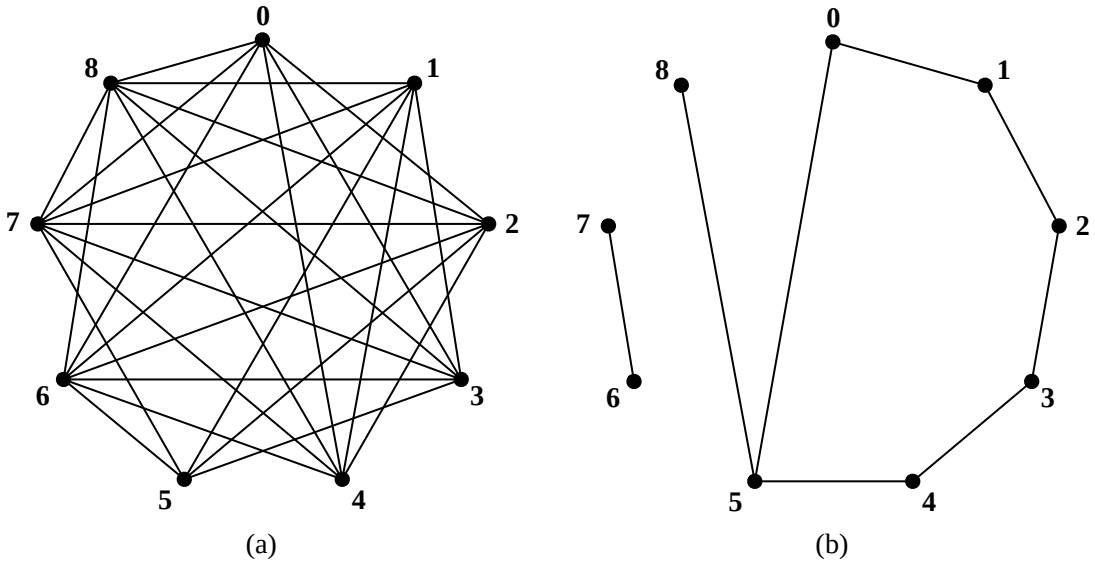


Figure 43. Indeterminate Nine-Vertex Graph #260909 and Complement

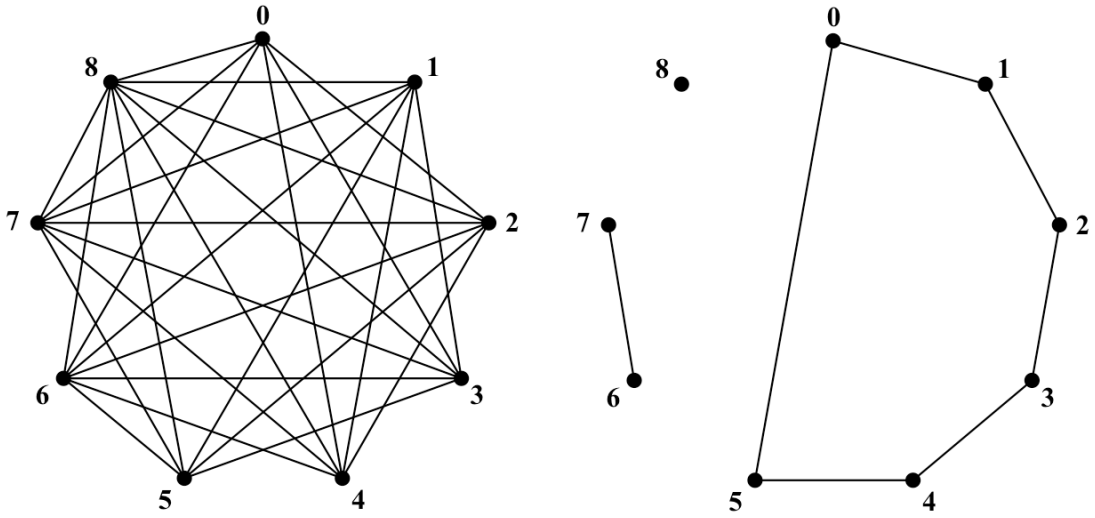


Figure 44. Indeterminate Nine-Vertex Graph #260910 and Complement

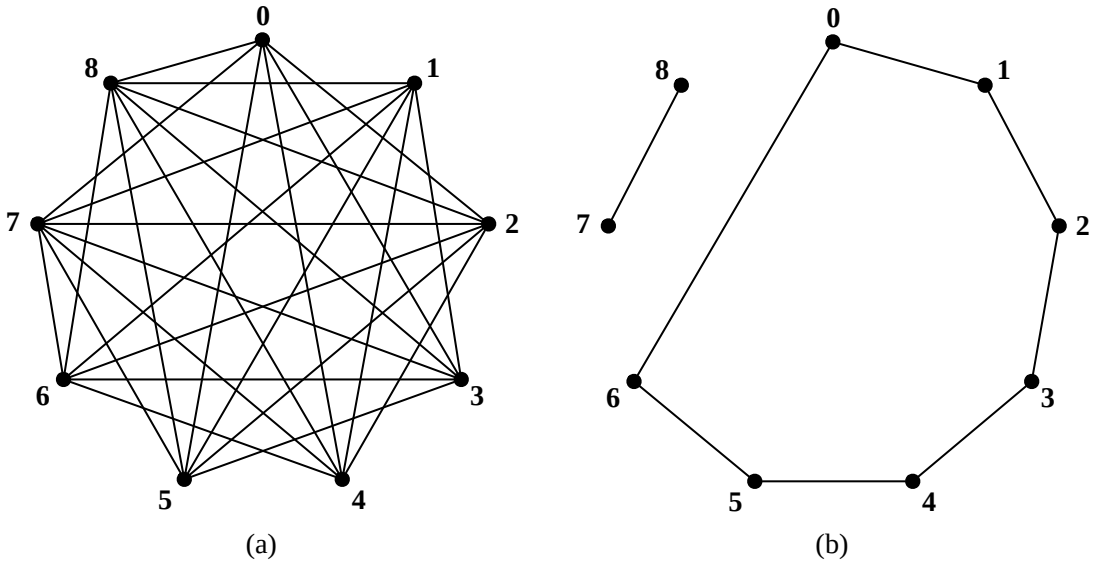


Figure 45. Indeterminate Nine-Vertex Graph #260920 and Complement

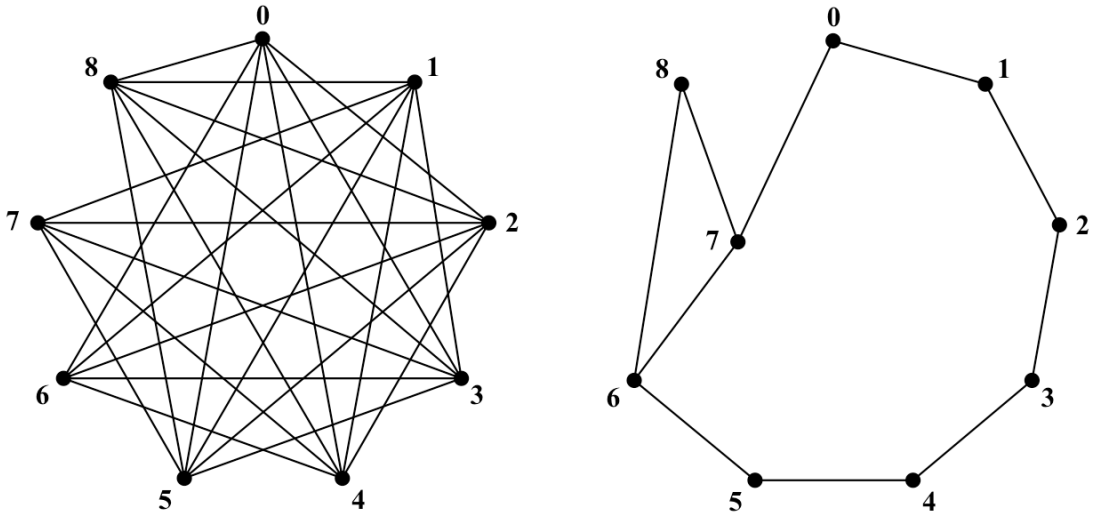


Figure 46. Indeterminate Nine-Vertex Graph #260922 and Complement

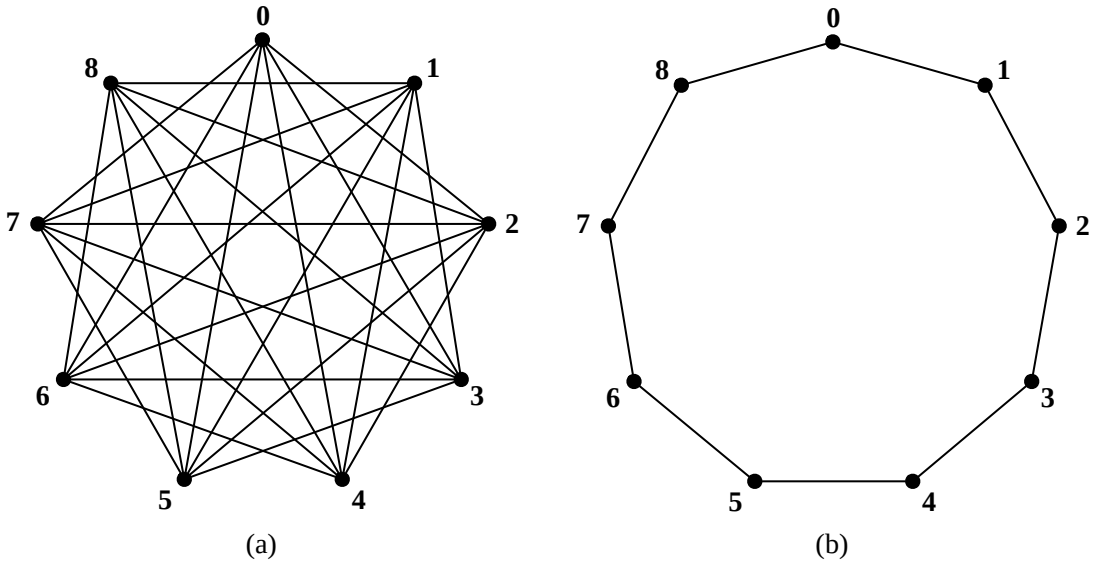


Figure 47. Indeterminate Nine-Vertex Graph #260928 and Complement

CHAPTER V

ANALYSIS

The goal of this project was to attempt to classify all connected graphs on seven, eight, and nine vertices with respect to the property of intrinsic knotting. While the summarized results in the previous section clearly address this goal, it is still important to evaluate these results and determine what they mean.

Intrinsically Knotted Classifications

The results of the classifications of the 853 distinct, connected graphs on seven vertices with respect to the property of intrinsic knotting were exactly as expected. Kohara and Suzuki proved that K_7 is minor minimal with respect to the property of intrinsic knotting and thus is the only graph on seven vertices that is intrinsically knotted, since every other graph on seven vertices is a minor of K_7 [6]. The tools classified the remaining 852 graphs as *not* intrinsically knotted, exactly as expected.

Similar to the seven-vertex graphs, the classification results for the 11,117 distinct, connected eight-vertex graphs were also as expected. Campbell, Mattman, Ottman et al. [8] and Blain, Bowlin, Fleming et al. [7] proved, independently, that there are 23 graphs on eight vertices that are intrinsically knotted. The project's classification efforts revealed 22 connected graphs that are intrinsically knotted. The one graph difference is due to the fact that the set of 23 graphs includes the graph K_7 with an extra

vertex (with zero edges connected to it). This graph, $K_7 + K_1$, is not connected and thus was not included in the set of graphs to test. It is unnecessary to test this graph because its intrinsic knotting is determined by that of its components. In particular, since K_7 is intrinsically knotted, the graph $K_7 + K_1$ is too. Thus, the 23 graphs proved to be intrinsically knotted less the one graph that is not connected leaves 22 connected graphs on eight vertices that are intrinsically knotted. The 22 graphs found to be intrinsically knotted by Campbell, Mattman, Ottman et al. [8] and Blain, Bowlin, Fleming et al. [7] are the same 22 graphs that the classification tests found to be intrinsically knotted. Since all of the graphs were classified, the fact that the remaining 11,095 graphs were *not* intrinsically knotted was also as expected. In summary, the classification of all 11,117 graphs on eight vertices with respect to the property of intrinsic knotting was as expected.

While the classifications on seven and eight vertices were a bit like a control group that gave confidence in the approach because the expected results were known, the graphs on nine vertices had not been addressed by the mathematics literature. Certainly, the tools were expected to find plenty of intrinsically knotted graphs, and a huge number more of *not* intrinsically knotted graphs, but exact numbers had never been proven. Even though these numbers were unknown, one would expect the percentage of unknown graphs to be similar across the different sets of graphs. As expected, the tools classified over 99% of the nine-vertex graphs as *not* intrinsically knotted, a classification percentage that was also found with the seven and eight-vertex graphs. Looking at it another way, the 1,993 intrinsically knotted graphs accounted for less than 1% of all of the graphs on nine vertices, which was expected since a similar percentage was found with the seven and eight-vertex graphs. There was an increase from 0.12% of the graphs

on seven vertices being intrinsically knotted to 0.20% on eight vertices. Since the graphs get more complex as they get larger, one would expect that on nine vertices there would be a higher percentage of intrinsically knotted graphs. Thus, the result of 0.76% of the graphs on nine vertices found to be intrinsically knotted was within the expectations. What was not expected, but was certainly a pleasant surprise, was the discovery of 32 graphs that could not be classified with respect to the property of intrinsic knotting.

The 32 graphs on nine vertices that could not be classified are the most exciting discovery in this whole project. Within this set of 32 graphs could exist one or more previously undiscovered minor minimal intrinsically knotted graphs. It is also possible that there are graphs within this set that are *not* intrinsically knotted. This would mean the set of classification tests to prove a graph is *not* intrinsically knotted is incomplete. Either way, the 261,080 graphs on nine vertices can be narrowed down to these 32 graphs for further investigation.

In order to accelerate the investigation of these 32 indeterminate graphs, the project included the creation of the *expansion_mapper* tool, which processes a list of graphs, determining which graphs are expansions or minors of the others. The output is a listing detailing how each graph is related to each of the other graphs. See Appendix A for more details and usage information about this tool. Figure 48 illustrates the results from running the *expansion_mapper* on the 32 indeterminate graphs. The graphs at the top of the illustration are not expansions of any other of the 32 graphs. These five graphs at the top—243680, 245103, 244632, 245677, and 256510—would likely be the best place to start further research, as any discovery among these graphs could have a cascading effect through the other graphs. Diagrams of each of these graphs with their

complement can be found in the Results Chapter. The edge listings as well as the raw output from *expansion_mapper* can be found in Appendix C.

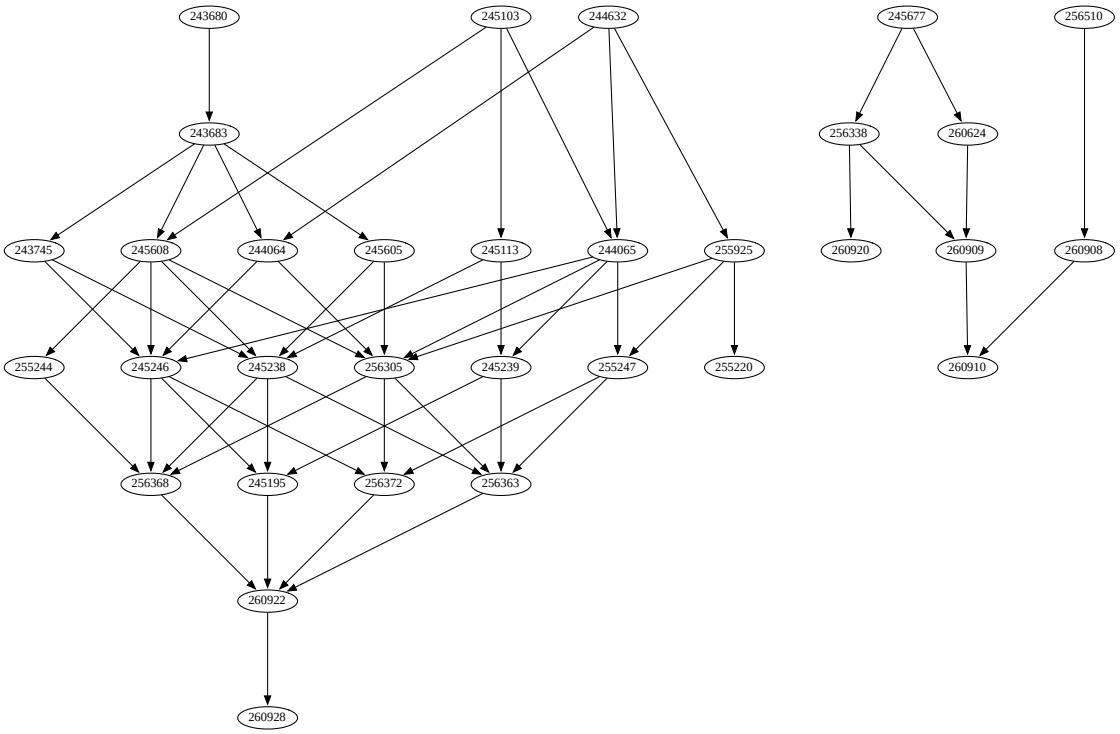


Figure 48. Expansion Map of All 32 Indeterminate Graphs on Nine Vertices

This diagram can prove (and, indeed already has proven) very useful to future research by simply applying the basic definitions of minor and a minor minimal property. For example, Ramin Naimi reports that students working with him have proved that graph 245103 is intrinsically knotted [3]. If so, Figure 48 would indicate that this graph and its 15 expansions are all intrinsically knotted, leaving just 16 indeterminate graphs. Naimi and his students also believe that graph 243680 is *not* intrinsically knotted, by proposing a possible unknotted embedding, but that graph 243683 is intrinsically knotted [3]. This proposed unknotted embedding of graph 243680 is shown in Figure 49.

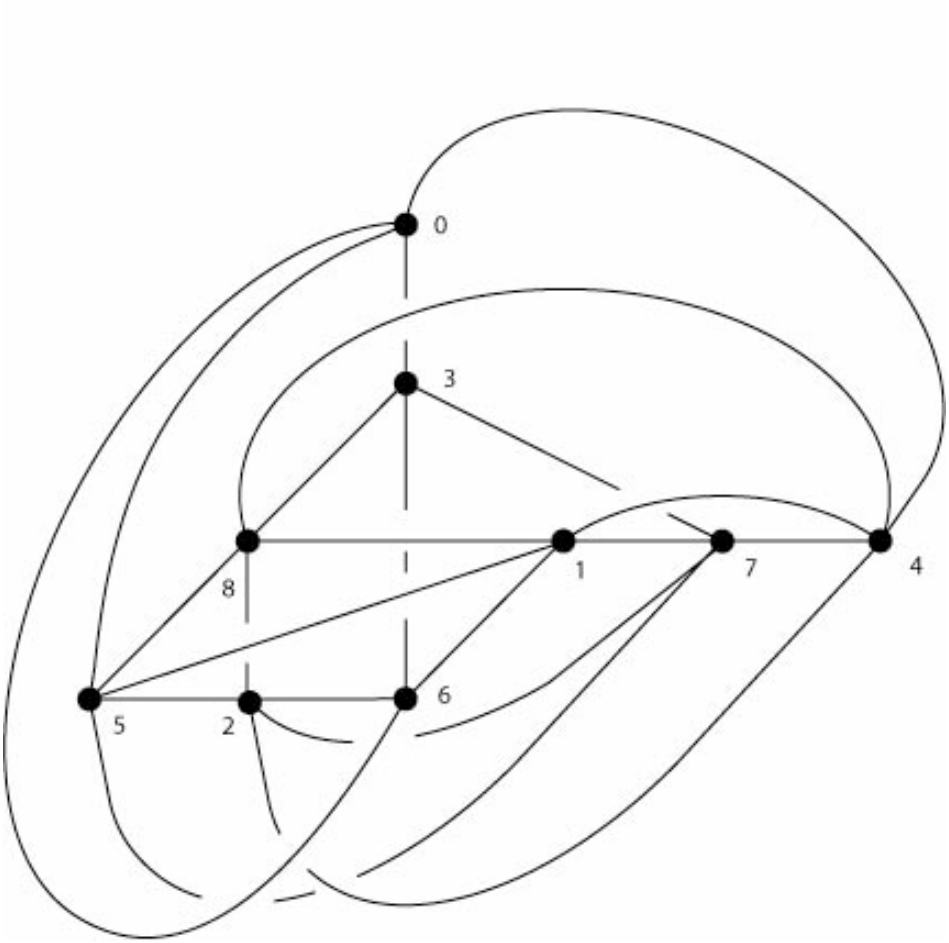


Figure 49. A Proposed Unknotted Embedding of Graph 243680

Using the diagram in Figure 48, one can deduce that graphs 243745, 244064, and 245605 are also intrinsically knotted, so that only 11 indeterminate graphs remain. Thus, the *expansion_mapper* output already is proving integral in the classification of the 32 indeterminate graphs. Coincidentally, if graph 243680 proves to be unknotted, then it is the first instance of a graph that is *not* intrinsically knotted yet fails the Planarity Classification test. All non-intrinsically knotted graphs on eight or fewer vertices, along with all others on nine vertices with 21 or fewer edges can be shown as *not* intrinsically knotted by the Planarity Classification.

While the classification of graphs on seven and eight vertices was as expected, the discovery of 32 indeterminate graphs on nine vertices is very exciting and should be investigated further in future research.

Classification Test Distributions

For all of the graphs that could be classified as either intrinsically knotted or *not* intrinsically knotted, there was a test that determined that classification. While this information was collected, it is not entirely relevant. It can be used as proof for the determined classification of a graph, but it is not necessarily the exclusive test that could determine that classification. It was merely the first test that gave a non-indeterminate result. The order the classification tests were run, which is discussed in the Methods Chapter, certainly impacted the distribution of those classifications. Many of the tests may have yielded the same outcome for a given graph, but that information was not important for this study. It is true however that once a classification test determined a graph as intrinsically knotted or *not* intrinsically knotted, no other test would change that outcome. As a result, the distribution between the tests is not as important as the outcome of the test.

With this said, it is particularly interesting just how useful the Planarity Classification proved to be. On the nine-vertex graphs, it was responsible for the classification of over 87% of the graphs. Furthermore, the fact that on eight vertices some graphs were classified by the Contains Minor H_8 and Contains Minor K_{3311} Classifications, give the project some validity that the classifications were working as expected, since H_8 and K_{3311} are eight-vertex graphs. Similarly, on nine vertices

classifications by the Contains Minor A_9, Contains Minor B_9, Contains Minor F_9 and Contains Minor H_9 Classifications also offer similar validity.

The prevalence of the Planarity Classification suggested that it could be used to classify almost all non-intrinsically knotted graphs. As mentioned above, further investigation showed that it in fact classifies all non-intrinsically knotted graphs on seven and eight vertices as well as all non-intrinsically knotted graphs on nine vertices of 21 or fewer edges except for 243680.

Even though the test that determined a classification is not immediately relevant, it can be used as a proof of the classification.

Timing

While speed and efficiency were not a priority when designing the classification tools, the tools did collect timing related information. The absolute timing, while collected, is not as interesting as the relative timing when comparing sets of graphs or individual graphs. Naturally, the speed of the CPU, the number of other running processes and the speed of the language implementations for the given operating system all affect the absolute timing results. A broad evaluation of all of the timing data does demonstrate clearly that the Java version is faster than the Ruby version. This is mostly due to the different algorithmic approaches between the two versions. The Java version has a more complex, original algorithm to determine if one graph is a minor of another, while the Ruby version uses a more simplistic, heavily recursive algorithm to perform the same task. This recursion in the Ruby shows its impact on the timing, as the graphs get more and more complex. While the Java version does not suffer tremendously as the size

of the graphs increase, the Ruby version definitely does. It should also be noted that Ruby as a language is historically not known for its speed, but rather its ease of use and readability. The implementers of Java definitely considered runtime speed. While only an exact port of the Java algorithm to Ruby would allow for a comparison of the actual speed of the languages, it is generally accepted that Java performs faster than Ruby for most tasks.

As expected, the total elapsed time as well as the total time to process all of the graphs increased as the number of graphs increased. It seems only logical that all of the eight-vertex graphs would take longer to process than all of the seven-vertex graphs since there are over 10 times more. Similarly, there are more than 20 times more nine-vertex graphs than eight-vertex graphs, so naturally, one would have expected the time to process all of the graphs to increase as it did.

Even the per graph processing times alone are not all that interesting, since nearly half of all of the graphs processed were classified in a few milliseconds regardless of the implementation. The only per graph times that proved interesting to look at are the maximum times. In Java, as the graphs got more complex, (vertices and edges were added), the results show that the maximum time spent on one graph did increase. It went from a maximum of 17 milliseconds for the set of eight-vertex graphs to a maximum of 692 milliseconds for the set of nine-vertex graphs, an increase of over 40 times. While this increase is substantial, it is nothing compared to what happened to the Ruby maximum times between the same sets of graphs. The Ruby version went from a maximum time of 2.152 seconds on the set of eight-vertex graphs to nearly an hour on the set of nine-vertex graphs. This is an increase of over 1,500 times! As was

mentioned, this timing increase is completely a result of the heavily recursive algorithm to determine if one graph is a minor of another graph in the Ruby version. In an attempt to make the algorithm simpler and less error prone, the timing complexity increased dramatically. It is most complex when the difference in the number of edges between the graph being tested and the minor being searched for is greatest. For example, the slowest graph was 260910, which had 29 edges. The bulk of the time spent classifying that graph was searching for the different minors on nine vertices with 21 and 22 edges. This is simply due to the large number of different ways to remove eight and seven different edges from the 29 edges in the graph.

When looking at the timing of the classification runs, it is natural to ask what would happen if the project had tested 10-vertex graphs. Since there are nearly 60 times more graphs on 10 vertices than on nine vertices, one might expect the elapsed time to be no less than 60 times longer than the elapsed time spent on nine-vertex graphs. With this said, the graphs get more and more complex on 10 vertices, so one would actually expect it to take much more than 60 times longer. Ultimately, the 10-vertex graphs would require a number of days if not weeks in the Java version. As for the Ruby version, it would complete, but it may not be for weeks, if not months.

Although little attention was paid to the performance of the algorithms in the classification efforts, some figures were interesting to look at, especially the maximum time spent on a single graph. The maximum time is especially interesting because as one looks beyond nine vertices, the graphs become more complex and, based on the algorithm designs, will take longer to process. This means that the maximum per graph

processing time will increase for 10-vertex graphs and beyond. Future research needs to consider this fact.

Limitations

With respect to the information collected about intrinsically knotted graphs, it is believed that the classification efforts in this project are completely accurate. While this is true, there are a couple of areas that could be further developed in order to provide more assurance as to the accuracy of the results.

One such area is in relation to the static encoding of the known minor minimal intrinsically knotted graphs. The project included three independent encodings of these known graphs from their pictorial representation to an encoding that the classification software could interpret. While it is believed that the encodings are accurate, they are only as good as the source diagrams that were used, as well as the interpretations of those diagrams. The results rely heavily on these known minor minimal intrinsically knotted graphs and consequently the encodings. Future classifications based on these efforts could have more independent encodings of these graphs from different diagrams and sources. This would help to ensure the accuracy of their encodings.

A second such limitation of this project is that it does not provided an alternative approach to achieve the same classifications. While it does provide two tools with different minor detection algorithms, the Ruby and Java versions share identical functional logic. In both versions, the algorithm applies a series of tests to each graph in a brute force fashion. A future study could provide a second approach to classifying graphs with respect to the property of intrinsic knotting. With two independent methods

to achieve the same goal, results could be compared. Assuming the results are identical, one could gain more assurance that the efforts were accurate.

A final limitation to the approach is that it does not include an encoding or use any known intrinsically knotted graphs larger than nine vertices. Although it should not matter, for completeness the project should have encoded all known intrinsically knotted graphs on more than nine vertices and used those in a Contains Minor and Minor Of Classification test. While it is not anticipated that this would affect the results, it would be beneficial to complete this exercise. These graphs were not encoded because there are dozens of them, many which are the result of *triangle-Y exchanges*, some of which have not been diagramed in any literature. The resulting diagramming and encoding efforts would have been error prone.

While not necessarily a limitation, but certainly a strong assumption, it is worth repeating that this project relies heavily on McKay's *geng* and *showg* tools to create lists of connected graphs on seven, eight, and nine vertices. The project's results on seven and eight vertices indicate that the graph lists seem accurate, since the expected list of intrinsically knotted graphs was known, but it is important to make clear that the project's results rely on complete accuracy in these lists of connected graphs.

CHAPTER VI

CONCLUSION

This project began with the goal of bringing the mathematics community one step closer to determining which graphs fall into the finite set of minor minimal intrinsically knotted graphs. To achieve this goal, a software-based toolset was created, which includes two independent tools that can classify a given graph into one of three distinct states—intrinsically knotted, *not* intrinsically knotted, or indeterminate. Each classification tool processed all distinct, connected graphs on seven, eight, and nine vertices, 273,050 total, and classified them into one of these three distinct states. Both tools classified the graphs identically; they only differed in the amount of time they took to perform the classification. As expected, the tools classified all of the seven and eight-vertex graphs as either intrinsically knotted or *not* intrinsically knotted, with the specific classifications matching previous results from the mathematics literature. In the case of nine-vertex graphs, a literature search showed there were no previous attempts to classify all graphs with respect to the property of intrinsic knotting. This project's classification efforts were the first of its kind. The tools classified all but 32 of the 261,080 connected nine-vertex graphs as either intrinsically knotted or *not* intrinsically knotted. The remaining 32 graphs were classified as indeterminate, meaning the tools were unable to definitively determine an intrinsic knotting classification. This set of 32 graphs likely includes new previously undiscovered minor minimal intrinsically knotted graphs. These

32 graphs are offered to the mathematics community as a starting point for the discovery of new minor minimal intrinsically knotted graphs. In fact, a group of mathematicians have already begun this work.

Future Work

There are quite a few different ways in which this project could be expanded. These possibilities fall into one of three categories—assurance, optimization, and expansion.

Assurance is an important category for future work. Ultimately, in order to publish this project's findings as a part of a mathematical classification of graphs on nine vertices, more certainty must be provided that the results meet the standard of mathematical proof. As mentioned in the limitations section, there are a couple of things that could be done to provide more assurance. First, more independent encodings of the known minor minimal intrinsically knotted graphs would certainly be beneficial. Also, as mentioned, developing a second classification approach would be extremely useful as a way to validate the results. Finally, simply writing thorough unit tests for the existing code base could provide more assurance.

Another area for future consideration would be optimization. While the implementation did provide some pruning of the problem set when it could, it was not concerned with the overall performance. It is believed that with some investment in optimization, the classifications could easily be 10 to 100 times faster. Of the implementations, the project focused more on the Ruby implementation because the minor detection algorithm is much simpler even though it is orders of magnitude slower.

This algorithm, which determines if one graph is a minor of another graph, needs to be optimized. Maybe there is some easy pruning that can be done, or perhaps there is a known minor detection algorithm that can be used? Unfortunately, the research conducted for this project did not reveal any. An improvement in this algorithm will substantially impact the overall running time. Besides the shear number of times this algorithm is invoked, one of the slower parts of its implementation is the call out to the unix *labelg* and *planarg* utilities. While the algorithm only made the calls when the solution was not obvious, much could be done to speed this up. One approach would be to just keep the process open for the duration of the classifications, so that the overhead of starting and stopping is not incurred potentially thousands of times for a single graph. If it could be kept open, then the interactions with it could be faster. An alternative would be to write a Ruby library to talk to the native C *nauty* library directly. This would also eliminate all of the unnecessary opening and closing of thousands of processes. Taking this one step further and re-implementing the entire Ruby project in C would certainly speed things up. If this were done, the *nauty* libraries could be used directly. Ultimately, a port to C would be a welcomed optimization technique.

A much more advanced optimization technique could be implemented where the program is bundled as a distributed application that can run in the background on computers all over the world. A server could pass one graph at a time out to client computers and accept the results. This is a perfect candidate for distributed processing like this because the processing units, individual graphs, are very easy to dissect. This would be modeled very much on the popular SETI@home and Folding@home projects.

An additional thought in regards to optimization is to do some research on the ordering of the classification tests. Although it is not anticipated that this would make a huge impact, especially when it comes to the indeterminate graphs, it could make an incremental improvement. As mentioned in the Methods Chapter, the current order was chosen simply based on opinions and expectations, but that could certainly be investigated further. Obviously code changes and algorithm optimizations are always welcomed, but the above list present focused ways to address optimizations.

The last category for future projects is expansion. The easiest and most obvious is to simply encode all known and newly discovered minor minimal intrinsically knotted graphs and incorporate them into Contains Minor and Minor Of Classification tests. This information alone could lead to the classification of the 32 unknown graphs on nine vertices. Speaking of these graphs, perhaps the best source of further work would be to research these graphs and attempt to determine if any of them are intrinsically knotted or not. One of the main purposes of this project was to try to discover graphs like these and turn them over to the mathematics community for further research. In addition, simply testing all graphs, not just connected graphs would offer a more complete picture of the set of graphs on seven, eight, and nine vertices, although it would not introduce any new indeterminate graphs. Finally, the next obvious step is to push beyond nine vertices. An attempt should be made to classify graphs on 10 or more vertices. Timing limitations did not allow this project to explore this set of graphs. In order to go beyond nine vertices some of the optimizations mentioned above should be done in order to cut down on the timing of the classification runs.

Future work could also benefit from the inclusion of an additional classification test. A test could be added that tested for intrinsic linking in a graph because if a graph is *not* intrinsically linked, then it is *not* intrinsically knotted. By the work of Robertson, Seymour, and Thomas, this amounts to testing if the graph has one of the seven Petersen graphs as a minor [14]. Although this test would not in fact add any information to the current classification efforts, as the 32 indeterminate graphs are all intrinsically linked, this test is very much in line with the current suite of tests and would be a welcomed addition. Also, the classification efforts of all of the graphs on seven, eight, nine vertices did reveal an interesting observation regarding one of the tests. Since an intrinsically knotted graph must be intrinsically linked, and, therefore, contain a Petersen graph minor, a test was made based on the fact that the Petersen graphs all have 15 edges. The Absolute Size Classification says that a graph is *not* intrinsically knotted if it has less than 15 edges, as it would then have no Petersen graph minor. Yet, in all the graphs processed for this project there were no graphs with less than 21 edges that were intrinsically knotted. This could mean that the 15 edge limit could be increased to 21. This is left as an open question for future research.

There are plenty of ways that this project could be taken and improved upon. A good foundation has been laid down on which others can build. Perhaps the most important way that this project could be used though is to analyze the 32 graphs that could not be classified. Indeed, mathematicians are already making use of this. It certainly would be exciting if there is a new class of minor minimal intrinsically knotted graphs in this set, and preliminary investigation by Naimi and others suggest this is very likely.

REFERENCES

REFERENCES

[1] N. Robertson and P.D. Seymour, "Graph Minors. XX. Wagner's Conjecture," *Journal of Combinatorial Theory Series B*, vol. 92, no. 2, pp. 325-357, Nov. 2004.

[2] C.C. Adams, *The Knot Book: An Elementary Introduction To The Mathematical Theory of Knots*. Providence, Rhode Island: American Math Society, 2004.

[3] R. Naimi, private communication.

[4] J. Foisy, "A Newly Recognized Intrinsically Knotted Graph," *Journal of Graph Theory*, vol. 43, no. 3, pp. 199-209, Jul. 2003.

[5] J.H. Conway and C.McA. Gordon, "Knots And Links In Spatial Graphs," *Journal of Graph Theory*, vol. 7, no. 4, pp. 445-453, Winter 1983.

[6] T. Kohara and S. Suzuki, "Some Remarks On Knots And Links In Spatial Graphs," *Knots 90, Osaka, 1990*, pp. 435-445. Berlin: Walter de Gruyter, 1992.

[7] P. Blain, G. Bowlin, T. Fleming, J.Foisy, J. Hendricks, and J. LaCombe, "Some Results On Intrinsically Knotted Graphs," *Journal of Knot Theory and Its Ramifications*, vol. 16, no. 6, pp. 749-760, Aug. 2007.

[8] J. Campbell, T.W. Mattman, R. Ottman, J. Pyzer, M. Rodrigues, and S. Williams, "Intrinsic Knotting And Linking Of Almost Complete Graphs," *Kobe Journal of Mathematics*, accepted for publication.

[9] B. McKay, "Combinatorial Data." Retrieved November 15, 2007 from the World Wide Web: http://cs.anu.edu.au/~bdm/data/graphs.html.

[10] J. Foisy, "Intrinsically Knotted Graphs", *Journal of Graph Theory*, vol. 39, no. 3, pp. 178-187, Mar. 2002.

[11] R. Motwani, R. Raghunathan, and H. Saran, "Constructive Results From Graph Minors: Linkless Embeddings," *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pp. 398-409. Washington DC: IEEE Computer Society, 1988.

[12] T.W. Mattman, R. Ottman, and M. Rodrigues, "Intrinsic Knotting And Linking Of Almost Complete Partite Graphs," preprint, 2003. math.GT/0312176

[13] J. Foisy, "More Intrinsically Knotted Graphs," *Journal of Graph Theory*, vol. 54, no. 2, pp. 115-124, Feb. 2007.

[14] N. Robertson, P. Seymour, R. Thomas, "Sachs' Linkless Embedding Conjecture", *Journal of Combinatorial Theory Series B*, vol, 64, no. 2, pp. 185-227, Jul. 1995.

[15] W. Mader, "Homomorphiesätze Für Graphen," *Mathematische Annalen*, vol. 178, pp. 154-168, 1968.

[16] J. Nešetřil and R. Thomas, "A Note On Spatial Representations Of Graphs," *Commentat. Math. Univ. Carolinae*, vol. 26, pp. 655-659, 1985.

[17] R. Faizullin and A.V. Prolubnikov, "The Direct Algorithm For Solving Of The Graph Isomorphism Problem," eprint arXiv:math/0502251v4, Feb. 2005.

[18] B. McKay, "Practical Graph Isomorphism," *Congressus Numerantium*, vol. 30, pp. 45-87, 1981.

[19] B. McKay, “The Nauty Page.” Retrieved November 15, 2007 from the World Wide Web: http://cs.anu.edu.au/~bdm/nauty/.

[20] J. Hopcroft and R. Tarjan, “Efficient Planarity Testing,” *Journal of the ACM*, vol. 21, no. 4, pp. 549-568, Oct. 1974.

APPENDIX A

THE INTRINSIC KNOTTING TOOLSET

expansion_mapper

```
Description:
       Prints a list of all of the expansions of each of the graphs with respect to each
other. Takes a list of graphs as an argument and looks at each graph with respect to all
of the other graphs. The tool determines which graphs in the supplied list of graphs are
minors and expansions of each other.

Options:
       -f, --infile  The path to the file to read the graphs from.
       -o, --outfile The path to the file to write the results to.
       -e, --edges   A space separated list of edges.

Sample Usage:
       tools/expansion_mapper -f graphs/connected_graphs_9.txt
       tools/expansion_mapper -f graphs/connected_graphs_9.txt 100:200 315 900:*
       tools/expansion_mapper -e 1 2  1 3  0 3  4 5
       tools/expansion_mapper -f graphs/connected_graphs_9.txt -o results/expansions.txt
```

graph_complementor

```
Description:
       Prints the complements to the specified graphs in a form that can be interpreted
by many of the other tools. This tool can be used to pick out only certain graphs from a
file of many graphs.

Options:
       -f, --infile  The path to the file to read the graphs from.
       -o, --outfile The path to the file to write the graphs to.
       -e, --edges   A space separated list of edges.

Sample Usage:
       tools/graph_complementor -f graphs/connected_graphs_9.txt
       tools/graph_complementor -f graphs/connected_graphs_9.txt 100:200 315 900:*
       tools/graph_complementor -e 1 2  1 3  0 3  4 5
       tools/graph_complementor -f graphs/connected_graphs_9.txt -o
graphs/complements_9.txt
```

graph_finder

```
Description:
       Prints the specified graphs in a form that can be interpreted by many of the other
tools. This tool can be used to pick out only certain graphs from a file of many graphs.

Options:
       -f, --infile  The path to the file to read the graphs from.
       -o, --outfile The path to the file to write the graphs to.
       -e, --edges   A space separated list of edges.

Sample Usage:
       tools/graph_finder -f graphs/connected_graphs_9.txt
       tools/graph_finder -f graphs/connected_graphs_9.txt 100:200 315 900:*
       tools/graph_finder -e 1 2  1 3  0 3  4 5
       tools/graph_finder -f graphs/connected_graphs_9.txt -o graphs/copy_9.txt
```

graph_generator

```
Description:
       Generates listings of all of the connected graphs on the given number of vertices.
The graph listings are in a form that can be interpreted by many of the other tools.
```

```
Options:
       -o, --outfile The path to the file to write the graphs to.

Sample Usage:
       tools/graph_generator 8
       tools/graph_generator 9 -o graphs/connected_graphs_9.txt
```

ik_classifier

```
Description:
       Classifies the supplied graphs with respect to the property of intrinsic knotting
using the Ruby implementation. The results of the classification efforts will be printed
to standard output or the supplied output file.

Options:
       -f, --infile  The path to the file to read the graphs from.
       -o, --outfile The path to the file to write the results to.
       -e, --edges   A space separated list of edges.

Sample Usage:
       tools/ik_classifier -f graphs/connected_graphs_9.txt
       tools/ik_classifier -f graphs/connected_graphs_9.txt 100:200 315 900:*
       tools/ik_classifier -e 1 2  1 3  0 3  4 5
       tools/ik_classifier -f graphs/connected_graphs_9.txt -o results/ruby_9.txt
```

ik_summarizer

```
Description:
       Summarizes the results from a run of ik_classifier or java_ik_classifier. The
summary will include classification results, as well as timing results in a succinct
presentation.

Options:
       -f, --infile  The path to the results file from a run of ik_classifier or
java_ik_classifier.
       -o, --outfile The path to the file to write the summarized results to.

Sample Usage:
       tools/ik_summarizer -f results/ruby_9.txt
       tools/ik_summarizer -f results/java_9.txt –o results/java_9_summarized.txt
```

installer

```
Description:
       Installs the necessary binaries and creates the needed files and directories in
order for the tools to work. This must be run before using any of the other tools.

Options:
       none

Sample Usage:
       tools/installer
```

java_ik_classifier

```
Description:
       Uses the Java code to classify the supplied graphs with respect to the property of
intrinsic knotting.  This is the only tool that references the Java code.  (If the more
robust options of ik_classifier are desired which allow the edges to be specified or
specific graphs, then first run those options through graph_finder and write the results
to a file.  Feed that file into the java_ik_classifier with the -f option.)

Options:
       -f, --infile  The path to the file to read the graphs from.
       -o, --outfile The path to the file to write the results to.

Sample Usage:
       tools/java_ik_classifier -f graphs/connected_graphs_9.txt
       tools/java_ik_classifier -f graphs/connected_graphs_9.txt -o results/java_9.txt
```

APPENDIX B

SUMMARIZED CLASSIFICATION RESULTS

Seven-Vertex Graphs—Java

```
Command Run: tools/ik_summarizer –f results/java_7.txt –o results/java_7_summary.txt

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~ java -jar knotfinder.jar graphs/connected_graphs_7.txt results/java_7.txt ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Classification Run At:       Sun Aug 24 18:58:05 -0700 2008
Classification Completed At: Sun Aug 24 18:58:05 -0700 2008

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ CLASSIFICATION SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graphs Processed
  Total: 853

States
  Not Intrinsically Knotted: 852     (99.88%)
  Intrinsically Knotted:     1       (0.12%)
  Indeterminate:             0       (0.00%)

Classification Tests
  AbsoluteSizeClassification:         773     (90.62%)
  RelativeSizeClassification:         1       (0.12%)
  PlanarityClassification:            79      (9.26%)
  Unclassified:                       0       (0.00%)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ TIMING SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Elapsed Time
  Total:   79ms

Graph Classifying Time
  Total:   5ms
  Mean:    0ms
  Median:  0ms
  Minimum: 0ms
  Maximum: 1ms


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ INDETERMINATE GRAPHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graph Ids
```

Eight-Vertex Graphs—Java

```
Command Run: tools/ik_summarizer –f results/java_8.txt –o results/java_8_summary.txt

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~ java -jar knotfinder.jar graphs/connected_graphs_8.txt results/java_8.txt ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Classification Run At:       Sun Aug 24 18:58:20 -0700 2008
Classification Completed At: Sun Aug 24 18:58:22 -0700 2008

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ CLASSIFICATION SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graphs Processed
```

```
  Total: 11117

States
  Not Intrinsically Knotted: 11095   (99.80%)
  Intrinsically Knotted:     22      (0.20%)
  Indeterminate:             0       (0.00%)

Classification Tests
  ContainsMinorK7Classification:      12      (0.11%)
  AbsoluteSizeClassification:         5850    (52.62%)
  ContainsMinorH8Classification:      5       (0.04%)
  ContainsMinorK3311Classification:   1       (0.01%)
  RelativeSizeClassification:         4       (0.04%)
  PlanarityClassification:            5245    (47.18%)
  Unclassified:                       0       (0.00%)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ TIMING SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Elapsed Time
  Total:   1s916ms

Graph Classifying Time
  Total:   1s633ms
  Mean:    0ms
  Median:  0ms
  Minimum: 0ms
  Maximum: 17ms


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ INDETERMINATE GRAPHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graph Ids
```

Nine-Vertex Graphs—Java

```
Command Run: tools/ik_summarizer –f results/java_9.txt –o results/java_9_summary.txt

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~ java -jar knotfinder.jar graphs/connected_graphs_9.txt results/java_9.txt ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Classification Run At:       Sun Aug 24 18:58:31 -0700 2008
Classification Completed At: Sun Aug 24 19:16:24 -0700 2008

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ CLASSIFICATION SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graphs Processed
  Total: 261080

States
  Not Intrinsically Knotted: 259055  (99.22%)
  Intrinsically Knotted:     1993    (0.76%)
  Indeterminate:             32      (0.01%)

Classification Tests
  ContainsMinorK7Classification:      1272    (0.49%)
  ContainsMinorB9Classification:      7       (0.00%)
  AbsoluteSizeClassification:         29913   (11.46%)
  ContainsMinorA9Classification:      6       (0.00%)
  ContainsMinorH8Classification:      543     (0.21%)
  ContainsMinorK3311Classification:   39      (0.01%)
  ContainsMinorH9Classification:      32      (0.01%)
  RelativeSizeClassification:         45      (0.02%)
  ContainsMinorF9Classification:      49      (0.02%)
  PlanarityClassification:            229142  (87.77%)
  Unclassified:                       32      (0.01%)
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ TIMING SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Elapsed Time
  Total:   17m53s302ms

Graph Classifying Time
  Total:   17m46s771ms
  Mean:    4ms
  Median:  2ms
  Minimum: 0ms
  Maximum: 692ms


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ INDETERMINATE GRAPHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graph Ids
  243680
  243683
  243745
  244064
  244065
  244632
  245103
  245113
  245195
  245238
  245239
  245246
  245605
  245608
  245677
  255220
  255244
  255247
  255925
  256305
  256338
  256363
  256368
  256372
  256510
  260624
  260908
  260909
  260910
  260920
  260922
  260928
```

Seven-Vertex Graphs—Ruby

```
Command Run: tools/ik_summarizer –f results/ruby_7.txt –o results/ruby_7_summary.txt

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~ tools/ik_classifier -f graphs/connected_graphs_7.txt -o results/ruby_7.txt ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Classification Run At:       Sun Aug 24 19:17:33 -0700 2008
Classification Completed At: Sun Aug 24 19:17:33 -0700 2008

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ CLASSIFICATION SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graphs Processed
  Total: 853

States
  Not Intrinsically Knotted: 852     (99.88%)
```

```
  Intrinsically Knotted:     1       (0.12%)
  Indeterminate:             0       (0.00%)

Classification Tests
  AbsoluteSizeClassification:         773     (90.62%)
  RelativeSizeClassification:         1       (0.12%)
  PlanarityClassification:            79      (9.26%)
  Unclassified:                       0       (0.00%)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ TIMING SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Elapsed Time
  Total:   505ms

Graph Classifying Time
  Total:   388ms
  Mean:    0ms
  Median:  0ms
  Minimum: 0ms
  Maximum: 6ms


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ INDETERMINATE GRAPHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graph Ids
```

Eight-Vertex Graphs—Ruby

```
Command Run: tools/ik_summarizer –f results/ruby_8.txt –o results/ruby_8_summary.txt

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~ tools/ik_classifier -f graphs/connected_graphs_8.txt -o results/ruby_8.txt ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Classification Run At:       Sun Aug 24 19:17:42 -0700 2008
Classification Completed At: Sun Aug 24 19:18:18 -0700 2008

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ CLASSIFICATION SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graphs Processed
  Total: 11117

States
  Not Intrinsically Knotted: 11095   (99.80%)
  Intrinsically Knotted:     22      (0.20%)
  Indeterminate:             0       (0.00%)

Classification Tests
  ContainsMinorK7Classification:      12      (0.11%)
  AbsoluteSizeClassification:         5850    (52.62%)
  ContainsMinorH8Classification:      5       (0.04%)
  ContainsMinorK3311Classification:   1       (0.01%)
  RelativeSizeClassification:         4       (0.04%)
  PlanarityClassification:            5245    (47.18%)
  Unclassified:                       0       (0.00%)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ TIMING SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Elapsed Time
  Total:   36s151ms

Graph Classifying Time
  Total:   33s887ms
  Mean:    3ms
  Median:  0ms
  Minimum: 0ms
```

```
  Maximum: 2s152ms


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ INDETERMINATE GRAPHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graph Ids
```

Nine-Vertex Graphs—Ruby

```
Command Run: tools/ik_summarizer –f results/ruby_9.txt –o results/ruby_9_summary.txt

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~ tools/ik_classifier -f graphs/connected_graphs_9.txt -o results/ruby_9.txt ~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Classification Run At:       Sun Aug 24 19:18:34 -0700 2008
Classification Completed At: Sun Aug 24 22:27:24 -0700 2008

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ CLASSIFICATION SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graphs Processed
  Total: 261080

States
  Not Intrinsically Knotted: 259055  (99.22%)
  Intrinsically Knotted:     1993    (0.76%)
  Indeterminate:             32      (0.01%)

Classification Tests
  ContainsMinorK7Classification:      1272    (0.49%)
  ContainsMinorB9Classification:      7       (0.00%)
  AbsoluteSizeClassification:         29913   (11.46%)
  ContainsMinorA9Classification:      6       (0.00%)
  ContainsMinorH8Classification:      543     (0.21%)
  ContainsMinorK3311Classification:   39      (0.01%)
  ContainsMinorH9Classification:      32      (0.01%)
  RelativeSizeClassification:         45      (0.02%)
  ContainsMinorF9Classification:      49      (0.02%)
  PlanarityClassification:            229142  (87.77%)
  Unclassified:                       32      (0.01%)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ TIMING SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Elapsed Time
  Total:   3h8m49s326ms

Graph Classifying Time
  Total:   3h7m31s264ms
  Mean:    43ms
  Median:  5ms
  Minimum: 0ms
  Maximum: 55m8s123ms


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ INDETERMINATE GRAPHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graph Ids
  243680
  243683
  243745
  244064
  244065
  244632
  245103
  245113
  245195
```

245238
245239
245246
245605
245608
245677
255220
255244
255247
255925
256305
256338
256363
256368
256372
256510
260624
260908
260909
260910
260920
260922
260928

APPENDIX C

THE 32 INDETERMINATE GRAPHS ON NINE VERTICES

Graph Listings

```
Command Run: tools/graph_finder -f graphs/connected_graphs_9.txt 243680 243683 243745
244064 244065 244632 245103 245113 245195 245238 245239 245246 245605 245608 245677
255220 255244 255247 255925 256305 256338 256363 256368 256372 256510 260624 260908
260909 260910 260920 260922 260928

Graph 243680, order 9.
9 21
0 3  0 4  0 5  0 6  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 6  3 7  3 8  4 7
4 8  5 7  5 8

Graph 243683, order 9.
9 22
0 3  0 4  0 5  0 6  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 6  3 7  3 8
4 7  4 8  5 7  5 8

Graph 243745, order 9.
9 23
0 3  0 4  0 5  0 6  0 7  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 6  3 7  3 8
4 7  4 8  5 7  5 8  6 8

Graph 244064, order 9.
9 23
0 3  0 4  0 5  0 6  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 6  3 7  3 8  4 6
4 7  4 8  5 7  5 8  6 8

Graph 244065, order 9.
9 23
0 3  0 4  0 5  0 6  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 6  3 7  3 8
4 6  4 7  4 8  5 7  7 8

Graph 244632, order 9.
9 22
0 3  0 4  0 6  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7  4 7
5 8  6 7  6 8  7 8

Graph 245103, order 9.
9 22
0 3  0 4  0 5  0 6  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7  3 8
4 7  4 8  5 8  6 8

Graph 245113, order 9.
9 23
0 3  0 4  0 5  0 6  0 7  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7
3 8  4 7  4 8  5 8  6 8

Graph 245195, order 9.
9 25
0 3  0 4  0 5  0 6  0 7  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7
3 8  4 6  4 7  4 8  5 7  5 8  6 8

Graph 245238, order 9.
9 24
0 3  0 4  0 5  0 7  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7
3 8  4 6  4 7  4 8  5 6  5 8

Graph 245239, order 9.
9 24
```

```
0 3  0 4  0 5  0 7  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7
3 8  4 6  4 7  4 8  5 6  6 8

Graph 245246, order 9.
9 24
0 3  0 4  0 5  0 7  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7  3 8
4 6  4 7  4 8  5 6  5 8  7 8

Graph 245605, order 9.
9 23
0 3  0 4  0 6  0 7  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7
3 8  4 5  4 7  5 8  7 8

Graph 245608, order 9.
9 23
0 3  0 4  0 6  0 7  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7
4 5  4 7  5 8  6 8  7 8

Graph 245677, order 9.
9 26
0 3  0 4  0 6  0 7  0 8  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6  3 7
3 8  4 5  4 7  4 8  5 7  5 8  6 7  6 8

Graph 255220, order 9.
9 24
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  3 6  3 7
4 6  4 7  4 8  5 8  6 8  7 8

Graph 255244, order 9.
9 24
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 4  1 5  1 6  1 8  2 4  2 5  2 6  2 7  2 8  3 6  3 7
3 8  4 6  4 7  4 8  5 7  5 8

Graph 255247, order 9.
9 24
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 4  1 5  1 6  1 8  2 4  2 5  2 6  2 7  3 6  3 7  4 6
4 7  4 8  5 7  5 8  6 8  7 8

Graph 255925, order 9.
9 23
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 5  1 6  1 7  2 4  2 5  2 6  2 7  2 8  3 4  3 6  3 8
4 7  5 8  6 7  6 8  7 8

Graph 256305, order 9.
9 24
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 4  3 5
3 7  4 6  4 7  4 8  5 8  7 8

Graph 256338, order 9.
9 27
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 4  3 5
3 7  3 8  4 6  4 7  4 8  5 7  5 8  6 7  6 8

Graph 256363, order 9.
9 25
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 4  3 5
3 7  3 8  4 6  4 7  4 8  5 6  5 8

Graph 256368, order 9.
9 25
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 4  3 5
3 7  4 6  4 7  4 8  5 6  5 8  7 8

Graph 256372, order 9.
9 25
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 5  1 6  1 7  2 4  2 5  2 6  2 7  2 8  3 4  3 5  3 7
3 8  4 6  4 7  5 6  5 8  6 8  7 8

Graph 256510, order 9.
```

```
9 27
0 3  0 4  0 5  0 6  0 7  0 8  1 3  1 4  1 5  1 6  1 7  1 8  2 3  2 4  2 5  2 6  2 7  2 8
3 5  3 6  3 7  4 6  4 7  4 8  5 7  5 8  6 8

Graph 260624, order 9.
9 27
0 2  0 4  0 5  0 6  0 7  0 8  1 3  1 4  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5  3 6
3 7  3 8  4 5  4 7  4 8  5 7  5 8  6 7  6 8

Graph 260908, order 9.
9 28
0 2  0 3  0 4  0 6  0 7  0 8  1 3  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5
3 6  3 7  3 8  4 6  4 7  4 8  5 6  5 7  5 8  6 8

Graph 260909, order 9.
9 28
0 2  0 3  0 4  0 6  0 7  0 8  1 3  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5
3 6  3 7  3 8  4 6  4 7  4 8  5 6  5 7  6 8  7 8

Graph 260910, order 9.
9 29
0 2  0 3  0 4  0 6  0 7  0 8  1 3  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5
3 6  3 7  3 8  4 6  4 7  4 8  5 6  5 7  5 8  6 8  7 8

Graph 260920, order 9.
9 28
0 2  0 3  0 4  0 5  0 7  0 8  1 3  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5
3 6  3 7  3 8  4 6  4 7  4 8  5 7  5 8  6 7  6 8

Graph 260922, order 9.
9 26
0 2  0 3  0 4  0 5  0 6  0 8  1 3  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5
3 6  3 7  3 8  4 6  4 7  4 8  5 7  5 8

Graph 260928, order 9.
9 27
0 2  0 3  0 4  0 5  0 6  0 7  1 3  1 4  1 5  1 6  1 7  1 8  2 4  2 5  2 6  2 7  2 8  3 5
3 6  3 7  3 8  4 6  4 7  4 8  5 7  5 8  6 8
```

Graph Complement Listings

```
Command Run: tools/graph_complementor -f graphs/connected_graphs_9.txt 243680 243683
243745 244064 244065 244632 245103 245113 245195 245238 245239 245246 245605 245608
245677 255220 255244 255247 255925 256305 256338 256363 256368 256372 256510 260624
260908 260909 260910 260920 260922 260928

Graph 243680_complement, order 9.
9 15
0 1  0 2  0 7  0 8  1 2  1 3  2 3  3 4  3 5  4 5  4 6  5 6  6 7  6 8  7 8

Graph 243683_complement, order 9.
9 14
0 1  0 2  0 7  1 2  1 3  2 3  3 4  3 5  4 5  4 6  5 6  6 7  6 8  7 8

Graph 243745_complement, order 9.
9 13
0 1  0 2  0 8  1 2  1 3  2 3  3 4  3 5  4 5  4 6  5 6  6 7  7 8

Graph 244064_complement, order 9.
9 13
0 1  0 2  0 7  0 8  1 2  1 3  2 3  3 4  3 5  4 5  5 6  6 7  7 8

Graph 244065_complement, order 9.
9 13
0 1  0 2  0 7  1 2  1 3  2 3  3 4  3 5  4 5  5 6  5 8  6 7  6 8

Graph 244632_complement, order 9.
9 14
0 1  0 2  0 5  0 7  1 2  1 3  2 3  3 4  3 8  4 5  4 6  4 8  5 6  5 7
```

```
Graph 245103_complement, order 9.
9 14
0 1  0 2  0 7  0 8  1 2  1 3  2 3  3 4  4 5  4 6  5 6  5 7  6 7  7 8

Graph 245113_complement, order 9.
9 13
0 1  0 2  0 8  1 2  1 3  2 3  3 4  4 5  4 6  5 6  5 7  6 7  7 8

Graph 245195_complement, order 9.
9 11
0 1  0 2  0 8  1 2  1 3  2 3  3 4  4 5  5 6  6 7  7 8

Graph 245238_complement, order 9.
9 12
0 1  0 2  0 6  1 2  1 3  2 3  3 4  4 5  5 7  6 7  6 8  7 8

Graph 245239_complement, order 9.
9 12
0 1  0 2  0 6  1 2  1 3  2 3  3 4  4 5  5 7  5 8  6 7  7 8

Graph 245246_complement, order 9.
9 12
0 1  0 2  0 6  0 8  1 2  1 3  2 3  3 4  4 5  5 7  6 7  6 8

Graph 245605_complement, order 9.
9 13
0 1  0 2  0 5  1 2  1 3  2 3  3 4  4 6  4 8  5 6  5 7  6 7  6 8

Graph 245608_complement, order 9.
9 13
0 1  0 2  0 5  1 2  1 3  2 3  3 4  3 8  4 6  4 8  5 6  5 7  6 7

Graph 245677_complement, order 9.
9 10
0 1  0 2  0 5  1 2  1 3  2 3  3 4  4 6  5 6  7 8

Graph 255220_complement, order 9.
9 12
0 1  0 2  1 2  2 3  2 8  3 4  3 5  3 8  4 5  5 6  5 7  6 7

Graph 255244_complement, order 9.
9 12
0 1  0 2  1 2  1 7  2 3  3 4  3 5  4 5  5 6  6 7  6 8  7 8

Graph 255247_complement, order 9.
9 12
0 1  0 2  1 2  1 7  2 3  2 8  3 4  3 5  3 8  4 5  5 6  6 7

Graph 255925_complement, order 9.
9 13
0 1  0 2  1 2  1 4  1 8  2 3  3 5  3 7  4 5  4 6  4 8  5 6  5 7

Graph 256305_complement, order 9.
9 12
0 1  0 2  1 2  1 4  2 3  3 6  3 8  4 5  5 6  5 7  6 7  6 8

Graph 256338_complement, order 9.
9 9
0 1  0 2  1 2  1 4  2 3  3 6  4 5  5 6  7 8

Graph 256363_complement, order 9.
9 11
0 1  0 2  1 2  1 4  2 3  3 6  4 5  5 7  6 7  6 8  7 8

Graph 256368_complement, order 9.
9 11
0 1  0 2  1 2  1 4  2 3  3 6  3 8  4 5  5 7  6 7  6 8
```

```
Graph 256372_complement, order 9.
9 11
0 1  0 2  1 2  1 4  1 8  2 3  3 6  4 5  4 8  5 7  6 7

Graph 256510_complement, order 9.
9 9
0 1  0 2  1 2  3 4  3 8  4 5  5 6  6 7  7 8

Graph 260624_complement, order 9.
9 9
0 1  0 3  1 2  1 5  2 3  3 4  4 6  5 6  7 8

Graph 260908_complement, order 9.
9 8
0 1  0 5  1 2  2 3  3 4  4 5  6 7  7 8

Graph 260909_complement, order 9.
9 8
0 1  0 5  1 2  2 3  3 4  4 5  5 8  6 7

Graph 260910_complement, order 9.
9 7
0 1  0 5  1 2  2 3  3 4  4 5  6 7

Graph 260920_complement, order 9.
9 8
0 1  0 6  1 2  2 3  3 4  4 5  5 6  7 8

Graph 260922_complement, order 9.
9 10
0 1  0 7  1 2  2 3  3 4  4 5  5 6  6 7  6 8  7 8

Graph 260928_complement, order 9.
9 9
0 1  0 8  1 2  2 3  3 4  4 5  5 6  6 7  7 8
```

Expansion Listings

```
Command Run: tools/expansion_mapper -f graphs/connected_graphs_9.txt 243680 243683 243745
244064 244065 244632 245103 245113 245195 245238 245239 245246 245605 245608 245677
255220 255244 255247 255925 256305 256338 256363 256368 256372 256510 260624 260908
260909 260910 260920 260922 260928

243680: 243683
243683: 243745, 244064, 245605, 245608
243745: 245238, 245246
244064: 245246, 256305
244065: 245239, 245246, 255247, 256305
244632: 244064, 244065, 255925
245103: 244065, 245113, 245608
245113: 245238, 245239
245195: 260922
245238: 245195, 256363, 256368
245239: 245195, 256363
245246: 245195, 256368, 256372
245605: 245238, 256305
245608: 245238, 245246, 255244, 256305
245677: 256338, 260624
255220:
255244: 256368
255247: 256363, 256372
255925: 255220, 255247, 256305
256305: 256363, 256368, 256372
256338: 260909, 260920
256363: 260922
256368: 260922
256372: 260922
256510: 260908
```

```
260624: 260909
260908: 260910
260909: 260910
260910:
260920:
260922: 260928
260928:
```

APPENDIX D

expansion_mapper

```
#!/usr/bin/env ruby

# For every graph provided, a list of the other graphs that are direct
# expansions of that graph will be determined.  The term 'direct' in this
# case means that there is no other graph that could be an 'intermediate'
# between these expansions.  The results will be in the form:
#
# 1: 2, 4
# 2: 3
# 3: 5
# 4: 6
# 5:
# 6:
#
# In this example, 2 and 4 are direct expansions but the full set of
# expansions from 1 are 2, 3, 4, 5, 6 since 3 is an expansion of 2,
# 5 is an expansion of 3, and 6 is an expansion of 4.  This form is more
# succinct than the complete list of all expansions for each graph, but
# conveys the same information.
#
# This algorithm is optimized by taking advantage of the transitive nature
# of expansions which means that if A is an expansion of B and B is an
# expansion of C, then we know that A is an expansion of C.
#
# Sending a USR1 signal will print the current progress to stderr.
ENV['PATH'] += ":#{File.join(File.dirname(__FILE__), '..', 'nauty')}"
$LOAD_PATH.unshift(File.join(File.dirname(__FILE__), '..', 'ruby', 'lib'))

require 'standard_options_parser'

Signal.trap("USR1") { print_expansions($stderr) }

@graphs     = []
@expansions = {}

while graph = @graph_parser.next
  @graphs            << graph
  @expansions[graph] =  []
end

# Will only print the direct expansions
def print_expansions(output_stream)
  @graphs.each do |minor|
    expansions = @expansions[minor].collect{|e| e.name}.sort.join(', ')
    output_stream.puts "#{minor.name}: #{expansions}"
  end
end

# All of the expansions for a given graph, direct (no intermediate jumps) and
# indirect (intermediate jump through another graph)
def expansions(minor_graph)
  direct   = @expansions[minor_graph]
  indirect = direct.collect{|exp| expansions(exp)}.flatten
  all      = (direct + indirect).uniq

  all.sort_by{|graph| graph.name.to_i}
end
```

```
min_size   = @graphs.collect{|graph| graph.size}.min
max_size   = @graphs.collect{|graph| graph.size}.max
difference = max_size - min_size

(0..difference).each do |size_diff|
  @graphs.each do |source|
    @graphs.select{|g| (source.size + size_diff) == g.size}.each do |test|
      unless expansions(source).include?(test)
        @expansions[source] << test if source.minor_of?(test, false)
      end
    end
  end
end

print_expansions(@out)
```

graph_complementor

```
#!/usr/bin/env ruby

# Print the complement of every graph provided in a parsable form for
# the tools.
$LOAD_PATH.unshift(File.join(File.dirname(__FILE__), '..', 'ruby', 'lib'))

require 'standard_options_parser'

while graph = @graph_parser.next
  @out.puts "\n#{graph.complement}"
end
```

graph_finder

```
#!/usr/bin/env ruby

# Print every graph provided in a parsable form for the tools.
$LOAD_PATH.unshift(File.join(File.dirname(__FILE__), '..', 'ruby', 'lib'))

require 'standard_options_parser'

while graph = @graph_parser.next
  @out.puts "\n#{graph}"
end
```

graph_generator

```
#!/usr/bin/env ruby

# Creates all of the connected graphs for the given order.  Should
# be used with the -o <output file> option because there is information
# printed to stderr that is not a part of the graph files.  The format
# created by this generator is the format that can be used for the other
# tools that can read from graph files as input.
ENV['PATH'] += ":#{File.join(File.dirname(__FILE__), '..', 'nauty')}"

require 'optparse'

OPTIONS = {}

OptionParser.new do |opts|
  opts.banner  = "Usage: #{$0} <order> [-o outfile]\n"

  opts.on("-o", "--outfile STRING", "The output file path.") do |filepath|
    OPTIONS[:outfile] = filepath
  end
end.parse!

begin
  @out = OPTIONS[:outfile] ? File.new(OPTIONS[:outfile], 'w') : $stdout
rescue => e
  $stderr.puts e.message
```

```
  exit 0
end

unless ARGV.size == 1
  $stderr.puts "Must supply the graph order as an argument."
  exit 0
end

@out.puts `geng -c #{ARGV.first} | showg -eF -l0`
```

ik_classifier

```
#!/usr/bin/env ruby

# Determine if the graphs provided are intrinsically knotted.  If they
# cannot be determined 'indeterminate' will be the result.
# If the results are to be processed, it is valuable to output to a file
# with the -o option because there is some information printed to stderr
# that does not belong on the file.
#
# If a kill signal is sent to the process while running, the current
# graph will be completed before killing the entire process.
#
# The output of this process can be interpreted by ik_summarizer
ENV['PATH'] += ":#{File.join(File.dirname(__FILE__), '..', 'nauty')}"
$LOAD_PATH.unshift(File.join(File.dirname(__FILE__), '..', 'ruby', 'lib'))

require 'standard_options_parser'
require 'time'
require 'ik_classification'

@stop_processing = false

Signal.trap("SIGINT")  { @stop_processing = true }
Signal.trap("SIGQUIT") { @stop_processing = true }
Signal.trap("SIGTERM") { @stop_processing = true }

@graph_count = 0
@start_time  = Time.now

@out.puts @start_time.xmlschema(3)
@out.puts COMMAND_ISSUED
@out.puts

while !@stop_processing && graph = @graph_parser.next
  ik, classification, time = IKClassification::Classifier.classify(graph)
  @out.puts "%s, %s, %s, %f" % [graph.name,
                                ik,
                                classification && classification.name,
                                time]
  @graph_count += 1
end

@end_time = Time.now

@out.puts
@out.puts @end_time.xmlschema(3)

$stderr.puts "Processed #{@graph_count} graphs in "      +
             "#{(@end_time - @start_time) / 60.0} mins."
```

ik_summarizer

```
#!/usr/bin/env ruby

# Summarizes the results from a run of ik_classifier or java_ik_classifier.
# Gives basic timing analysis and test classification distributions.  Also
# gives a list of the indeterminate graphs if any were found.
require 'optparse'
```

```
require 'erb'
require 'time'

OPTIONS = {}

OptionParser.new do |opts|
  opts.banner  = "Usage: #{$0} -f <results file path> [options]"
  opts.on("-f", "--infile  STRING", "The results file path.") do |filepath|
    OPTIONS[:infile] = filepath
  end

  opts.on("-o", "--outfile STRING", "The output file path.") do |filepath|
    OPTIONS[:outfile] = filepath
  end
end.parse!

unless OPTIONS[:infile]
  $stderr.puts "A results source file (-f) is required."
  exit 0
end

begin
  @out = OPTIONS[:outfile] ? File.new(OPTIONS[:outfile], 'w') : $stdout
rescue => e
  $stderr.puts e.message
  exit 0
end

def format_elapsed_time(seconds)
  miliseconds  = (seconds * 1000).round
  days         = miliseconds / (1000 * 60 * 60 * 24)
  miliseconds %= 1000 * 60 * 60 * 24
  hours        = miliseconds / (1000 * 60 * 60)
  miliseconds %= 1000 * 60 * 60
  minutes      = miliseconds / (1000 * 60)
  miliseconds %= (1000 * 60)
  seconds      = miliseconds / 1000
  miliseconds %= 1000

  output  = ''
  output << "#{days}d"    unless days    == 0
  output << "#{hours}h"   unless hours   == 0 && output == ''
  output << "#{minutes}m" unless minutes == 0 && output == ''
  output << "#{seconds}s" unless seconds == 0 && output == ''
  output << "#{miliseconds}ms"
  output
end

@results_file        = File.open(OPTIONS[:infile])

@stats               = {}
@stats[:graph_times] = []
@stats[:graph_count] = 0
@stats[:tests]       = {}
@stats[:states]      = {'ik' => [], 'not_ik' => [], 'indeterminate' => []}
@stats[:start_time]  = Time.parse(@results_file.readline)
@stats[:command]     = @results_file.readline.strip

# Read in the blank line
@results_file.readline

while (next_line = @results_file.readline.strip) != ''
  parts = next_line.split(', ')

  @stats[:states][parts[1]] << parts[0]

  unless parts[1] == 'indeterminate'
    @stats[:tests][parts[2]] ||= 0
    @stats[:tests][parts[2]]  += 1
```

```
  end

  @stats[:graph_times] << parts[3].to_f
  @stats[:graph_count] += 1
end

@stats[:end_time] = Time.parse(@results_file.readline)

@results_file.close

@stats[:graph_times].sort!

@stats[:elapsed_time]      = @stats[:end_time] - @stats[:start_time]
@stats[:total_graph_time]  = @stats[:graph_times].inject{|sum, i| sum + i}
@stats[:mean_graph_time]   = @stats[:total_graph_time] / @stats[:graph_count]
@stats[:median_graph_time] = @stats[:graph_times][@stats[:graph_count]/2]
@stats[:min_graph_time]    = @stats[:graph_times][0]
@stats[:max_graph_time]    = @stats[:graph_times][-1]

@out.puts ERB.new(DATA, nil, '%<>').result

__END__

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
<%= size = @stats[:command].size
    if size >= 78
      @stats[:command]
    elsif size >= 76
      " #{@stats[:command]}"
    else
      leading  = (76 - size) / 2
      trailing = leading
      leading += 1 if size % 2 == 1
      "#{'~' * leading} #{@stats[:command]} #{'~' * trailing}"
    end
 %>
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Classification Run At:       <%= @stats[:start_time]  %>
Classification Completed At: <%= @stats[:end_time]    %>

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ CLASSIFICATION SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graphs Processed
  Total: <%= @stats[:graph_count] %>

States
  Not Intrinsically Knotted: <%= "%-7d (%1.2f%%)" %
    [@stats[:states]['not_ik'].size,
     @stats[:states]['not_ik'].size * 100.0 / @stats[:graph_count]] %>
  Intrinsically Knotted:     <%= "%-7d (%1.2f%%)" %
    [@stats[:states]['ik'].size,
     @stats[:states]['ik'].size * 100.0 / @stats[:graph_count]] %>
  Indeterminate:             <%= "%-7d (%1.2f%%)" %
    [@stats[:states]['indeterminate'].size,
     @stats[:states]['indeterminate'].size * 100.0 / @stats[:graph_count]] %>

Classification Tests
<% @stats[:tests].each do |test, count| %>
  <%= "%-35s %-7s (%1.2f%%)" % [test + ':',
                                count,
                                count * 100.0 / @stats[:graph_count]] %>
<% end %>
  <%= "%-35s %-7s (%1.2f%%)" %
      ['Unclassified:',
       @stats[:states]['indeterminate'].size,
       @stats[:states]['indeterminate'].size * 100.0 / @stats[:graph_count]]
   %>
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ TIMING SUMMARY ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Elapsed Time
  Total:   <%= format_elapsed_time(@stats[:elapsed_time])      %>

Graph Classifying Time
  Total:   <%= format_elapsed_time(@stats[:total_graph_time])  %>
  Mean:    <%= format_elapsed_time(@stats[:mean_graph_time])   %>
  Median:  <%= format_elapsed_time(@stats[:median_graph_time]) %>
  Minimum: <%= format_elapsed_time(@stats[:min_graph_time])    %>
  Maximum: <%= format_elapsed_time(@stats[:max_graph_time])    %>


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~ INDETERMINATE GRAPHS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Graph Ids
<% @stats[:states]['indeterminate'].each do |graph_id| %>
  <%= graph_id %>
<% end %>
```

installer

```
#!/usr/bin/env ruby

# This will download the nauty source code, compile it, and extract
# the necessary binaries.  It will also create a the necessary directories as
# well as generate all of the connected graphs up to 9 vertices, placing
# them in the graphs directory.  Finally, some documentation will be generated
# for the docs directory.
$LOAD_PATH.unshift(File.join(File.dirname(__FILE__), '..', 'ruby', 'lib'))

require 'fileutils'
require 'ik_classification'

NAUTY_URL     = "http://cs.anu.edu.au/~bdm/nauty/nauty24b7.tar.gz"
NAUTY_BIN_DIR = File.join(File.dirname(__FILE__), '..', 'nauty')
DOCS_DIR      = File.join(File.dirname(__FILE__), '..', 'docs')
GRAPHS_DIR    = File.join(File.dirname(__FILE__), '..', 'graphs')
RESULTS_DIR   = File.join(File.dirname(__FILE__), '..', 'results')
JAVA_DIR      = File.join(File.dirname(__FILE__), '..', 'java')
RUBY_DIR      = File.join(File.dirname(__FILE__), '..', 'ruby')

def initialize_directories
  [NAUTY_BIN_DIR, DOCS_DIR, GRAPHS_DIR, RESULTS_DIR].each do |dir|
    unless File.exists?(dir)
      puts "\nCreating #{dir} directory..."
      Dir.mkdir(dir)
    end
  end
end

def install_nauty
  unless File.exists?('nauty24b7.tar.gz')
    puts "\nDownloading nauty from #{NAUTY_URL}...\n"
    `curl #{NAUTY_URL} > nauty24b7.tar.gz`
  end

  puts "\nExpanding nauty...\n"
  puts `tar -xzf nauty24b7.tar.gz`
  puts "\nRunning configure...\n"
  puts `cd nauty24b7 && ./configure`
  puts "\nRunning make...\n"
  puts `cd nauty24b7 && make`

  puts "\nCopying nauty binaries to #{NAUTY_BIN_DIR} directory...\n"
  ['geng', 'genbg', 'directg', 'multig', 'genrang', 'copyg', 'labelg',
   'shortg', 'listg', 'showg', 'amtog', 'dretog', 'complg', 'catg',
```

```
   'addedgeg', 'deledgeg', 'newedgeg', 'countg', 'pickg', 'biplabg',
   'planarg'].each do |tool|
    FileUtils.mv("nauty24b7/#{tool}", NAUTY_BIN_DIR)
  end

  puts "\nCopying manual to #{DOCS_DIR} directory...\n"
  FileUtils.mv('nauty24b7/nug.pdf', DOCS_DIR)

  puts "\nRemoving unused files...\n"
  FileUtils.rm_rf('nauty24b7')
  FileUtils.rm('nauty24b7.tar.gz')
end

def compile_java
  puts "\nBuilding the Java tools...\n"
  puts `ant -f #{JAVA_DIR}/build.xml`
end

def generate_graphs
  generator = File.join(File.dirname(__FILE__), 'graph_generator')

  (1..9).each do |order|
    filepath = File.join(GRAPHS_DIR, "connected_graphs_#{order}.txt")
    puts "\nGenerating connected graphs of order #{order} at #{filepath}\n"
    `#{generator} #{order} -o #{filepath}`
  end
end

def generate_docs
  File.open(File.join(DOCS_DIR, 'classification_tests.txt'), 'w') do |file|
    file.write("The following lists the classification tests that are run " +
               "by the ik_classifier program, along with descriptions of "  +
               "each.  The order listed is the order the tests are "        +
               "executed from within the tool.\n\n")

    IKClassification::Classifier::CLASSIFICATIONS.each do |classification|
      file.write("#{classification.name}\n")
      file.write("  #{classification.description}\n\n")
    end
  end

  puts "\nGenerating rdocs for the Ruby source code...\n"
  ruby_docs = File.join(DOCS_DIR, 'rdocs')
  puts `rdoc #{File.join(RUBY_DIR, 'lib')} -aSN -f html -o #{ruby_docs}`

  puts "\nGenerating javadocs for Java source code...\n"
  puts `ant -f #{File.join(JAVA_DIR, 'build.xml')} docs`
end

initialize_directories
install_nauty
compile_java
generate_graphs
generate_docs
```

java_ik_classifier

```
#!/usr/bin/env ruby

# Determines if the graphs provided are intrinsically knotted by running
# the Java based tool.  The only allowed options are a required input file
# (-f) and an optional output file (-o).  Example:
#
# tools/java_ik_classifier -f graphs/connected_graphs_4.txt -o output.txt
#
# If specific graphs are desired or simply a list of edges, then first run
# the list through the 'graph_finder' tool in order to create a file for the
# desired graphs and then run them through this tool.  Example:
#
```

```
# tools/graph_finder -o test_graph.txt -e 0 1  0 2  0 3  1 2
# tools/java_ik_classifier -f test_graph.txt
#
# or
#
# tools/graph_finder -f graphs/connected_graphs_9.txt -o test_graph.txt 20:30
# tools/java_ik_classifier -f test_graph.txt
#
# This tool is merely a wrapper around the java tool at:
# java/dist/lib/knotfinder.jar
require 'optparse'

OPTIONS = {}

OptionParser.new do |opts|
  opts.banner  = "Usage: #{$0} -f <graph_file> [-o outfile]\n"

  opts.on("-f", "--infile  STRING", "The file with the graphs.") do |filepath|
    OPTIONS[:infile] = filepath
  end

  opts.on("-o", "--outfile STRING", "The output file path.") do |filepath|
    OPTIONS[:outfile] = filepath
  end
end.parse!

unless OPTIONS[:infile]
  $stderr.puts "A graph source file (-f) is required."
  exit 0
end

jar_file = File.join(File.dirname(__FILE__),
                     '..',
                     'java',
                     'dist',
                     'knotfinder.jar')

puts `java -jar #{jar_file} #{OPTIONS[:infile]} #{OPTIONS[:outfile]}`
```

APPENDIX E

JAVA SOURCE CODE

AbsoluteSizeClassification.java

```
package ik;

/**
 * A test that can rule out some graphs from being IK.  This is based
 * on Robertson, Seymour, and Thomas' work.
 */
public class AbsoluteSizeClassification implements IKClassification {
  public String classify(Graph graph) {
    return (graph.getSize() < 15) ? IS_NOT_IK : CANNOT_DETERMINE_IK;
  }

  public String getName() {
    return "AbsoluteSizeClassification";
  }

  public String getDescription() {
    return "If there are less than 15 edges then the graph is " +
           "NOT intrinsically knotted.";
  }
}
```

ContainsMinorClassification.java

```
package ik;

/**
 * If a graph contains as a minor or is isomorphic to a known IK graph, then by
 * the definition of a minor, the graph exhibits the intrinsic knotting
 * property.  This classification uses this logic to determine if a graph
 * is intrinsically knotted.
 */
public class ContainsMinorClassification implements IKClassification {
  private Graph ikGraph;

  public ContainsMinorClassification(Graph ikGraph) {
    this.ikGraph = ikGraph;
  }

  public String classify(Graph graph) {
    String result = CANNOT_DETERMINE_IK;

    if (graph.containsMinor(ikGraph)) {
      result = IS_IK;
    }

    return result;
  }

  public String getName() {
    return "ContainsMinor"+ikGraph.getName()+"Classification";
  }

  public String getDescription() {
    return "Any graph that contains the known IK graph " + ikGraph.getName() +
           " as a minor (including isomorphisms) is intrinsically knotted.";
  }
}
```

Graph.java

```
package ik;

import java.util.*;
import java.io.InputStreamReader;
import java.io.BufferedReader;

/**
 * This class is the heart of the project to determine intrinsically
 * knotted graphs.  It is objects of this Graph class that represents the
 * graphs that we are testing.  The classification tests are performed on
 * these objects.
 */
public class Graph {
  private String      name;
  private int         order;
  private boolean[][] data;

  /**
   * Creates a new Graph object.
   *
   * @param name A String name for the graph
   * @param order An integer number of vertices for the graph
   */
  public Graph(String name, int order) {
    this.name  = name;
    this.order = order;
    this.data  = new boolean[order][order];
  }

  /**
   * The name of the graph.
   *
   * @return A String name for the graph.
   */
  public String getName() {
    return name;
  }

  /**
   * Returns the number of vertices in the graph.
   *
   * @return An integer representing the number of vertices in the graph.
   */
  public int getOrder() {
    return order;
  }

  /**
   * Returns the number of edges in the graph.
   *
   * @return An integer representing the number of edges in the graph.
   */
  public int getSize() {
    return edges().size();
  }

  /**
   * Adds the given edge to the Graph.
   *
   * @param fromVertex The from vertex for the edge
   * @param toVertex The to vertex for the edge
   */
  public void addEdge(int fromVertex, int toVertex) {
    setEdge(fromVertex, toVertex, true);
  }
```

```
/**
 * Determines whether or not this graph includes the provided edge.
 *
 * @param fromVertex The from vertex for the edge
 * @param toVertex The to vertex for the edge
 * @return A boolean value indicating whether or not this edge is a part
 *         of the graph.
 */
public boolean hasEdge(int fromVertex, int toVertex) {
  return getEdge(fromVertex, toVertex);
}

/**
 *  The list of edges in this graph.
 *
 * @return A List of edges that are a part of this graph.
 */
public List<int[]> edges() {
  List<int[]> edges = new ArrayList<int[]>(maxSize());

  for (int from = 0; from < getOrder(); from++) {
    for (int to = from + 1; to < getOrder(); to++) {
      if (hasEdge(from, to)) {
        int[] edge = {from, to};
        edges.add(edge);
      }
    }
  }

  return edges;
}

/**
 * Returns a list of vertices that are connected via an edge to the
 * provided vertex.
 *
 * @param fromVertex The vertex we are investigating.
 * @return A List of integers which represent edges from the fromVertex
 *         provided to the vertices returned.
 */
public List<Integer> connectedVertices(int fromVertex) {
  List<Integer> connectedVertices = new ArrayList<Integer>(getOrder());

  for (int toVertex = 0; toVertex < getOrder(); toVertex++) {
    if (fromVertex != toVertex && hasEdge(fromVertex, toVertex)) {
      connectedVertices.add(new Integer(toVertex));
    }
  }

  return connectedVertices;
}

/**
 * Create a new graph with the supplied edge contracted.
 *
 * @param fromVertex The from vertex for the edge.
 * @param toVertex The to vertex for the edge.
 * @return A new Graph with the provided edge removed.
 */
public Graph contractEdge(int fromVertex, int toVertex) {
  Graph minor = null;

  // Cannot contract an edge if we don't have one
  if (hasEdge(fromVertex, toVertex)) {
    minor = new Graph(getName(), getOrder() - 1);
    int      maxVertex    = fromVertex > toVertex ? fromVertex : toVertex;
    int      minVertex    = fromVertex > toVertex ? toVertex   : fromVertex;
    Iterator edgeIterator = edges().iterator();
```

```
    while (edgeIterator.hasNext()) {
      int[] edge    = (int[])edgeIterator.next();
      int   newFrom = -1;
      int   newTo   = -1;

      // The vertex labels for the edge may have changed as a result of
      // one less vertex
      if (edge[0] == maxVertex) {
        newFrom = minVertex;
      } else if (edge[0] > maxVertex) {
        newFrom = edge[0] - 1;
      } else {
        newFrom = edge[0];
      }

      if (edge[1] == maxVertex) {
        newTo = minVertex;
      } else if (edge[1] > maxVertex) {
        newTo = edge[1] - 1;
      } else {
        newTo = edge[1];
      }

      if (newFrom != newTo) {
        minor.addEdge(newFrom, newTo);
      }
    }
  }

  return minor;
}

/**
 * Create a new graph with the vertices provided removed.
 *
 * @param vertices An array of vertices to remove from our graph
 * @return A new Graph without the vertices provided
 */
public Graph removeVertices(int[] vertices) {
  Graph subGraph = new Graph(getName(), getOrder() - vertices.length);

  Iterator edgeIterator = edges().iterator();

  while (edgeIterator.hasNext()) {
    int[] edge = (int[])edgeIterator.next();
    int   from = edge[0];
    int   to   = edge[1];

    // If either of the points is in the vertices array, then the point
    // is thrown out
    boolean disregardEdge = false;
    for (int i = 0; i < vertices.length; i++) {
      if (from == vertices[i] || to == vertices[i]) {
        disregardEdge = true;
      }
    }

    if (!disregardEdge) {
      int origFrom = from;
      int origTo   = to;

      // The vertex labels for the edge may have changed as a result of
      // less vertices
      for (int i = 0; i < vertices.length; i++) {
        if (origFrom > vertices[i]) {
          from--;
        }
        if (origTo > vertices[i]) {
          to--;
```

```
        }
      }

      subGraph.addEdge(from, to);
    }
  }

  return subGraph;
}

/**
 * Determine if our graph contains as a minor the provided graph.
 *
 * @param minor The Graph we are comparing to our graph.
 * @return A boolean value indicating whether or not our graph
 *         contains as a minor the provided graph.
 */
public boolean containsMinor(Graph minor) {
  if (getOrder() < minor.getOrder()) { return false; }
  if (getSize()  < minor.getSize())  { return false; }
  if (containsSubgraph(minor))       { return true;  }

  // Contracting an edge will result in one less edge and one less vertex
  // We check to see if we can afford to lose one edge and one vertex
  if (getSize() <= minor.getSize() || getOrder() <= minor.getOrder()) {
    return false;
  }

  Iterator edgeIterator = edges().iterator();

  // Try contracting each edge, then check again
  while(edgeIterator.hasNext()) {
    int[] edge = (int[])edgeIterator.next();

    Graph newMinor = contractEdge(edge[0], edge[1]);

    if (newMinor.containsMinor(minor)) {
      return true;
    }
  }

  return false;
}

/**
 * Determine if our graph contains the provided graph as a subgraph.
 *
 * @param subGraph The graph we are testing against our own.
 * @return A boolean value indicating whether or not our graph
 *         contains the provided graph as a subgraph.
 */
public boolean containsSubgraph(Graph subGraph) {
  boolean containsSubgraph = false;

  // Has to have at least as many edges and vertices as the subgraph
  // in order to even be considered
  if (getSize()  >= subGraph.getSize() &&
      getOrder() >= subGraph.getOrder()) {
    // Initialize an empty vertexMap from our graph to the subgraph
    int[] vertexMap = new int[subGraph.order];
    for (int i = 0; i < vertexMap.length; i++) {
      vertexMap[i] = -1;
    }

    // Do the work
    containsSubgraph = searchForSubgraph(subGraph, vertexMap, 0);
  }

  return containsSubgraph;
```

```
}

/**
 * Determines if two graphs are isomorphic to each other.
 *
 * @param graph The graph to check to see if we are isomorphic to.
 * @return A boolean value indicating whether or not we are isomorphic.
 */
public boolean isIsomorphicTo(Graph graph) {
  return containsSubgraph(graph)       &&
         getSize()  == graph.getSize() &&
         getOrder() == graph.getOrder();
}

/**
 * Does the recursive work of trying to determine if a mapping from our
 * graph to the subgraph can be found.
 *
 * @param subGraph The Graph that we are trying to determine if we are a
 *                 subgraph of
 * @param vertexMap The current mapping from our vertices to those of the
 *                  subgraph.
 * @param nextIndex The index we will be working with next.
 * @return A boolean value indicating if the mapping 'so far' is a complete
 *         mapping to the subgraph.
 */
private boolean searchForSubgraph(Graph subGraph,
                                  int[] vertexMap,
                                  int   nextIndex) {
  // We first need to verify our current map state and make sure that it
  // is a 'possible' solution for what we know so far
  if (!verifyPartialMapping(subGraph, vertexMap, nextIndex - 1)) {
    return false;
  }

  // If we are trying to work on an index that is past our length of the
  // the vertexMap, that means that we have a valid mapping
  if (vertexMap.length == nextIndex) {
    return true;
  }

  // Walk through each vertex
  for (int currentIndex = 0; currentIndex < this.order; currentIndex++) {
    if (!containsVertex(vertexMap, currentIndex)) {
      // Assume this vertex is correct, and then test it
      vertexMap[nextIndex] = currentIndex;
      if (searchForSubgraph(subGraph, vertexMap, nextIndex+1)) {
        return true;
      }
      else {
        // It wasn't correct, try the next
        vertexMap[nextIndex] = -1;
      }
    }
  }

  return false;
}

/**
 * Checks whether the current mapping that has been constructed for the
 * graph to a possible subgraph 'is possible'.  It just confirms the
 * in progress solution.
 *
 * @param subGraph The Graph that we are trying to determine if we are a
 *                 subgraph of
 * @param vertexMap The current mapping from our vertices to those of the
 *                  subgraph.
 * @param workingIndex The index we are working with currently.
```

```
 * @return A boolean value indicating if the mapping 'so far' is
 *         possible or not.
 */
private boolean verifyPartialMapping(Graph subGraph,
                                     int[] vertexMap,
                                     int   workingIndex) {
  boolean successfulMapping = true;

  if (workingIndex >= 0 && workingIndex < vertexMap.length) {
    int      mappedFromVertex  = vertexMap[workingIndex];
    List     connectedVertices = subGraph.connectedVertices(workingIndex);
    Iterator verticesIterator  = connectedVertices.iterator();

    while (verticesIterator.hasNext() && successfulMapping) {
      int toVertex       = ((Integer)verticesIterator.next()).intValue();
      int mappedToVertex = vertexMap[toVertex];
      if (mappedToVertex >= 0) {
           // If the edge exists in the subgraph but not in our graph via
           // the proposed mapping, then we are done and this mapping doesn't
           // work
        successfulMapping = this.hasEdge(mappedFromVertex, mappedToVertex);
      }
    }
  }

  return successfulMapping;
}

/**
 * Determines if the vertexMap contains the vertex provided.  It is
 * merely a lookup into the array for the value.
 *
 * @param vertexMap The current array of integer vertices.
 * @param vertex    The vertex to check.
 * @return A boolean indicating whether or not the vertex was found.
 */
private boolean containsVertex(int[] vertexMap, int vertex) {
  boolean containsVertex = false;
  int     currentIndex   = 0;

  while ((currentIndex < vertexMap.length) && (!containsVertex)) {
    containsVertex = (vertexMap[currentIndex] == vertex);
    currentIndex++;
  }

  return containsVertex;
}

/**
 * Sets the value for an edge.
 *
 * @param fromVertex The starting vertex of the edge.
 * @param toVertex   The ending vertex of the edge.
 * @param value      The boolean value to set it to, where true indicates
 *                   the edge exists, false indicates that it does not.
 */
private void setEdge(int fromVertex, int toVertex, boolean value) {
  validateEdge(fromVertex, toVertex);

  data[fromVertex][toVertex] = data[toVertex][fromVertex] = value;
}

/**
 * Returns the value set for the edge.
 *
 * @return A boolean result where true means there was an edge, and false
 *         means there was not.
 */
private boolean getEdge(int fromVertex, int toVertex) {
```

```
    validateEdge(fromVertex, toVertex);

    return data[fromVertex][toVertex];
  }

  /**
   * Validates that the edge is legitimate.
   *
   * @throws IllegalArgumentException if it is invalid.
   */
  private void validateEdge(int fromVertex, int toVertex) {
    validateVertex(fromVertex);
    validateVertex(toVertex);

    if (fromVertex == toVertex)
      throw new IllegalArgumentException("Loops are not allowed: " +
                                         fromVertex+" == "+toVertex);
  }

  /**
   * Validates that the vertex is legitimate.
   *
   * @throws IllegalArgumentException if it is invalid.
   */
  private void validateVertex(int vertex) {
    if (vertex < 0 || vertex >= getOrder())
      throw new IllegalArgumentException("Vertex is not in the valid range:" +
                                         " 0 <= "+vertex+" < "+getOrder());
  }

  /**
   * Determines the maximum number of edges that are possible with the given
   * order.  This is used to know how bit to create Arrays.
   *
   * @return An integer representing the max possible size.
   */
  private int maxSize() {
    return (getOrder() * (getOrder() - 1)) / 2;
  }
}
```

GraphConstants.java

```
package ik;

/**
 * These are the graph constants that are used in various classification
 * tests.
 *
 * Note: There are more graphs which could be added here
 *         Foisy identified H, G15, H15, J14, J'14
 *         There are also more Triangle-Y exchanges on > 9 vertices
 */
public class GraphConstants
{
  public static final Graph K5;
  public static final Graph K7;
  public static final Graph K33;
  public static final Graph K3311;
  public static final Graph H8;
  public static final Graph H9;
  public static final Graph F9;
  public static final Graph A9;
  public static final Graph B9;

  static {
    // Create K5
    K5 = new Graph("K5", 5);
    for(int i = 0; i < 5; i++) {
```

```
  for(int j = i+1; j < 5; j++) {
    K5.addEdge(i,j);
  }
}

// Create K7
K7 = new Graph("K7", 7);
for(int i = 0; i < 7; i++) {
  for(int j = i+1; j < 7; j++) {
    K7.addEdge(i,j);
  }
}

// Create K33
K33 = new Graph("K33", 6);
K33.addEdge(0,3);
K33.addEdge(0,4);
K33.addEdge(0,5);
K33.addEdge(1,3);
K33.addEdge(1,4);
K33.addEdge(1,5);
K33.addEdge(2,3);
K33.addEdge(2,4);
K33.addEdge(2,5);

// Create K3311
K3311 = new Graph("K3311", 8);
K3311.addEdge(0,1);
K3311.addEdge(0,3);
K3311.addEdge(0,4);
K3311.addEdge(0,5);
K3311.addEdge(0,7);
K3311.addEdge(1,2);
K3311.addEdge(1,3);
K3311.addEdge(1,4);
K3311.addEdge(1,6);
K3311.addEdge(2,3);
K3311.addEdge(2,4);
K3311.addEdge(2,5);
K3311.addEdge(2,7);
K3311.addEdge(3,4);
K3311.addEdge(3,5);
K3311.addEdge(3,6);
K3311.addEdge(3,7);
K3311.addEdge(4,5);
K3311.addEdge(4,6);
K3311.addEdge(4,7);
K3311.addEdge(5,6);
K3311.addEdge(6,7);

// Create H8
H8 = new Graph("H8", 8);
H8.addEdge(0,1);
H8.addEdge(0,2);
H8.addEdge(0,4);
H8.addEdge(0,5);
H8.addEdge(0,6);
H8.addEdge(0,7);
H8.addEdge(1,2);
H8.addEdge(1,3);
H8.addEdge(1,5);
H8.addEdge(1,7);
H8.addEdge(2,4);
H8.addEdge(2,5);
H8.addEdge(2,6);
H8.addEdge(2,7);
H8.addEdge(3,4);
H8.addEdge(3,6);
H8.addEdge(4,5);
```

```
H8.addEdge(4,7);
H8.addEdge(5,6);
H8.addEdge(5,7);
H8.addEdge(6,7);

// Create H9
H9 = new Graph("H9", 9);
H9.addEdge(0,1);
H9.addEdge(0,2);
H9.addEdge(0,4);
H9.addEdge(0,5);
H9.addEdge(0,7);
H9.addEdge(0,8);
H9.addEdge(1,2);
H9.addEdge(1,3);
H9.addEdge(1,5);
H9.addEdge(1,8);
H9.addEdge(2,4);
H9.addEdge(2,6);
H9.addEdge(2,7);
H9.addEdge(3,4);
H9.addEdge(3,7);
H9.addEdge(4,5);
H9.addEdge(4,8);
H9.addEdge(5,6);
H9.addEdge(5,7);
H9.addEdge(6,8);
H9.addEdge(7,8);

// Create F9
F9 = new Graph("F9", 9);
F9.addEdge(0,1);
F9.addEdge(0,2);
F9.addEdge(0,4);
F9.addEdge(0,5);
F9.addEdge(0,6);
F9.addEdge(0,8);
F9.addEdge(1,2);
F9.addEdge(1,3);
F9.addEdge(1,5);
F9.addEdge(1,8);
F9.addEdge(2,4);
F9.addEdge(2,5);
F9.addEdge(2,6);
F9.addEdge(2,8);
F9.addEdge(3,4);
F9.addEdge(3,6);
F9.addEdge(4,5);
F9.addEdge(4,8);
F9.addEdge(5,7);
F9.addEdge(6,7);
F9.addEdge(7,8);

// Create A9
A9 = new Graph("A9", 9);
A9.addEdge(0,1);
A9.addEdge(0,3);
A9.addEdge(0,5);
A9.addEdge(0,8);
A9.addEdge(1,2);
A9.addEdge(1,4);
A9.addEdge(1,5);
A9.addEdge(1,7);
A9.addEdge(2,4);
A9.addEdge(2,5);
A9.addEdge(2,6);
A9.addEdge(2,8);
A9.addEdge(3,4);
A9.addEdge(3,6);
```

```
    A9.addEdge(4,5);
    A9.addEdge(4,7);
    A9.addEdge(4,8);
    A9.addEdge(5,6);
    A9.addEdge(5,7);
    A9.addEdge(5,8);
    A9.addEdge(6,7);
    A9.addEdge(7,8);

    // Create B9
    B9 = new Graph("B9", 9);
    B9.addEdge(0,1);
    B9.addEdge(0,3);
    B9.addEdge(0,4);
    B9.addEdge(0,6);
    B9.addEdge(0,8);
    B9.addEdge(1,2);
    B9.addEdge(1,3);
    B9.addEdge(1,4);
    B9.addEdge(1,5);
    B9.addEdge(2,3);
    B9.addEdge(2,4);
    B9.addEdge(2,6);
    B9.addEdge(2,8);
    B9.addEdge(3,6);
    B9.addEdge(3,7);
    B9.addEdge(3,8);
    B9.addEdge(4,6);
    B9.addEdge(4,7);
    B9.addEdge(4,8);
    B9.addEdge(5,6);
    B9.addEdge(5,7);
    B9.addEdge(5,8);
  }
}
```

IKClassification.java

```
package ik;

/**
 * This is the interface which represents a classification test
 * for determining if a graph is intrinsically knotted or not.
 */
public interface IKClassification {
    public static final String IS_IK               = "ik";
    public static final String IS_NOT_IK           = "not_ik";
    public static final String CANNOT_DETERMINE_IK = "indeterminate";

    /**
     * This method will classify the given graph into one of three states--
     * ik, not ik, or indeterminate.
     *
     * @param  graph The graph to be classified
     * @return       A String result which is one of the constants IS_IK
     *               IS_NOT_IK or CANNOT_DETERMINE_IK
     */
    public String classify(Graph graph);

    /**
     * The name of the classification test.
     *
     * @return A String name.
     */
    public String getName();

    /**
     * A description of the classification test.
     *
```

```
     * @return A String description.
     */
    public String getDescription();
}
```

KnotFinder.java

```
package ik;

import java.io.*;
import java.util.*;
import java.util.regex.*;
import java.text.DateFormat;
import java.text.SimpleDateFormat;

import static ik.GraphConstants.*;

/**
 * This class will actually perform the classification efforts. It will
 * use all of the classification tests and apply them one at a time to a
 * given graph until a non-indeterminate state can be determined for the graph.
 */
public class KnotFinder
{
  private static final Pattern            DIGIT_REGEX         =
                       Pattern.compile("\\d+");
  private static final SimpleDateFormat   DATE_FORMAT         =
                       new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ");
  public  static final IKClassification[] IK_CLASSIFICATIONS  =
                       {new NullClassification(),
                        new OrderClassification(),
                        new AbsoluteSizeClassification(),
                        new RelativeSizeClassification(),
                        new PlanarityClassification(),
                        new ContainsMinorClassification(K7),
                        new ContainsMinorClassification(H8),
                        new ContainsMinorClassification(H9),
                        new ContainsMinorClassification(F9),
                        new ContainsMinorClassification(K3311),
                        new ContainsMinorClassification(A9),
                        new ContainsMinorClassification(B9),
                        new MinorOfClassification(K7),
                        new MinorOfClassification(H8),
                        new MinorOfClassification(H9),
                        new MinorOfClassification(F9),
                        new MinorOfClassification(K3311),
                        new MinorOfClassification(A9),
                        new MinorOfClassification(B9)};

  private static BufferedReader brGraphs = null;
  private static BufferedWriter bwOut    = null;
  private static String         command  = null;

  /**
   * The main method which drives the classification attempt on the graphs.
   */
  public static void main(String[] args) throws Exception {
    if (args.length == 0) {
      System.out.println("usage: java KnotFinder <graph file> [output file]");
      System.exit(0);
    }

    String infile  = args[0];
    String outfile = args.length == 2 ? args[1] : null;

    // Recreate what the command looked like
    command = "java -jar knotfinder.jar "+infile;

    if (outfile != null) {
```

```
    command += " " + outfile;
  }

  initializeFiles(infile, outfile);
  printHeader();

  Graph currentGraph = nextGraph();

  // Iterate over each graph
  while (currentGraph != null) {
    String           result      = IKClassification.CANNOT_DETERMINE_IK;
    int              testIndex   = 0;
    IKClassification currentTest = null;
    long             startTime   = new Date().getTime();

    // Try each classification test until we find a non indeterminate
    // result
    while ((testIndex < IK_CLASSIFICATIONS.length) &&
           (result == IKClassification.CANNOT_DETERMINE_IK)) {
      currentTest = (IKClassification)IK_CLASSIFICATIONS[testIndex];
      result      = currentTest.classify(currentGraph);
      testIndex++;
    }

    long   endTime = new Date().getTime();
    double seconds = (endTime - startTime) / 1000.0;
    logResult(currentGraph, result, currentTest, seconds);

    currentGraph = nextGraph();
  }

  closeFiles();
}

/**
 * Print the result of the classification attempt to the output stream.
 *
 * @param graph The graph we are working with.
 * @param result The result of the classification.
 * @param lastTest The test which determined that result.
 * @param seconds The number of seconds to perform the classification.
 * @throws IOException if there is an IO error.
 */
private static void logResult(Graph            graph,
                              String           result,
                              IKClassification lastTest,
                              double           seconds) throws IOException {
  String resultString = null;

  if (result == IKClassification.CANNOT_DETERMINE_IK) {
    resultString = graph.getName() + ", "   +
                   result          + ", , " +
                   seconds;
  } else {
    resultString = graph.getName()    + ", " +
                   result             + ", " +
                   lastTest.getName() + ", " +
                   seconds;
  }

  bwOut.write(resultString);
  bwOut.newLine();
  bwOut.flush();
}

/**
 * Reads the next graph from the input file.
 *
 * @return The new Graph that is read, null if the end has been reached.
```

```
 * @throws IOException if there is an IO issue.
 * @throws IllegalArgumentException if there is a problem with the
 *         graph data.
 */
private static Graph nextGraph() throws IOException,
                                        IllegalArgumentException {
  Graph graph = null;

  // Remove any blank lines
  String nextLine = brGraphs.readLine();

  while (nextLine != null && nextLine.trim().length() == 0) {
    nextLine = brGraphs.readLine();
  }

  // Parse the next graph
  if (nextLine != null) {
    String titleLine = nextLine;
    String descLine  = brGraphs.readLine();
    String edges     = brGraphs.readLine();

    // Are there multiple lines of edges?
    nextLine = brGraphs.readLine();
    while (nextLine != null && nextLine.trim().length() != 0) {
      edges   += " "+nextLine;
      nextLine = brGraphs.readLine();
    }

    // The title line looks like "Graph 3, order 8." so pull
    // out the numbers
    Matcher match = DIGIT_REGEX.matcher(titleLine);
    match.find();
    String name  = match.group();
    match.find();
    int order = Integer.parseInt(match.group());

    graph = new Graph(name, order);
    addAllEdges(graph, edges);
  }

  return graph;
}

/**
 * Adds all of the edges to the graph object from the 'edges' String
 * which was read from a file.
 *
 * @param graph The current graph we are working with.
 * @param edges The String line from the file which represents the edges.
 */
private static void addAllEdges(Graph graph, String edges) {
  StringTokenizer stEdges = new StringTokenizer(edges);

  while (stEdges.hasMoreElements()) {
    int fromVert = Integer.parseInt(stEdges.nextToken());
    int toVert   = Integer.parseInt(stEdges.nextToken());
    graph.addEdge(fromVert, toVert);
  }
}

/**
 * Opens the input and output files.  If an output file isn't supplied,
 * then output goes to stdout.
 *
 * @param graphFilePath The filepath where the graphs will be read from.
 * @param outputFilePath The filepath for the output file (can be null).
 * @throws IOException if there is an IO issue.
 */
private static void initializeFiles(String graphFilePath,
```

```
                                      String outputFilePath)
                                     throws IOException {
    brGraphs = new BufferedReader(new FileReader(graphFilePath));

    if (outputFilePath == null) {
      bwOut = new BufferedWriter(new PrintWriter(System.out));
    } else {
      bwOut = new BufferedWriter(new FileWriter(outputFilePath));
    }
  }

  /**
   * Closes the input and output files.
   */
  private static void closeFiles() throws IOException {
    bwOut.write("\n"+DATE_FORMAT.format(new Date())+"\n");

    brGraphs.close();
    bwOut.close();
  }

  /**
   * Prints the first line of the output
   */
  private static void printHeader() throws IOException {
    bwOut.write(DATE_FORMAT.format(new Date())+"\n");
    bwOut.write(command+"\n\n");
  }
}
```

MinorOfClassification.java

```
package ik;

/**
 * If a graph is a proper minor (not isomorphic) to a graph that is minor
 * minimal with respect to the property of intrinsic knotting, then by
 * the definition of minor minimal, any graph that is a minor of that graph
 * is known to NOT exhibit the same property and thus is not intrinsically
 * knotted.  This classification test uses this logic to determine if a graph
 * is not IK.
 */
public class MinorOfClassification implements IKClassification {
  private Graph minorMinimalIKGraph;

  public MinorOfClassification(Graph minorMinimalIKGraph) {
    this.minorMinimalIKGraph = minorMinimalIKGraph;
  }

  public String classify(Graph subgraph) {
    String result = CANNOT_DETERMINE_IK;

    if (!minorMinimalIKGraph.isIsomorphicTo(subgraph) &&
         minorMinimalIKGraph.containsMinor(subgraph)) {
      result = IS_NOT_IK;
    }

    return result;
  }

  public String getName() {
    return "MinorOf"+minorMinimalIKGraph.getName()+"Classification";
  }

  public String getDescription() {
    return "Any graph that is a minor (excluding isomorphisms) of the "    +
           "known minor minimal IK graph " + minorMinimalIKGraph.getName() +
           " is NOT intrinsically knotted.";
  }
```

```
}
```

NullClassification.java

```
package ik;

/**
 * A simple test to make sure the graph is truly a Graph object and not nil.
 */
public class NullClassification implements IKClassification {
  public String classify(Graph graph) {
    return (graph == null) ? IS_NOT_IK : CANNOT_DETERMINE_IK;
  }

  public String getName() {
    return "NullClassification";
  }

  public String getDescription() {
    return "If the graph is null, then it is NOT intrinsically knotted.  " +
           "This test is just to rule out any invalid graphs.";
  }
}
```

OrderClassification.java

```
package ik;

/**
 * It was proven by Kohara and Suzuki that a graph with six or
 * fewer vertices is not intrinsically knotted.  This classification test
 * encodes that logic.
 */
public class OrderClassification implements IKClassification {
  public String classify(Graph graph) {
    return (graph.getOrder() < 7) ? IS_NOT_IK : CANNOT_DETERMINE_IK;
  }

  public String getName() {
    return "OrderClassification";
  }

  public String getDescription() {
    return "If there are 6 or less vertices then the graph is " +
           "NOT intrinsically knotted.";
  }
}
```

PlanarityClassification.java

```
package ik;

import static ik.GraphConstants.K33;
import static ik.GraphConstants.K5;

/**
 * It was proven by Blain, Bowlin, Fleming et al. that if a graph is formed
 * from a planar graph plus two vertices, then the graph is not
 * intrinsically knotted.  This classification tests this logic.
 */
public class PlanarityClassification implements IKClassification {
  public String classify(Graph graph) {
    // Remove each possible pair of vertices and see if the remaining graph
    // is planar, by determining if it does not have a K5 or K33 minor
    for (int from = 0; from < graph.getOrder(); from++) {
      for (int to = from+1; to < graph.getOrder(); to++) {
        int[] vertices = {from,to};
        Graph subGraph = graph.removeVertices(vertices);

        if (!((subGraph.getOrder() >= 3) &&
```

```
            (subGraph.getSize() > (3*subGraph.getOrder() - 6))) &&
           !(subGraph.containsMinor(K5))                        &&
           !(subGraph.containsMinor(K33))) {
         return IS_NOT_IK;
       }
     }
   }

   return CANNOT_DETERMINE_IK;
 }

 public String getName() {
   return "PlanarityClassification";
 }

 public String getDescription() {
   return "Any graph that has a planar subgraph after removing any 2 " +
          "vertices is NOT intrinsically knotted.";
 }
}
```

RelativeSizeClassification.java

```
package ik;

/**
 * Campbell, Mattman, Ottman et al. proved that if the number of edges in a
 * graph is greater than or equal to 5 * the number of vertices - 14, then
 * the graph is intrinsically knotted because it contains a K7 minor.
 */
public class RelativeSizeClassification implements IKClassification {
 public String classify(Graph graph) {
   if ((graph.getOrder() >= 7) &&
       (graph.getSize() >= (5 * graph.getOrder() - 14))) {
     return IS_IK;
   } else {
     return CANNOT_DETERMINE_IK;
   }
 }

 public String getName() {
   return "RelativeSizeClassification";
 }

 public String getDescription() {
   return "If there are 5n-14 or more edges (where n is the order) then " +
          "the graph is intrinsically knotted.";
 }
}
```

build.xml

```
<project name="KnotFinder" default="build" basedir=".">
 <description>Build file for KnotFinder</description>

 <!-- set global properties for this build -->
 <property name="src"   location="src"/>
 <property name="build" location="build"/>
 <property name="dist"  location="dist"/>
 <property name="docs"  location="../docs/javadocs"/>

 <target name="init">
   <!-- Create the time stamp -->
   <tstamp/>
 </target>

 <target name        = "build"
         depends     = "init"
         description = "compile the source and generate the distribution">
```

```
    <!-- Compile the java code -->
    <mkdir dir     = "${build}"/>
    <javac srcdir  = "${src}"
           destdir = "${build}"/>

    <!-- Build the distribution jar -->
    <mkdir dir     = "${dist}"/>
    <jar   jarfile = "${dist}/knotfinder-${DSTAMP}.jar"
          basedir = "${build}">
     <manifest>
       <attribute name = "Built-By"   value = "${user.name}"/>
       <attribute name = "Main-Class" value = "ik.KnotFinder"  />
     </manifest>
    </jar>
    <delete file   = "${dist}/knotfinder.jar"/>
    <copy   file   = "${dist}/knotfinder-${DSTAMP}.jar"
            tofile = "${dist}/knotfinder.jar"/>

    <!-- Remove the compiled java class files -->
    <delete dir = "${build}"/>
  </target>

  <target name        = "docs"
          description = "build the javadocs">
    <javadoc destdir = "${docs}">
      <fileset dir = "${src}/ik">
        <include name = "*"/>
      </fileset>
    </javadoc>
  </target>
</project>
```

APPENDIX F

graph.rb

```
# This class is the heart of the project to determine intrinsically
# knotted graphs.  It is objects of this Graph class that represents the
# graphs that we are testing.  The classification tests are performed on
# these objects.  The 'meaty' parts of this class are the minor_of?,
# contains_minor?, subgraph_of?, and isomorphic_to? methods.
#
# An object of type Graph is immutable.  This means that upon initialization
# the edges of the graph are set and cannot be modified.  Any calls to
# remove_vertex, remove_edges, contract_edge will result in a new graph
# being created.  The current graph will not be manipulated.
#
# This class does assume that the nauty tools--dretog, labelg, and planarg
# are in the environment PATH.
class Graph
  attr_reader :name
  attr_reader :order
  attr_reader :edges
  attr_reader :safe_mode

  # Creates a new graph.
  #
  # The name is an arbitrary name for the graph.
  # The order is the number of vertices in the graph.
  # The edges are an array of all of the edges in the graph.
  # The safe mode allows checks to be performed on inputs to the graph.  If
  # set, edges and vertices will be confirmed to be valid before proceeding.
  # The reason that this is optional is because these validations,
  # while valuable, result in roughly a 30% performance hit during the
  # classification efforts.  Since over 99% of the graphs created in this
  # project are created indirectly as a result of a call to 'minor_of' we know
  # that the graphs are valid and do not need any validation, and thus it
  # is safe to remove these tests.  If this class were used 'outside' of this
  # single approach, then the conditions would not necessarily always be met.
  # For this reason the 'safe_mode' option is available and can be set to
  # which will perform the validations.
  #
  # Edges are simply two element arrays (ex. [1,2]).
  #
  # We create an adjaceny matrix in order to make 'easy' operations much
  # more effecient.
  def initialize(name, order, edges, safe_mode=true)
    @name       = name
    @order      = order
    @edges      = edges
    @safe_mode  = safe_mode
    @adj_matrix = (0...@order).to_a.collect{Array.new(@order)}

    if @safe_mode
      Graph.validate_edges!(edges, @order)
      @edges = Graph.canonicalize_edges(@edges)
    end

    @edges.each do |edge|
      @adj_matrix[edge[0]][edge[1]] = @adj_matrix[edge[1]][edge[0]] = true
    end
  end

  # The number of edges in the graph.
```

```
def size
  edges.size
end

# Determines if the supplied edge exists in the graph or not.
def has_edge?(edge)
  Graph.validate_edge!(edge, order) if @safe_mode

  @adj_matrix[edge[0]][edge[1]]
end

# The number of edges attached to the supplied vertex.
def degree(vertex)
  Graph.validate_vertex!(vertex, order) if @safe_mode

  histogram[vertex]
end

# An array of the number of edges attached to every vertex from 0 to order.
def histogram
  @histogram ||= (0...order).to_a.collect{|vert| @adj_matrix[vert].nitems}
end

# Removes the vertex from the graph, returning a new graph without that
# vertex and the edges connected to that vertex.
def remove_vertex(vertex)
  Graph.validate_vertex!(vertex, order) if @safe_mode

  new_order = order - 1
  new_edges = []

  edges.each do |from, to|
    next if from == vertex || to == vertex

    new_from = from > vertex ? from - 1 : from
    new_to   = to   > vertex ? to   - 1 : to

    new_edges << [new_from, new_to]
  end

  Graph.new(name,
            new_order,
            new_edges,
            false)
end

# Removes all of the vertices from the graph, returning a new graph
# without those vertices and edges attached to those vertices.
def remove_vertices(*vertices)
  # Note: This can be optimized to remove multiple vertices in one go,
  #       instead of removing one vertex at a time
  vertices.sort.reverse.inject(self) do |new_graph, vertex|
    new_graph.remove_vertex(vertex)
  end
end

# Removes all of the edges from the graph, returning a new graph
# without those edges.
def remove_edges(*edges_to_remove)
  if @safe_mode
    unless edges_to_remove.all?{|edge| has_edge?(edge)}
      raise "Cannot remove an edge that does not exist."
    end

    edges_to_remove = Graph.canonicalize_edges(edges_to_remove)
  end

  Graph.new(name,
            order,
```

```
            edges - edges_to_remove,
            false)
end

# Contracts the given edge, returning a new graph with that edge contracted.
# Contracting an edge [0,1] means to bring the vertex 1 on to 0, so edges
# that were previously connected to 1 are now connected to 0, and the
# vertex 1 no longer exists.
def contract_edge(edge)
  if @safe_mode
    unless has_edge?(edge)
      raise "Cannot contract an edge which does not exist."
    end

    edge = Graph.canonicalized_edge(edge)
  end

  new_order = order - 1
  new_edges = []

  edges.each do |from, to|
    new_from = if from == edge[1]
                 edge[0]
               elsif from > edge[1]
                 from - 1
               else
                 from
               end

    new_to   = if to == edge[1]
                 edge[0]
               elsif to > edge[1]
                 to - 1
               else
                 to
               end

    # Do not allow loops
    new_edges << [new_from, new_to] if new_from != new_to
  end

  Graph.new(name,
            new_order,
            Graph.canonicalize_edges(new_edges),
            false)
end

# Determines if two graphs, no matter the vertex labelling, are identical.
def isomorphic_to?(graph)
  return false unless order          == graph.order
  return false unless size           == graph.size
  return false unless histogram.sort == graph.histogram.sort

  to_canonical == graph.to_canonical
end

# Determines if the our graph contains as a minor, the supplied graph.
# Optionally can allow two graphs that are isomorphic to be considered
# minors.
def contains_minor?(graph, check_isomorphic=true)
  graph.minor_of?(self, check_isomorphic)
end

# Determines if our graph is a minor of the supplied graph.  Optionally
# can allow two graphs that are isomorphic to be considered minors.
def minor_of?(graph, check_isomorphic=true)
  return false if order > graph.order
  return false if size  > graph.size
  return true  if subgraph_of?(graph, check_isomorphic)
```

```
  # Contracting an edge will result in one less vertex and at least
  # one less edge
  if order < graph.order && size < graph.size
    graph.edges.each do |edge|
      new_graph = graph.contract_edge(edge)

      return true if minor_of?(new_graph)
    end
  end

  false
end

# Determines if our graph is a subgraph of the supplied graph.  Optionally
# can allow two graphs that are isomorphic to be considered subgraphs.
def subgraph_of?(graph, check_isomorphic=true)
  return false if order > graph.order
  return false if size  > graph.size

  if order < graph.order
    # Get the number of vertices equivalent in both graphs, then check
    remove_count = graph.order - order

    each_combination(graph.order, remove_count) do |vertices_to_remove|
      # Note: Could be optimized by determining how many edges would be
      #       removed by removing these vertices, and checking that this
      #       number is at least as many as edges in self.size
      new_graph = graph.remove_vertices(*vertices_to_remove)

      return true if subgraph_of?(new_graph)
    end
  elsif size < graph.size
    # Get the number of edges equivalent in both graphs, then check
    remove_count = graph.size - size

    each_combination(graph.size, remove_count) do |edge_indexes_to_remove|
      edges_to_remove = graph.edges.values_at(*edge_indexes_to_remove)
      new_graph       = graph.remove_edges(*edges_to_remove)

      return true if subgraph_of?(new_graph)
    end
  elsif check_isomorphic
    return isomorphic_to?(graph)
  end

  false
end

# Determines if the given graph is planar.  Planar means that the graph
# could be arranged in a plane with no edges crossing over each other.
def planar?
  # Note: I imagine there is more pruning that could be added in here if
  #       it is necessary.  This hadn't proved to be particularly slow.
  return false if order >= 3 && size > (3 * order - 6)

  results = `echo \"#{to_dreadnaut}\" | dretog -q | planarg -qu 2>&1`

  return false if results =~ /0 graphs planar$/
  return true  if results =~ /1 graphs planar$/

  raise "ERROR: Unable to process planarg results '#{results}'."
end

# The dreadnaut form looks like this for the complete graph on 7 vertices:
# n=7g1 2 3 4 5 6;2 3 4 5 6;3 4 5 6;4 5 6;5 6; 6.
def to_dreadnaut
  return @dreadnaut if @dreadnaut
```

```
  buckets = Array.new(@order)
  edges.each do |edge|
    (buckets[edge[0]] ||= []) << edge[1]
  end

  edge_list = buckets.collect{|bucket| bucket.join(' ') if bucket}.join(';')

  @dreadnaut = "n=#{order}g#{edge_list}."
end

# This is a universal, simplistic labeling for a graph that no matter
# the labelling of vertices, two graphs that are isomorphic will have the
# same canonical label.
def to_canonical
  @canonical ||= `echo \"#{to_dreadnaut}\" | dretog -q | labelg -q`
end

# This is the format that can be interpreted by the graph_parser.
def to_s
  "Graph #{name}, order #{order}.\n" +
  "#{order} #{size}\n"               +
  "#{edges.collect{|edge| edge.join(' ')}.join('  ')}\n"
end

# The 'opposite' of a graph.  Where there is an edge in a graph, its
# complement does not have an edge.  Likewise, where there is no edge in
# a graph, the complement does have an edge.
def complement
  Graph.new("#{name}_complement",
            order,
            Graph.complete_graph(order).edges - edges,
            false)
end

# Creates a graph with all of the possible edges for a graph of the
# given order.
def self.complete_graph(order)
  edges = []

  order.times do |from|
    ((from + 1)...order).each do |to|
      edges << [from, to]
    end
  end

  Graph.new("K#{order}", order, edges, false)
end

# Utility methods

# Make sure that the edges are uniq, sorted, and each edge is in the form
# [0,1] and not [1,0].
def self.canonicalize_edges(edges)
  edges.collect{|edge| Graph.canonicalized_edge(edge)}.uniq.sort
end

# Makes an edge in the form [0,1] and not [1,0]
def self.canonicalized_edge(edge)
  edge[0] <= edge[1] ? edge : edge.reverse
end

# Confirms that for the given order, that all of the edges are valid.
def self.validate_edges!(edges, order)
  raise "Edges cannot be nil." unless edges

  edges.each{|edge| Graph.validate_edge!(edge, order)}
end

# Confirms that for the given order the edge is valid.
```

```
  def self.validate_edge!(edge, order)
    Graph.validate_vertex!(edge[0], order)
    Graph.validate_vertex!(edge[1], order)

    if edge[0] == edge[1]
      raise "Loops are not allowed: #{edge[0]} != #{edge[1]}"
    end
  end

  # Confirms that for the given order the vertex is valid.
  def self.validate_vertex!(vertex, order)
    unless vertex >= 0 && vertex < order
      raise "Vertex is not in the valid range: 0 <= #{vertex} < #{order}"
    end
  end

  private
    # Utility method to execute a block for each possible combination of
    # selection_size elements chosen from total_size elements.  The block
    # will be passed the elements (0 based) in the current combination.
    def each_combination(total_size, selection_size, &block)
      set        = Array.new(selection_size)
      next_index = 0
      last_value = -1

      while next_index >= 0 && next_index < selection_size
        if last_value < (total_size - 1)
          set[next_index]  = last_value + 1
          last_value      += 1
          next_index      += 1
        else
          last_value  = set[next_index - 1]
          next_index -= 1
        end

        if next_index == selection_size
          yield set
          last_value  = set[next_index - 1]
          next_index -= 1
        end
      end
    end
end
```

graph_constants.rb

```
# These are the graph constants that are used in various classification
# tests.
#
# Note: There are more graphs which could be added here
#         Foisy identified H, G15, H15, J14, J'14
#         There are also more Triangle-Y exchanges on > 9 vertices
require 'graph'

# 21 Edge Graphs
Graph::K7    = Graph.new('K7',
                         7,
                         [[0,1],[0,2],[0,3],[0,4],[0,5],[0,6],[1,2],[1,3],
                          [1,4],[1,5],[1,6],[2,3],[2,4],[2,5],[2,6],[3,4],
                          [3,5],[3,6],[4,5],[4,6],[5,6]])

Graph::H8    = Graph.new('H8',
                         8,
                         [[0,1],[0,2],[0,4],[0,5],[0,6],[0,7],[1,2],[1,3],
                          [1,5],[1,7],[2,4],[2,5],[2,6],[2,7],[3,4],[3,6],
                          [4,5],[4,7],[5,6],[5,7],[6,7]])

Graph::H9    = Graph.new('H9',
                         9,
```

```
                        [[0,1],[0,2],[0,4],[0,5],[0,7],[0,8],[1,2],[1,3],
                         [1,5],[1,8],[2,4],[2,6],[2,7],[3,4],[3,7],[4,5],
                         [4,8],[5,6],[5,7],[6,8],[7,8]])

Graph::F9    = Graph.new('F9',
                        9,
                        [[0,1],[0,2],[0,4],[0,5],[0,6],[0,8],[1,2],[1,3],
                         [1,5],[1,8],[2,4],[2,5],[2,6],[2,8],[3,4],[3,6],
                         [4,5],[4,8],[5,7],[6,7],[7,8]])

# 22 Edge Graphs
Graph::K3311 = Graph.new('K3311',
                        8,
                        [[0,1],[0,3],[0,4],[0,5],[0,7],[1,2],[1,3],[1,4],
                         [1,6],[2,3],[2,4],[2,5],[2,7],[3,4],[3,5],[3,6],
                         [3,7],[4,5],[4,6],[4,7],[5,6],[6,7]])

Graph::A9    = Graph.new('A9',
                        9,
                        [[0,1],[0,3],[0,5],[0,8],[1,2],[1,4],[1,5],[1,7],
                         [2,4],[2,5],[2,6],[2,8],[3,4],[3,6],[4,5],[4,7],
                         [4,8],[5,6],[5,7],[5,8],[6,7],[7,8]])

Graph::B9    = Graph.new('B9',
                        9,
                        [[0,1],[0,3],[0,4],[0,6],[0,8],[1,2],[1,3],[1,4],
                         [1,5],[2,3],[2,4],[2,6],[2,8],[3,6],[3,7],[3,8],
                         [4,6],[4,7],[4,8],[5,6],[5,7],[5,8]])
```

graph_parser.rb

```
# This class is responsible for parsing all of the graphs from files
# that are in the form:
#   Graph 6, order 4.
#   4 6
#   0 1  0 2  0 3  1 2  1 3  2 3
# It can also parse supplied edges into a graph.  Furthermore, it allows
# for arbitrary parsing of a file so that all graphs are not necessarily
# returned, only graphs indicated.  By default, all graphs in a file will
# be parsed, but it is possible to parse only some of the graphs in the
# form '2:6 8 10:*'
require 'enumerator'
require 'graph'

class GraphParser
  # Creates a new GraphParser that is responsible for parsing each
  # graph.  When limiting the graphs to parse by 'id' it is assumed
  # that the names of the graphs are integer values.  In a normal use
  # of this class, either edges or a graph_filepath will be supplied for
  # 'what graphs' to use.  The graph_ids simply limits to certain
  # graphs within the file in the form (ex. ['2:6', '8', '10:*']).  If no
  # graph_ids are supplied, '*' will be the default.
  def initialize(edges, graph_filepath, graph_ids)
    if !edges.nil? && !edges.empty?
      order          = edges.flatten.max + 1
      @static_graphs = [Graph.new('0', order, edges)]
    else
      @graph_file = File.open(graph_filepath)
      @graph_ids  = graph_ids
    end
  end

  # Return the next graph that should be included in the set of requested
  # graphs.
  def next
    return @static_graphs.shift if @static_graphs

    begin
      new_graph = next_graph
```

```
    end until new_graph.nil? || include_graph?(new_graph.name)

    new_graph
  end

  private
    # Determines if the current graph should be returned or not.  Does
    # it fall in the graph_ids to include.
    def include_graph?(graph_id)
      return true if @graph_ids.nil?
      return true if @graph_ids.empty?
      return true if @graph_ids.include?('*')
      return true if @graph_ids.include?('*:*')
      return true if @graph_ids.include?(graph_id)
      return true if @graph_ids.include?(graph_id.to_i)
      return true if @graph_ids.include?(graph_id.to_s)

      @graph_ids.select{|gid| gid =~ /:/}.each do |id_range|
        min_id, max_id = id_range.split(':')

        return true if min_id == '*' && (graph_id.to_i <= max_id.to_i)
        return true if max_id == '*' && (graph_id.to_i >= min_id.to_i)
        return true if (graph_id.to_i >= min_id.to_i) &&
                       (graph_id.to_i <= max_id.to_i)
      end

      false
    end

    # Returns the next graph from the file.
    def next_graph
      return nil unless lines = next_graph_lines

      id          = /^Graph (\d+), order (\d+).$/.match(lines[0])[1]
      order, size = /^(\d+) (\d+)$/.match(lines[1])[1,2].collect{|n| n.to_i}
      edges       = lines[2].split.collect{|v| v.to_i}.enum_slice(2).to_a

      raise "Error parsing graph #{id}." unless edges.size == size

      Graph.validate_edges!(edges, order)

      Graph.new(id,
                order,
                Graph.canonicalize_edges(edges),
                false)
    end

    # Gets the 3 lines from the file that represent the graph.
    def next_graph_lines
      return nil if @graph_file.closed?

      graph_lines = []
      4.times{ graph_lines << @graph_file.readline.strip }

      @graph_file.close if @graph_file.eof?

      # First line is blank
      graph_lines[1..3]
    end
end
```

ik_classification.rb

```
# Includes all of the IK Classification related classes.
require 'graph'
require 'graph_constants'
require 'ik_classifications/base'
require 'ik_classifications/null_classification'
require 'ik_classifications/order_classification'
```

```
require 'ik_classifications/absolute_size_classification'
require 'ik_classifications/relative_size_classification'
require 'ik_classifications/planarity_classification'
require 'ik_classifications/minor_of_classification'
require 'ik_classifications/contains_minor_classification'
require 'ik_classifications/classifier'
```

ik_classifications/absolute_size_classification.rb

```
# A test that can rule out some graphs from being IK.  This is based
# on Robertson, Seymour, and Thomas' work.
class AbsoluteSizeClassification < IKClassification::Base
  def classify(graph)
    graph.size < 15 ? IS_NOT_IK : CANNOT_DETERMINE_IK
  end

  def description
    "If there are less than 15 edges then the graph is " +
    "NOT intrinsically knotted."
  end
end
```

ik_classifications/base.rb

```
# This is the base class which represents a classification test
# for determining if a graph is intrinsically knotted or not.
module IKClassification
  class Base
    IS_IK               = 'ik'
    IS_NOT_IK           = 'not_ik'
    CANNOT_DETERMINE_IK = 'indeterminate'

    # This method will classify the given graph into one of three states--
    # ik, not ik, or indeterminate
    def classify(graph)
      CANNOT_DETERMINE_IK
    end

    # The name of this classification test.
    def name
      self.class.name
    end

    # A description for this classification test.
    def description
      "Classifies the graph."
    end
  end
end
```

ik_classifications/classifier.rb

```
# This class will actually perform the classification efforts. It will
# use all of the classification tests and apply them one at a time to a
# given graph until a non-indeterminate state can be determined for the graph.

require 'graph'

module IKClassification
  class Classifier
    CLASSIFICATIONS = [NullClassification.new,
                       OrderClassification.new,
                       AbsoluteSizeClassification.new,
                       RelativeSizeClassification.new,
                       PlanarityClassification.new,
                       ContainsMinorClassification.new(Graph::K7),
                       ContainsMinorClassification.new(Graph::H8),
                       ContainsMinorClassification.new(Graph::H9),
                       ContainsMinorClassification.new(Graph::F9),
                       ContainsMinorClassification.new(Graph::K3311),
```

```
                        ContainsMinorClassification.new(Graph::A9),
                        ContainsMinorClassification.new(Graph::B9),
                        MinorOfClassification.new(Graph::K7),
                        MinorOfClassification.new(Graph::H8),
                        MinorOfClassification.new(Graph::H9),
                        MinorOfClassification.new(Graph::F9),
                        MinorOfClassification.new(Graph::K3311),
                        MinorOfClassification.new(Graph::A9),
                        MinorOfClassification.new(Graph::B9)]

    # Classify the given graph with respect to the property of
    # intrinsic knotting.  Returns an array of the results in the form:
    # [classification result, classification test name, execution time]
    def self.classify(graph)
      start_time                = Time.now
      result                    = nil
      determining_classification = nil

      CLASSIFICATIONS.each do |classification|
        result = classification.classify(graph)

        if result != Base::CANNOT_DETERMINE_IK
          determining_classification = classification
          break
        end
      end

      end_time = Time.now

      [result, determining_classification, end_time - start_time]
    end
  end
end
```

ik_classifications/contains_minor_classification.rb

```
# If a graph contains as a minor or is isomorphic to a known IK graph, then by
# the definition of a minor, the graph exhibits the intrinsic knotting
# property.  This classification uses this logic to determine if a graph
# is intrinsically knotted.
class ContainsMinorClassification < IKClassification::Base
  def initialize(ik_graph)
    @ik_graph = ik_graph
  end

  def classify(graph)
    graph.contains_minor?(@ik_graph) ? IS_IK : CANNOT_DETERMINE_IK
  end

  def name
    "ContainsMinor#{@ik_graph.name}Classification"
  end

  def description
    "Any graph that contains the known IK graph #{@ik_graph.name} " +
    "as a minor (including isomorphisms) is intrinsically knotted."
  end
end
```

ik_classifications/minor_of_classification.rb

```
# If a graph is a proper minor (not isomorphic) to a graph that is minor
# minimal with respect to the property of intrinsic knotting, then by
# the definition of minor minimal, any graph that is a minor of that graph
# is known to NOT exhibit the same property and thus is not intrinsically
# knotted.  This classification test uses this logic to determine if a graph
# is not IK.
class MinorOfClassification < IKClassification::Base
  def initialize(minor_minimal_ik_graph)
```

```
    @minor_minimal_ik_graph = minor_minimal_ik_graph
  end

  def classify(graph)
    if graph.minor_of?(@minor_minimal_ik_graph, false)
      IS_NOT_IK
    else
      CANNOT_DETERMINE_IK
    end
  end

  def name
    "MinorOf#{@minor_minimal_ik_graph.name}Classification"
  end

  def description
    "Any graph that is a minor (excluding isomorphisms) of the known " +
    "minor minimal IK graph #{@minor_minimal_ik_graph.name} is NOT "   +
    "intrinsically knotted."
  end
end
```

ik_classifications/null_classification.rb

```
# A simple test to make sure the graph is truly a Graph object and not nil.
class NullClassification < IKClassification::Base
  def classify(graph)
    graph ? CANNOT_DETERMINE_IK : IS_NOT_IK
  end

  def description
    "If the graph is nil, then it is NOT intrinsically knotted.  This " +
    "test is just to rule out any invalid graphs."
  end
end
```

ik_classifications/order_classification.rb

```
# It was proven by Kohara and Suzuki that a graph with six or
# fewer vertices is not intrinsically knotted.  This classification test
# encodes that logic.
class OrderClassification < IKClassification::Base
  def classify(graph)
    graph.order <= 6 ? IS_NOT_IK : CANNOT_DETERMINE_IK
  end

  def description
    "If there are 6 or less vertices then the graph is " +
    "NOT intrinsically knotted."
  end
end
```

ik_classifications/planarity_classification.rb

```
# It was proven by Blain, Bowlin, Fleming et al. that if a graph is formed
# from a planar graph plus two vertices, then the graph is not
# intrinsically knotted.  This classification tests this logic.
class PlanarityClassification < IKClassification::Base
  def classify(graph)
    each_possible_graph(graph) do |new_graph|
      return IS_NOT_IK if new_graph.planar?
    end

    CANNOT_DETERMINE_IK
  end

  def description
    "Any graph that has a planar subgraph after removing any 2 vertices is "+
    "NOT intrinsically knotted."
  end
```

```
  private
    # Remove each possible pair of vertices
    def each_possible_graph(graph, &block)
      (0...graph.order).each do |first|
        (first+1...graph.order).each do |second|
          yield graph.remove_vertices(first, second)
        end
      end
    end
end
```

ik_classifications/relative_size_classification.rb

```
# Campbell, Mattman, Ottman et al. proved that if the number of edges in a
# graph is greater than or equal to 5 * the number of vertices - 14, then
# the graph is intrinsically knotted because it contains a K7 minor.
class RelativeSizeClassification < IKClassification::Base
  def classify(graph)
    if graph.order >= 7 && graph.size >= ((5 * graph.order) - 14)
      IS_IK
    else
      CANNOT_DETERMINE_IK
    end
  end

  def description
    "If there are 5n-14 or more edges (where n is the order) then the " +
    "graph is intrinsically knotted."
  end
end
```

standard_options_parser.rb

```
#!/usr/bin/env ruby

# This will be used by some of the tools as a common interface to
# indicate which graphs to work with.
#
# The following forms are supported:
# ./tool_name -f <path_to_graphs>
# ./tool_name -f <path_to_graphs> [graph ids]
# ./tool_name -e <edges for a graph>
#
# Specific Examples:
# ./tool_name -f ../graphs/connected_graphs_7.txt
# ./tool_name -f ../graphs/connected_graphs_7.txt 3 4 5:8 12:*
# ./tool_name -f ../graphs/connected_graphs_7.txt *:15 17 19
# ./tool_name -e 0 1  0 2  0 3  1 2  1 3  2 4
#
# This file makes available the following constants and variables to the
# tools that use it:
#   OPTIONS        - The command line options that were used in a hash
#   COMMAND_ISSUED - The command that was invoked
#   @graph_parser  - The GraphParser object that is ready to be used
#   @out           - The output stream to be used
require 'optparse'
require 'enumerator'
require 'graph_parser'

COMMAND_ISSUED = "#{$0} #{ARGV.join(' ')}"
OPTIONS        = {}

OptionParser.new do |opts|
  opts.banner  = "Usage: #{$0} -f <graph_file> [ids list (ex. 2 3:10 23:*)]\n"
  opts.banner += "Usage: #{$0} -e <list of edges (ex. 1 2  0 3  4 5)>\n"

  opts.on("-e", "--edges", "Space separated list of edges.") do
    OPTIONS[:edges] = ARGV.collect{|v| v.to_i}.enum_slice(2).to_a
```

```
  end

  opts.on("-f", "--infile  STRING", "The file with the graphs.") do |filepath|
    OPTIONS[:infile]      = filepath
    OPTIONS[:graph_names] = ARGV
  end

  opts.on("-o", "--outfile STRING", "The output file path.") do |filepath|
    OPTIONS[:outfile] = filepath
  end
end.parse!

unless OPTIONS[:infile] || OPTIONS[:edges]
  $stderr.puts "A graph source file (-f) or list of edges (-e) is required."
  exit 0
end

begin
  @graph_parser = GraphParser.new(OPTIONS[:edges],
                                  OPTIONS[:infile],
                                  OPTIONS[:graph_names])

  @out = OPTIONS[:outfile] ? File.new(OPTIONS[:outfile], 'w') : $stdout
rescue => e
  $stderr.puts e.message
  exit 0
end
```