DERIVING RAMSEY NUMBERS

THE NEW MEXICO

SUPERCOMPUTING CHALLENGE

FINAL REPORT

APRIL 4, 2007

TEAM 7

ALBUQUERQUE ACADEMY

Project Members:

Punit Shah

Jack Ingalls

Teacher:

Jim Mims

Project Mentor:

David Metzler, Ph.D.

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This project intends to calculate Ramsey numbers as efficiently as possible. Ramsey numbers, a topic in mathematics and graph theory, give the smallest graph size so that when its edges are colored in any pattern of only two colors, the graph must contain subgraphs of size *r* or *b*, which are both integers. Applications of Ramsey numbers include in the areas of telecommunication, network security, and networking related fields; Ramsey numbers could be used to find the minimum number of necessary connections between computers in a network to create the most secure yet productive network.

We developed a program in Java to calculate Ramsey numbers. The program first takes three integer inputs: the sizes of the two subgraphs that the program will be searching for and a guess of what the Ramsey number will be for the inputted subgraph sizes. The program creates a virtual representation of the graph and begins to iterate through all possible edge colorings, finding whether there are monochromatic subgraphs in each coloring or not. If the program finds a coloring with no monochromatic subgraphs, then it knows that the Ramsey number is greater than the guess and outputs a key for the second, graphics program that will draw the graph and its coloring to prove visually that the program is correct. However, if the graph has no monochromatic colorings, the program knows that the Ramsey number if less than or equal to the guess. By testing various guesses, the user can calculate Ramsey number or narrow the known range of higher input value Ramsey numbers.

Some of the major innovations of the project were the algorithms developed to reduce both the number of colorings to analyze and the processing time. While a "brute-force" approach to calculating Ramsey numbers would test every coloring and every subgraph, our strategic algorithms only tested a subset of the total colorings. These improvements decreased computing time.

The results of the program and algorithms were very promising. For low guesses like seven, the program could theoretically reduce computing time by 80%. This percentage increases for even larger guesses. In practice, even after implementing just one of our performance enhancement algorithms, there was a 33% processing time reduction. Combined with the other algorithms and programming tactics like exiting loops once the program has enough information to form a generalization, the computing time savings would likely approach their theoretical values for larger graph sizes.

This research demonstrates that a computer-based approach to calculating Ramsey numbers is possible and that the algorithms that we used to calculate and reduce the computing time of the program are all accurate and useful in this context. While our time savings alone might not be enough to calculate Ramsey numbers with large inputs, these algorithms could be used in a much larger and broader professional research project intending to use supercomputers to calculate Ramsey numbers.

The major achievements of the project were accurately and efficiently calculating Ramsey numbers within an expandable code structure.

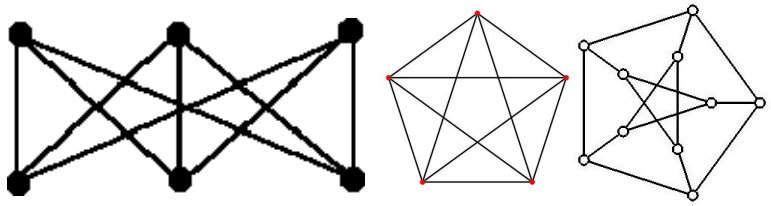
We hope to run our program on a Los Alamos National Labs supercomputer by April 23, 2007.

INTRODUCTION

In order to fully understand the basis of our project, here is some basic background information and definitions. These definitions are necessary in understanding graph theory and Ramsey Numbers.

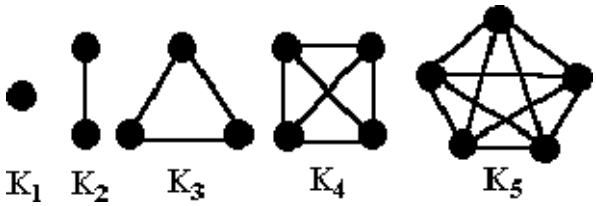
DEFINITIONS

- **Graph**: A finite set of dots (called nodes or vertices) connected by links (called edges or arcs). Throughout our project, we use somewhat non-standard notation and may also refer to a graph as a **map**. There is not difference between these two terms.



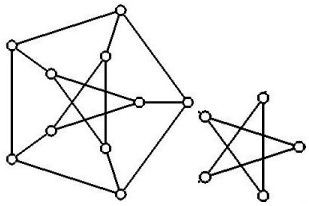
Some examples of graphs
http://www.utm.edu/departments/math/graph/glossary.html
http://www.stephanspencer.com
http://www.math.fau.edu

- **Node or Vertex**: The "dots" in a graph that are connected by edges
- **Edge or Arc**: The connections between different nodes
- **Complete graph**: A complete graph is one such that each of its *n* vertices is connected to each of the other points



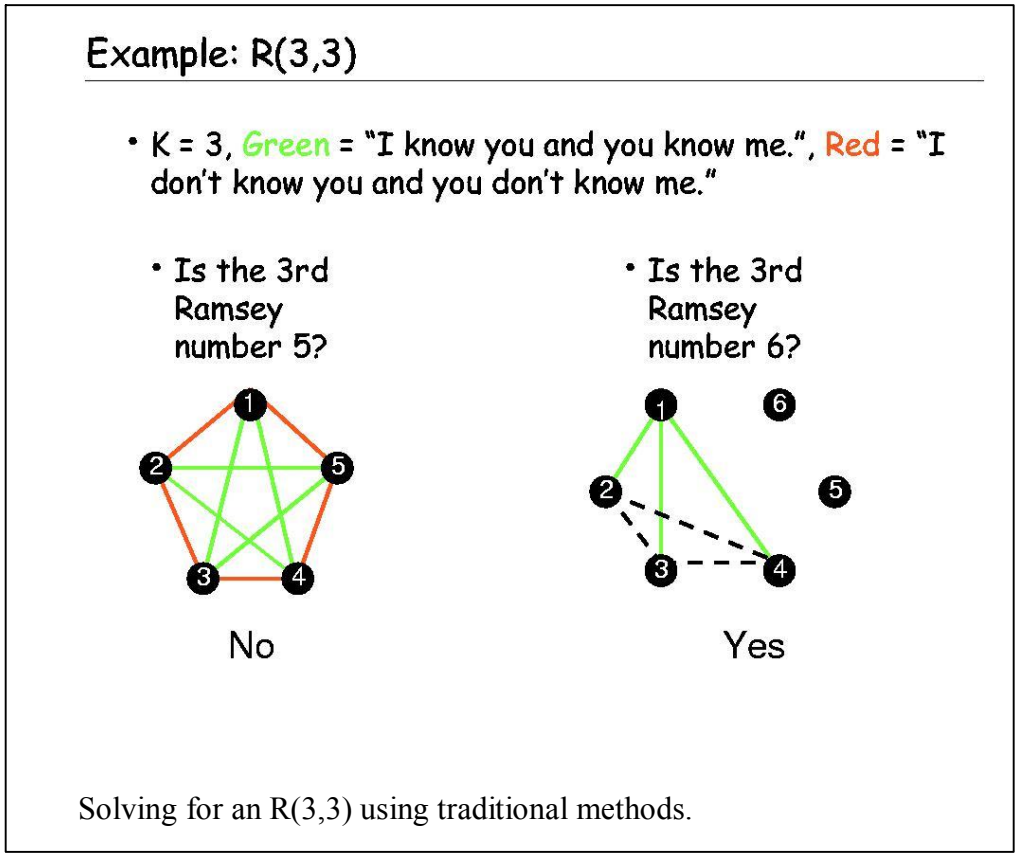
The first 5 complete graphs, http://www.utm.edu/departments/math/graph/glossary.html

- **Subgraph**: A graph that contains a certain set of nodes that are contained in the original graph with all of the edges between the nodes of the subgraph.



The second graph is a subgraph of the first graph.
http://www.math.fau.edu

- **Ramsey Numbers**: A part of graph theory where there are independent variables *b* and *r*. The Ramsey number for those independent variables is the minimum number *n* such that any coloring of a K_n with each edge colored either red or blue will have a complete, monochromatic subgraph colored all blue with *b* nodes or colored all red with *r* nodes. This is a representation of the classic "party problem" which tries to determine the minimum number of people such that there are *b* people who all know each other or *r* people who all do not know each other.



Solving for an R(3,3) using traditional methods.

NOTATION

- K_n: A complete graph with n nodes
- ***R(r, b) = n***: The function that returns the Ramsey number n. The r and b denote the size of the mono-colored subgraphs that are to be used (r is used to mean red while b is used to mean blue). Note that a $R(r, b) = R(b, r)$ because the only difference is that the colors are reversed, but the same values are being used.

The reason why we choose this project is that it is a relatively new topic in mathematical history (it is only about 75 years old) and is also a relatively unexplored topic; to date, there has not been a very large effort by the mathematical community to calculate these numbers, possibly

because of the extreme complexity for high values of inputs *r* and *b*. As you can see from Appendix C, many Ramsey number variables have not been solved for. Therefore, we decided to choose this problem to see if we could make a small yet significant contribution to the mathematical community that future researchers could use as part of a more intensive project to reduce the complexity of calculating Ramsey numbers for larger values of *r* and *b*. We also realized that a project to calculate Ramsey umbers would easily lends itself directly to a computer-based solution as it is only pure mathematics that is going on. There are no random occurrences and no variables that might make the computer model different from a real life model.

Ramsey numbers have a plethora of theoretical applications. While many of those applications are purely mathematical, many others connect to fields like communications, computer network security, telecommunications, theoretical computer science, and nearly anything that deals with networking.

One of the interesting applications where Ramsey numbers could be used is network security. Suppose a network administrator has a certain number of computers. In this network, every pair of computers either has a connection between them or they do not have a connection. The network administrator wants to achieve a balance of security and networking. To improve security, no set of *b* number of computers may all be connected at once since fewer connections will allow for less of a chance of a virus spreading or for information being sent when it should not. However, productivity must still continue at an adequate pace and therefore no set of *r* number of computers may be completely unconnected. This is related to Ramsey numbers since connections between computers could be represented by red edges and absent connections between computers could be represented by blue edges. Now the problem is turned into a

Ramsey number problem looking at R(b,r). Therefore, if the network administrator knows what the Ramsey number is for an R(b,r), then he can determine if a network with his parameters may be created with the number of computers that are inside of the network. If R(b,r) is greater than the number of computers in the network, then there is a way to network the computers together under those parameters and a program could figure out possible ways to connect the computers. If R(b,r) is less than or equal to the number of computers that are being networked, there will be no way to avoid breaking one of those parameters. Therefore, Ramsey numbers have a theoretical application in network security, among other fields.

DESCRIPTION

Our project intended to calculate Ramsey numbers. Even though this may seem like a simply task, the time required to calculate even an R(3,4) with simplistic methods on a home computer can be in terms of hours to days. Therefore more specifically, our project attempted to find algorithms and make use of higher-level programming ideas in order to most efficiently calculate Ramsey numbers in the least amount of time. In the end, the results about how much time we were able to save as a result of implementing new algorithms should be just as important as being able to calculate the numbers themselves.

Our project mentor, David Metzler, Ph.D., recommended that we also try not only to calculate Ramsey numbers, but also to reduce the ranges of unknown Ramsey numbers. For instance, an R(4, 6) is known to be greater than 34, but less than 42. If we found a 34-node graph with no monochromatic subgraphs of neither sizes four nor six, we would have made a significant and new discovery in the mathematical sphere. (A list of stated values and known range for Ramsey numbers is available in Appendix C.

METHOD AND COMPUTATIONAL MODEL

THE MAIN AND PRIMARY METHODS OF THE PROGRAM

To solve this problem of calculating Ramsey numbers, we created a program that would search for subgraphs of user-determined sizes within a larger graph of a guessed number of nodes, which is essentially a guess at the value of the Ramsey number for the inputted subgraph sizes. The program creates a representation of the guess-sized graph in the computer and cycles through all possible coloring schemes. In all the schemes, the program tries to find a coloring that has no monochromatic colorings of either size of subgraph, what we defined to be a "good" coloring. If it finds such a coloring, then the program will exit knowing that that the Ramsey number is larger than the guess. If the program finds no colorings with no monochromatic subgraphs of either color, then the program reports that the Ramsey number is less than or equal to the guess. (For a visual representation of the steps taken by the program, see Appendix A, Figure 1 for a flowchart. Please refer to this diagram throughout this section of the report.)

In writing the program to solve our problem, we considered various programming languages and ultimately chose Java. Java provided features like portability that were important for our project. While we planned to run the program on home PCs, we have plans to also run the program on a supercomputer, many of which use UNIX. Therefore, being able to move our code from machine to machine with ease is essential. Additionally, features like garbage collection and object orientated capabilities helped with processor and memory management as well as program simplification. Java also can easily output data graphically, which would be useful for such a visual problem like ours. It should be noted that Java is sometimes criticized for its possibly slower performance compared to languages like C++ because it has an additional layer between the code and the processor: the Java Virtual Machine. However the performance

differences between Java and other languages has been significantly reduced in previous years, and therefore, we ultimately found that Java's benefits outweighed any shortcoming for our applications.

The program to calculate Ramsey numbers starts by requesting three inputs, all positive integers. (Input windows: Appendix B, Figure 1) The inputs are:

- A guess at the value of the Ramsey number the user would like to calculate. This would be the number of nodes in the larger graph, the graph within which the program would search for monochromatic subgraphs. This value is saved to a variable named *ptsInMap*. (Here in, we will refer to this graph that we search inside as the "larger graph" because it is larger than the subgraphs discussed next.)
- The values r and b in the expression R(r,b). These are the sizes of the subgraphs that the program will be searching for within the larger graph. The program saves these values to two variables named *submapSize1* and *submapSize2*. *SubmapSize1* is always made to be the larger of the two if the two are not equal.

Note that the program has error catching and logic checks to ensure that all inputs are possible.

After these values are inputted and saved, the program goes on to create the array that will represent the larger graph. While initially we considered using advanced dynamic data structures like Java's hash map, we determined while keeping with the motto "keep it simple" that a simple array would have used less overhead processing time, thus making the program more efficient and thereby fulfilling a major goal of the project.

The array that represented the larger graph is called *theMap*. In principle, it represents and contains all the information about the edges between all of the nodes in the graph. It is an

unbalanced, two-dimensional, Boolean array (see Appendix A, Figure 2 for a concise and visual explanation of the array). In Java, multi-dimensional arrays are stored as arrays of arrays. Our first array that contained the sub-arrays was as long as the number of points in the graph (*ptsInMap*). The sub-arrays were as long as their index in the first array. Note that in doing this, the program is essentially assigning each node an index value going from 0 to *ptsInMap*. Each edge of the graph is then represented by a cell in *theMap* whose indices are the two nodes at the ends of the edge. The edge's color is determined by the Boolean value of its cell in the array: the two "colors" are true and false.

Note that this array is unbalanced so that only one cell represents each edge (there is only one cell that shares the same two indices). This helps in multiple ways. First, no extra and possibly conflicting data is stored in the array representation of the larger graph. Secondly, the program does not have to waste valuable time saving superfluous and repeated information. This approach forced us as programmers to be much more careful when calling the array by always calling it with the larger index first, but the investment in carefulness paid out in improved performance, a worthy sacrifice.

The program then moves into setting up the array for testing colorings. We initially set all connections to the last node (with index *ptsInMap* – 1) to "color" false. Afterwards, the program defines variables to keep track its status in terms of solving the problem. These variables tell, for instance, whether there are colorings of the larger graph left to test or whether the program has found a graph that allows it to quit a while loop early.

After all of this initial setup, the program starts the first, outermost while loop that cycles through all possible colorings of *theMap*. The program changes the colorings of *theMap* by changing the values of its cells to true or false, our two "colors". As previously stated, "good"

colorings are defined to be those that have no monochromatic subgraphs of either color in it. By finding even one good coloring scheme, the program can definitively declare that the real Ramsey number of the given values of r and b is greater than the guess. The real Ramsey number or guesses above the real value will by definition only result in graph colorings that have monochromatic subgraphs in it.

The color-switching occurs in a method called *fillTheMap*. The color-rotating algorithm works by generating all permutations for the different possible coloring schemes. This uses an object of the class *ColoringGenerator*, which as the name implies, will generate all colorings. We will discuss this class and the coloring generation process in detail later.

For each coloring, the program then calls the method *testColoringForASize* for each "color" true or false and its respective subgraph size. *TestColoringForASize*, while keeping a coloring constant, goes through the graph and searches for monochromatic subgraphs of a specific subgraph size and "color" (true or false) passed to it. If the method does not find a single monochromatic subgraph within *theMap* with its current coloring, *testColoringForASize* returns false. If it finds a monochromatic subgraph, however, it returns true. *TestColoringForASize* is called twice within the Boolean statement of an if statement. First, it is passed the smaller subgraph size, *submapSize2*, (this subgraph size was set to be the smaller subgraph when the program queried the user for its inputs) with a color of false. The program checks for smaller monochromatic subgraphs first because it will take less time to find smaller monochromatic subgraphs within a coloring than to find larger ones, allowing the program to potentially be able to draw a conclusion earlier. Secondly, the program calls *testColoringForASize* and passes *submapSize1* and true as the color. Note that the if statement uses short-circuit evaluation to save time; if *testColoringForASize* returns false for the first set of parameters, the program does not

even check the second set of parameters because there is no way that the expression in the if's Boolean statement will come out to be true. This is achieved using the "*&&*" operator instead of the "*&*" operator for standard evaluation.

TestColoringForASize is a function within the *Ramsey* class, the same class as the program's *main*. As stated, it tests a coloring for a subgraph of a specified size and color. Inside *testColoringForASize* are two while loops. The first (i.e. outermost) of the two loops cycles through all the possible combinations of nodes in the larger graph to create subgraphs of the specified size. In order to cycle though all possible subgraphs, the function uses the class *CombinationGenerator*, which returns all possible combinations of the nodes in *theMap* in sets sized to the number of nodes in the subgraph for which the program is searching. This class in described in further detail below.

The second (i.e. inner) loop rotates through the different edges in the specific subgraph. It rotates through the different edges by using *CombinationGenerator* as well; this time, it tries to finds all possible pairs of nodes in the submap.

If in the inner loop of the program ever finds an edge that is not of the passed color, then it immediately exits the inner loop and continues to cycle through the other subgraphs, trying to find one that is monochromatic. If the inner loop only finds edges of the color passed by the calling method and goes through all the edges in the subgraph, then the program can determine that the coloring indeed has monochromatic subgraphs. Therefore, *testColoringForASize* immediately returns false, telling the calling method, the *main*, that it needs to continue to cycle through colorings as this coloring is not void of monochromatic subgraphs. If, however, *testColoringForASize* goes through all the subgraphs and does not find a single one that is

monochromatic, then the program has found a coloring without monochromatic versions of the specified subgraph in the specified color. In this case, *testColoringForASize* returns true.

When the *main* calls *testColoringForASize* for both colors/subgraph sizes and is returned true both times, the program will know that there are no monochromatic subgraphs inside the current coloring. As previously stated, when *testColoringForASize* returns true, it means that there are no monochromatic colorings of the passed color/subgraph size. When this function returns true for both sets of parameters, it makes sense that there should be no monochromatic colorings of any type in the graph with its present coloring.

If at any time in the program *testColoringForASize* returns true for both sets of parameters, then the program immediately reports its findings: having found a graph with no monochromatic subgraphs, it knows that the Ramsey number must be, by definition, greater than the guess. (Example of this output: Appendix B, Figures 3 and 7) If however the program goes through all the colorings and finds that they all have monochromatic subgraphs within them, then by definition the program knows that the Ramsey number is less than or equal to the guess. (Example of this output: Appendix B, Figure 6)

After determining its findings, the program finally outputs the information to the screen. If the Ramsey number is known to be greater than the guess, then the program will also output an Edge Sequence Key. This key contains the information about the coloring so that it can be graphed using the separate program that we have developed for the task. The graphics and calculation portions of the project were separated in order to allow the calculations to occur on a supercomputer, most of which do not support GUIs. The graphics part was added to visually verify and prove the findings of the calculation part of the project. The workings of the graphics program are described in greater detail below.

ColoringGenerator Class and the *fillTheMap* Function.

Together, the class *ColoringGenerator* and the method *fillTheMap* cycle *theMap* through its different possible colorings. The function *fillTheMap* uses an object of the *ColoringGenerator* class in order to generate all the possible permutations of colorings.

The class *ColoringGenerator* creates the permutations in a function called *getNext*. This function returns an integer array full of ones and zeros. This array is as long as the number of edges in the graph since we need to figure out the colorings of all of the edges. Therefore, each cell represents a certain edge. The algorithm in *getNext* will cycle through all possible permutations of the array with 1s and 0s in order to find all of the colorings of the graph.

The generation of these permutations is achieved by using an algorithm by Donald E. Knuth from his well regarded texts, *The Art of Computer Programming*. These books contain algorithms with the primary goal of reducing the processing time to solve the problem at hand. A reasonable question to ask would be why we did not generate the algorithms to create permutations on our own. The answer is that we could have; creating a simplistic algorithm is a relatively trivial problem. We could have, for instance, created a program that incremented a counter and every time the method *getNext* was called, the program converted the value of the counter into a binary string. This would not have been very efficient, however. Knuth's algorithms are, to the contrary, very efficient and fulfill the tasks we need in the least amount of time. In fact, his algorithm only involves five assignment steps for every call to *getNext*. As researchers, it is important to make use of our resources and past findings by other more established computer scientists. For these reasons, we felt it was justified to consult and make use of the algorithms so we could spend time doing "new" research. In the end, while this is an

important part of the program, these algorithms do not form the core ideas behind the code, all of which are original.

(We developed all the code for this class; generally as in this case, Knuth only provides algorithms, not code.)

Despite how fast it is for the program to generate colorings, it still takes a significant amount of time to analyze the colorings. Therefore, instead of calling *getNext* directly, *fillTheMap* instead calls the method *nextColoring*, which calls *getNext*, but also analyzes the coloring before returning it to the calling method. One of the large time-saving methods is that the program counts the number of each binary digit in the array returned by *getNext*. If there are not enough 0s or 1s to make a graph with no monochromatic subgraph colorings, then the program will skip the whole coloring, resulting in significant time savings.

Another even larger time saver occurs in the method that calls he function *nextColoring*. This method and the class *ColoringGenerator* act as if *theMap* is actually one node smaller than it really is. The idea is that the automatic permutation generator will only systematically go through all the edges containing all the nodes except one. For all the connections going to the last node, a for loop in *fillTheMap* smartly rotates the colorings of the edges with the last node so that some colorings that are just repeats of other colorings are skipped. One of the ways that it can become repeated is if the graph is only rotated. Because position does not matter, one can remove the other colorings that are just rotated. This theoretically should result in very significant time savings.

COMBINATIONGENERATOR CLASS

The *CombinationGenerator* class is used to cycle through the different subgraphs in a coloring and to cycle through the different edges within a subgraph. In order to perform these operations, the class must generate combinations.

In order to generate these combinations, this class makes use of the algorithms of a renowned computer scientist as well. Like in the previous class, we could have made an algorithm to complete the task, however it would be very unlikely that we could make our algorithm as fast as a professional and renowned computer scientist could. For this class, we made use of Kenneth H. Rosen and his book *Discrete Mathematics and Its Applications*. Using Rosen's algorithm, the program in the method *getNext* can get the next combination in as little as four or five assignment operations, much faster than the algorithm that we would have developed.

In order to use the class *CombinationGenerator*, the program had to first create a temporary "holding" array containing all the nodes for which we were creating a combination, whether it was to generate a subgraph or to select an edge between two nodes in a subgraph. *GetNext* outputs an array that tells the program the locations of the nodes in the temporary holding array. The two arrays are then correlated, the node indices extracted, and then the subgraph or edge used in the subsequent calculations.

Note: the class *CombinationGenerator* was adapted from code made available to programmers free on the internet. The class is quite bare as it essentially only contains Rosen's algorithm and a few supporting methods and functions. More information about the exact attribution of this code is available in the source code. We created all the code that utilizes the class and that integrates it into our program. The problem of generating combinations is also

trivial like generating permutations, but it is definitely unlikely that we would be able to create an algorithm more efficient than one from a renowned computer scientist. As with *ColoringGenerator*, this code is not part of the principle ideas behind our program. For these reasons, we felt it was justified to use the code.

ADDITIONAL COMMENTS ON PERFORMANCE ENHANCEMENTS

As a primary goal, the program was supposed to find as many ways to reduce the time needed to calculate a Ramsey number. We implemented these ideas in trying to reduce the number of coloring tested, for instance, by pre-analyzing and eliminating colorings that we know we do not need to analyze for one reason or another. Throughout the project, we also realized that it makes sense to reduce the time spent in the innermost while loop of the function *testColoringForASize* (the while loop that gets the color of specific edges themselves) as this is the part of the code that will by far be run the most times and thus will be the largest contributor to total processing time. So when trying to reduce processing time, we made use of this observation so we could reduce time spent on the process that inherently take the most time.

The program also makes use of other simple, yet remarkably effective ideas to reduce processing time. For instance, at many point in the loops, if a "special" case has been found, the program does not need to continue to look through the different cases. For instance, once the program has found a coloring without any monochromatic colorings, it immediately stops goring through the various colorings and loops. Therefore, one of the simple yet very important time savers is the break command in Java (or the return command depending on the circumstances) that will allow the program to immediately terminate loops and move on rather than make calculation after an accurate generalization can already be made.

Other simple yet very helpful time savers include short-circuit evaluation for Boolean statements. This concept is related to the idea behind the break: stop doing a calculation whenever you have enough information to generalize. Specifically, short-circuit evaluation allows the program to know that a statement has to be either true or false after it has only evaluated part of the Boolean expression. Instead of continuing to evaluate, the program can instead stop, draw its conclusion, and move on. One specific example of our use of short-circuit evaluation is when the program searched for subgraphs within a coloring. If we found a monochromatic coloring of the first color, then we would not even look at the second color, as it would bear no influence on the conclusion about that coloring: it contained monochromatic subgraphs. We only looked at the second coloring if it was possible that it could influence the outcome of the Boolean statement. For example, if we did not find a monochromatic subgraph of the first color, we would not check for the second color since it does not matter.

THE GRAPHING PROGRAM

The second program in our project is the graphing program, which can be used to visually prove that a graph of a guessed number of nodes can indeed contain no monochromatic subgraphs if the calculation program has found this to be true. This program takes the Edge Sequence Key and visually displays the coloring.

After obtaining the key from the user (Example of prompt: Appendix B, Figure 4), the program decomposes the string into the different variables contained within it: the guess (the number of nodes in the larger graph), the sizes of the subgraphs, and then finally the binary string representing the edge coloring.

The program first divides the circumference of a circle into as many arcs as there were points in *theMap* in the calculation program. The program then gets the polar coordinates of the endpoints of these arcs and converts them to Cartesian coordinates. It plots small circles at each of these points to represent the nodes.

Then, through a series of for loops, the program does exactly what the calculation program does to convert the binary array from the class *ColoringGenerator* into a real coloring, but instead of an integer array, the program is dealing with a String, essentially an array of characters. For every zero, the program draws a red line between the nodes, and for every one, the program draws a blue line. After this process, the program prints essential information to the screen (like what the guess and subgraph sizes were) alongside the graph. (Example of the window after graphed and text drawn: Appendix B, Figures 5 and 8)

RESULTS

The first result to look at is whether the program can accurately calculate Ramsey numbers and test guesses. The program indeed accurately calculates Ramsey numbers for small numbers/guesses *n*. Theoretically, the program is able to calculate any Ramsey number that is inputted, and through our benchmarking, we know that the answer will be accurate. So far, the program has solved Ramsey numbers for small values of *r* and *b*. Once we run the code on a supercomputer in the coming weeks, we will be able to generate more results and further benchmark our code. (For a list of stated values from the mathematical community that we used in benchmarking, see Appendix C). The Ramsey numbers that we calculated have all been accurate, indicating that our program is likely accurate.

The second and arguable more important set of results from the program is the improvements in processing time that we have made through the implementation of new algorithms. Processing time has been the primary obstruction for us to calculate Ramsey numbers for larger values of *r* and *b*, the primary reason we hope to run our code on a supercomputer. Therefore, after creating a program that worked to calculate Ramsey Numbers, we created algorithms that sped up the program and more quickly gave the results.

Our first speed-up algorithm determined whether there were enough red edges in a given coloring so that a blue monochromatic subgraph with its corresponding number of nodes could even exist, and vice versa with blue edges to red subgraphs. By not testing the colorings that would definitely have monochromatic subgraphs, we reduced the processing time from about 13.25 seconds of dedicated processing power, to about 13 seconds on a test for R(4, 3) with a guess of 7. This therefore produced a small, yet noticeable difference in the amount of time needed to calculate these Ramsey numbers.

Our second performance enhancement algorithm selected one node and did not go through every possible coloring of the edges connected to it. Instead, the program went through and eliminated all the permutations of *B* blue edges and *(N-B-1)* red edges and looked at only one of those. We can do this because in the graph, nodes can swap positions with other nodes but still result in the same overall coloring. The positioning of the nodes does not matter as long as the connections are the same. Since we go through all the permutations with the rest of the graph, we will get that "swapped" graph at some point. This short-cut reduced the processing time of the R(4,3) with a trial of 7 from our fast time of 13 seconds of dedicated processing power to an even faster 9 seconds.

An additional set of results of our program are the graphs seen in some of the figures in Appendix B. We created a second program that takes an alphanumeric key outputted from the original program and displays it visually as a graph that does not have the monochromatic subgraphs. This visual output proves that the value must be greater than the guessed number for those inputs of *r* and *b*. This is additional output further helped to benchmark the calculations code and for us to see what the graphs that do not contain monochromatic subgraphs look like if there was some sort of pattern among the colorings.

ANALYSIS

The most important results to analyze are those regarding the reductions in processing time. Many of the results that we received came from equations that showed how much time we theoretically should have saved. There are $2^{(\text{total number of edges})}$ of total colorings the program would have to analyze if we did not implement our coloring-reduction algorithms to save time. In the case that the number of nodes guessed is 7, there are 21 total edges and 2^{21} total colorings or 2,097,152 colorings. Using our first method, the number of total colorings that we need to check is reduced to 2,088,043. The ratio of these different numbers of graphs to test is about 1:1.005 or 200:201. This is a small ratio, but enough to make some difference even though the difference may be quite small.

The second algorithm (alone) reduces the number of colorings to $2^{(\text{total number of edges} - \text{the number of nodes} + 1)}$ x (the number of nodes). The general ratio is therefore (the number of nodes):$2^{(\text{the number of nodes} - 1)}$. One can easily notice that the right side will eventually become significantly larger than the left side as we increase the number of edges. Therefore, the larger the *n* (number of nodes) that we test, the larger the time saving is as a result of the time-saving algorithms. For example, using only the second time reduction algorithm on an R(4, 3) with a guess of 7 nodes, the number of colorings is cut from 2,097,152 to 229,376. The ratio is about 1:9 – a substantial time savings.

However, our results show that the theoretical and actual time savings do not completely coincide. One reason is that our program stops cycling through colorings once it reaches a coloring with no monochromatic subgraphs (this is a result of our previously described use of the "break" operator in Java). Additionally, the two algorithms also overlap in that they would both eliminate some of the same colorings and thus we cannot simply multiply their percent time

savings. In addition, especially for the first algorithm, the time to analyze whether the coloring should be tested or not can cut into the expected time savings. Also, many of the beginning graphs contain a node that has only one color coming from it. These colorings have will almost always have a monochromatic subgraph. This is an area for future improvement.

CONCLUSIONS

In general, our project was successful in achieving its objectives. While we still have many more ideas to even further improve our program and computational model, we can still draw many valuable conclusions form the current program and its various performance enhancements. These conclusions include the following:

- A computer and logic-based approach to calculating Ramsey numbers is possible, reaffirming previous research. Only after gathering research to understand the complex ideas behind calculating Ramsey numbers did we even start to apply computer-based thinking to calculating Ramsey numbers. This clearly paid off as we too were able to conclude that it is possible to calculate Ramsey numbers on the computer like the few, past researchers who have tried.
- Our basic algorithms that cycle through colorings, subgraphs, and edges is accurate as the program that we created with these algorithms produced accurate results. Generating the larger framework in which the calculate Ramsey numbers is a challenge in itself both conceptually and computationally, and having solved it, we have created a solution available for future researchers.
- Furthermore, the algorithms that we developed to reduce the number of colorings tested are both accurate created significant time savings. These algorithms improved upon our original, somewhat slow program and created a program that succeeded in reducing the amount of time for calculations. In some cases, the time is reduced by 80% and even more time could be removed in higher sizes. So ultimately, the ideas and algorithms behind our program have all worked to effectively reduce the processing time over a “brute force”-type method.

- Even though the time savings as a result of our innovative algorithms are significant, the are not nearly enough to overcome the significant increases in required processing power for larger and larger graphs. In running the program, we found that processing power skyrocketed when we even added one node to a graph. Nevertheless, the algorithms we developed could be used as part of a larger project to calculate Ramsey numbers on a supercomputer, and with other innovative algorithms, possibly simplify the problem to a manageable size for larger graph sizes.
- As this program accurately calculates Ramsey numbers and does so with a number of time savers, the program fulfill the original intentions and specifications of the project.

These findings, especially the ones relating to successful algorithms, are based on the data and analysis from numerous runs with different inputs as well as visual proof provided by the second, graphing program. This program showed coloring with no monochromatic subgraphs and allowed the user to self-verify the first, computational program's findings.

ACHIEVEMENTS OF THE PROJECT

The primary achievement of this project is that it successfully calculated Ramsey numbers and provided algorithms that successfully and significantly cut processing time, among other things. According to our mentor David Metzler, the hardest parts in calculating Ramsey numbers is to cycle through colorings and to check colorings for monochromatic subgraphs. This program does both of those and does it with more efficiency than a simple brute-force method. In fact, while the time saving mechanisms in the program already save a significant percent of time when run with small inputs on a home PC, the program's time savings should increase logarithmically as we input larger and larger graphs and subgraphs. Therefore, apart from just overcoming the challenges to calculate Ramsey numbers, this program adds significant performance enhancements.

Research in the field of Ramsey numbers is still limited. While the ideas behind Ramsey numbers are simple, computer science approaches to calculating Ramsey numbers are still rare. So while the research that we are doing is not unique, our program has created algorithms that have the potential to overcome the barriers that other computer science approaches to the problem have hit, namely in processing power. This program has actual potential to contribute to the mathematical community if it were included as part of further and much broader professional research into using supercomputers to calculate Ramsey numbers or a related problem.

However, apart from the basic requirements, another achievement of the project is that the program is fully expandable and structured for continual improvement. This type of program is very dependent on having the most efficient algorithms and if at any point we found, for instance, an even quicker way to generate combinations, we could easily switch out the combination generator classes and within minutes have implemented a whole new time-saving

algorithm. Additionally, our use of objects and many methods/functions helps to modularize the code and give it a dynamic, expandable structure. For a problem like this where performance is key, this type of structure fit the project very well. All of these features also allow the program to be open to solving derivative problems with only small changes in the code. For instance, we could easily try to find three subgraphs of three different colors within the larger graph using nearly all of the code from this project.

RECOMMENDATIONS

This project has many natural extensions and side problems that we could explore. To date, we have only implemented a few primary algorithms reduce the number of colorings that the program test. However, there are many other methods that we could try to implement that would reduce processing time and thus improve our project. One way is simply to run the program on a higher performance computer than the one we are currently using. We ran our program on a 2.2 GHz. Microsoft Windows XP machine with 512 MB of RAM. We are currently working with the Supercomputing Challenge to gain access to a higher performance computer at Los Alamos National Laboratories. This will allow us to test and benchmark our program for larger inputs. We hope to have run our program on a Supercomputer by April 23, 2007, the date of the final presentations.

We can also continue to improve our project by implementing new algorithms to speed up our program. One area is to continue to get rid of some of the unnecessary colorings. This includes improving the primary short-cuts that we use to further reduce the number of nodes. We can also apply these extremely fast methods to other nodes, but this will also add an additional layer of complexity; if we extended our algorithm to additional nodes, there will edges that connect to multiple nodes with controlled colorings and that will have been colored twice. Another algorithm we could implement would take all the colorings with n-1 nodes that do not have the monochromatic subgraphs and then just add one more node to the graph. This essential "recycling" of our work would reduce the number of colorings to work through and the amount of tests that need to be done. Yet another algorithm that could consider implementing is trying different ways of searching for monochromatic subgraphs to see which way is the fastest.

The other main way to speed up the program is to only try a few colorings for any given *n*. If we find a graph that has no monochromatic subgraphs, then we know that the Ramsey number is larger. Otherwise, the test would be inconclusive. There are many ways to go about this. Working with symmetry is one and could reduce the number of possible colorings by almost a square-rooting factor. For large numbers of colorings, this is a major improvement. Another way is to begin with a coloring and change the segments in a "smart" way such that the number of monochromatic graphs is reduced until there are no more. A way would be to implement some sort of Monte Carlo coloring algorithm that would randomly color a graph and by probability could give us a good chance of finding a coloring that would result in a conclusive result. These methods are very fast, but at the same time they may not always return a conclusive result. This is something we simply have to see.

We hope to implement at least some of these recommendations within the coming weeks.

ACKNOWLEDGEMENTS AND CITATIONS

We would like to thank Jim Mims, our faculty sponsor, for all of his time and effort in ensuring our success. His approach gave us an ideal balance of independence versus guidance that helped us get as much as possible from the Challenge.

We would also like to extend our thanks to David Metzler, Ph.D., our project mentor. Dr. Metzler was not only also instrumental in keeping us on task, but also provided us with invaluable information about what had hindered mathematical researchers attempting problems like ours that require high performance computers. He also warned us about what we were getting into, encouraging us to look at the problem and find what other side-problems we should explore as well.

We also appreciate all the help, guidance, and support from the everyone on the Supercomputing Challenge's staff. From their very useful comments during the February project evaluation to their help in possibly securing time on a supercomputer, we truly appreciate all their time and effort.

CITATIONS

REFERENCES ON RAMSEY NUMBERS

Exoo, Geoffrey. "Ramsey Numbers," <isu.indstate.edu/ge/RAMSEY/>

Haanpää, Harri. "Computational Methods for Ramsey Numbers,"

http://citeseer.ist.psu.edu/660990.html

Radziszowski, Stanisław P. "Small Ramsey Numbers,"

www.combinatorics.org/Surveys/ds1/sur.pdf

Rosta, Vera. "Ramsey Theory Applications," www.combinatorics.org/Surveys/ds13.pdf

Weisstein, Eric W. Wolfram *MathWorld*, "Ramsey Number," <mathworld.wolfram.com/RamseyNumber.html>

Wikipedia, "Ramsey's theorem," <en.wikipedia.org/wiki/Ramsey's_theorem>

REFERENCES ON ALGORITHMS AND PROGRAMMING

Dynamic programming. (2007, April 3). In *Wikipedia*. Retrieved April 3, 2007, from Wikimedia Foundation Web site: http://en.wikipedia.org/wiki/Dynamic_programming

Gilleland, M. (n.d.). *Combination Generator*. Retrieved April 2, 2007, from Merriam Park Software Web site: http://www.merriampark.com/comb.htm

Knuth, D. E. (2005). Fascicle 3. In *The Art of Computer Programming: Vol. 4. Generating All Combinations and Partitions*. Upper Saddle River, NJ: Pearson.

Knuth, D. E. (2005). Fascicle 2. In *The Art of Computer Programming: Vol. 4. Generating All Tuples and Permutations*. Upper Saddle River, NJ: Pearson.

Rosen, K. H. (1991). *Discrete Mathematics and Its Applications* (2nd ed.). NY: McGraw-Hill.

APPENDIX A: FIGURES FROM THE REPORT

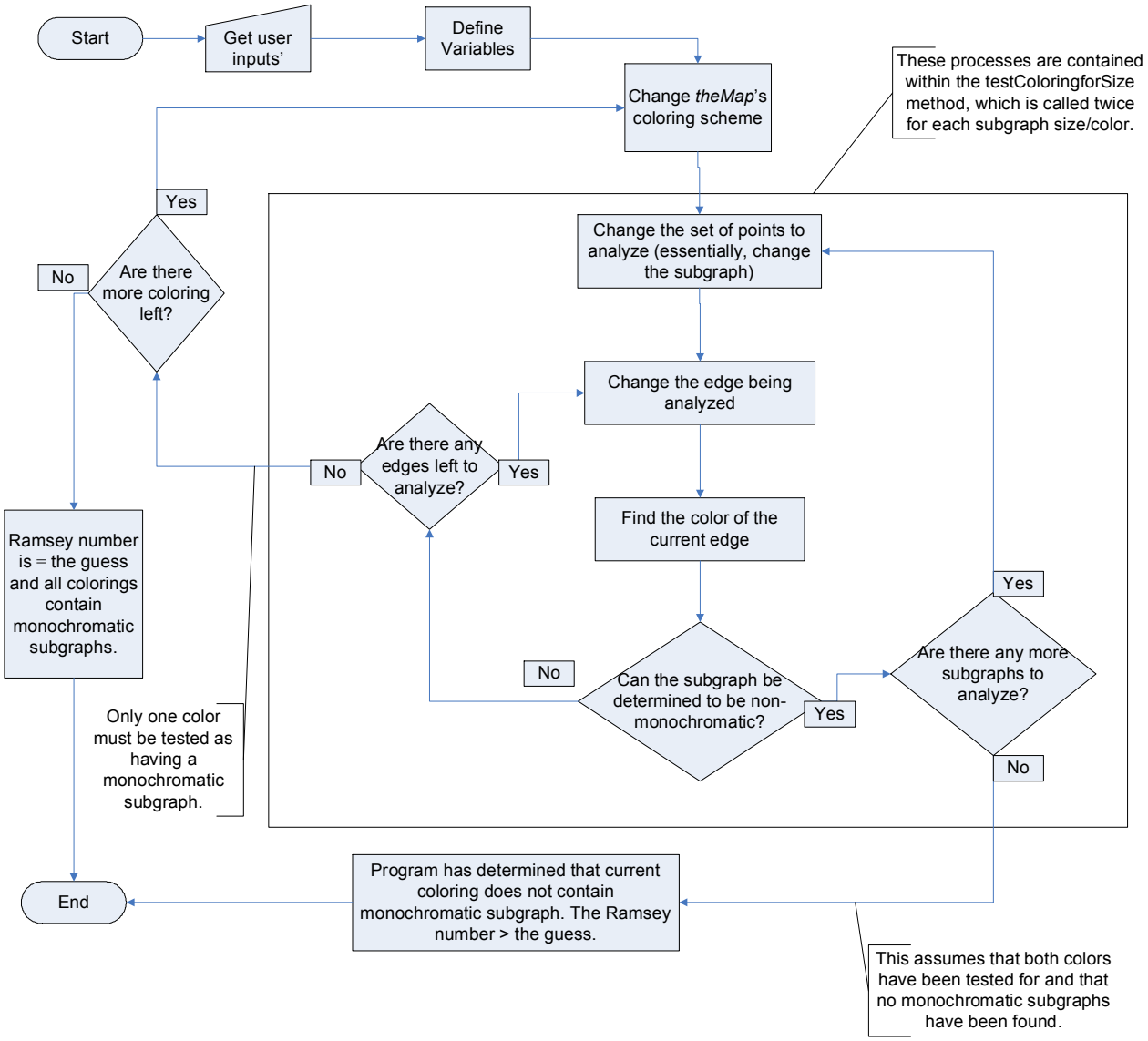


Figure 1: Flowchart of the major structures within the calculation program.

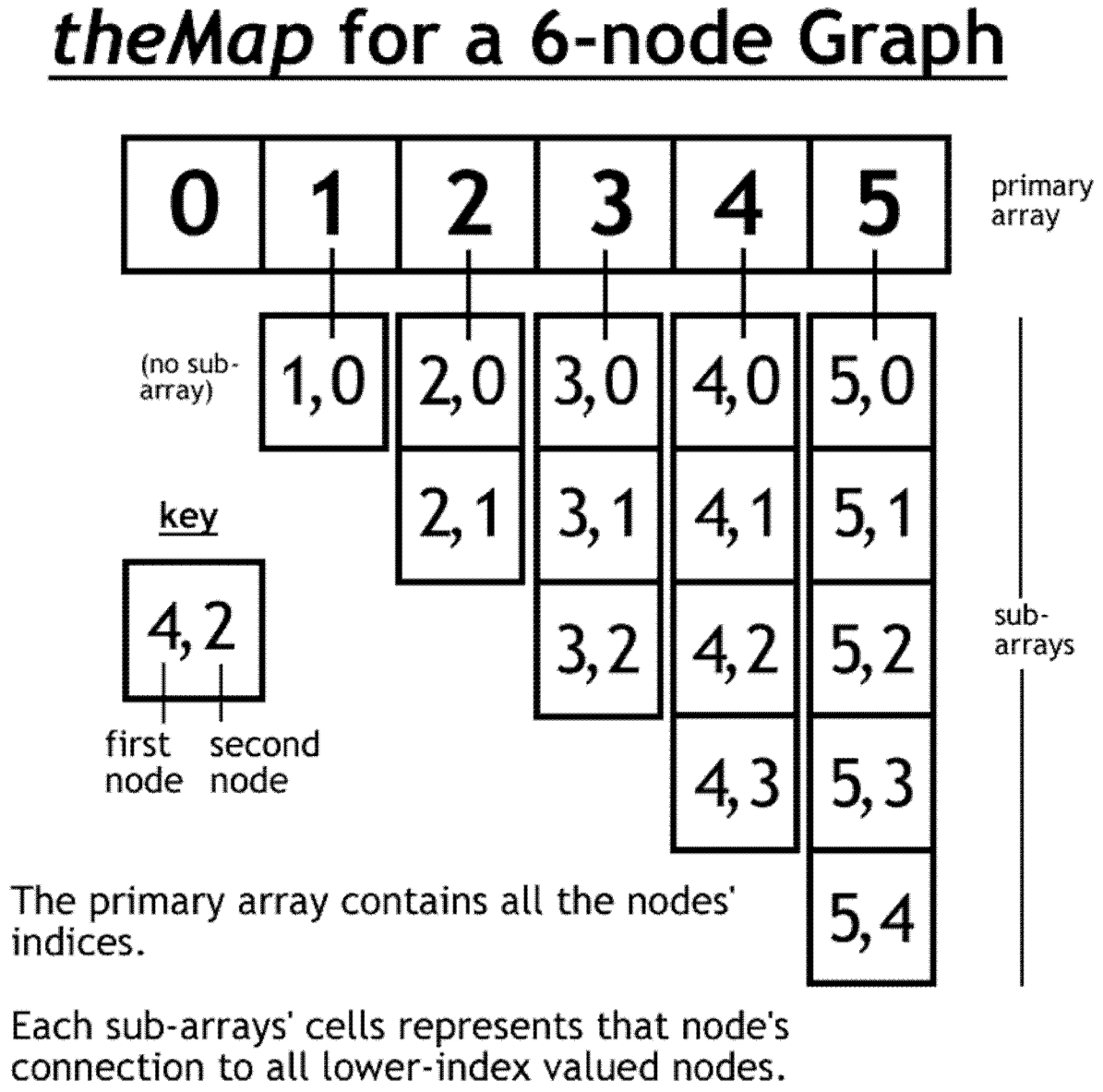
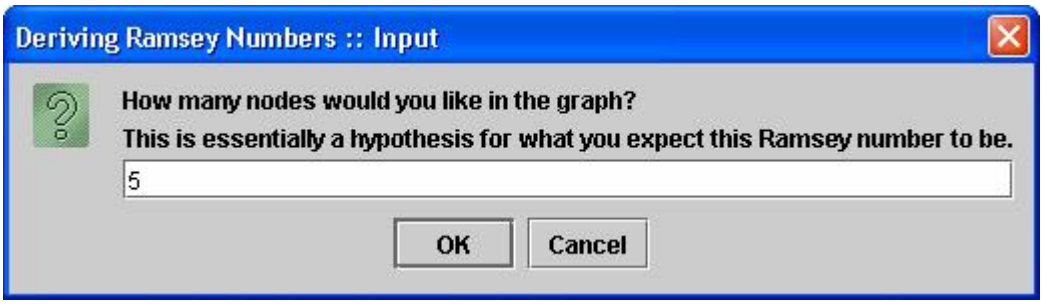
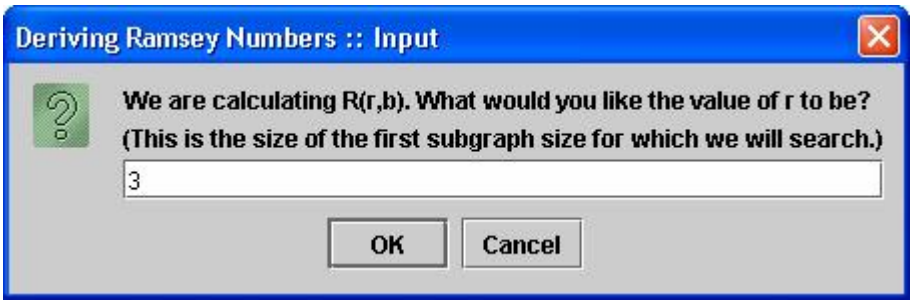


Figure 2: A diagram visually showing how *theMap* works as an unbalanced array. The above diagram is how it looks if the guess was six (so there would be six nodes in the larger map).

APPENDIX B: PROGRAM SCREENSHOTS





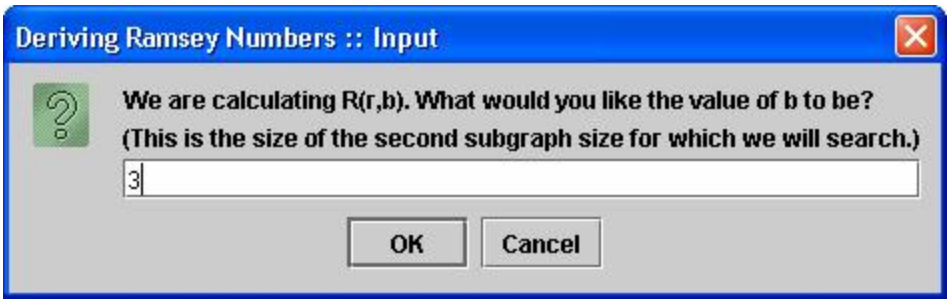


Figure 1: The three input prompts for the calculation program. If this program were to be run on a supercomputer, these prompts could be changed to console window prompts. These would be the inputs for a R(3, 3) with a guess of 5.

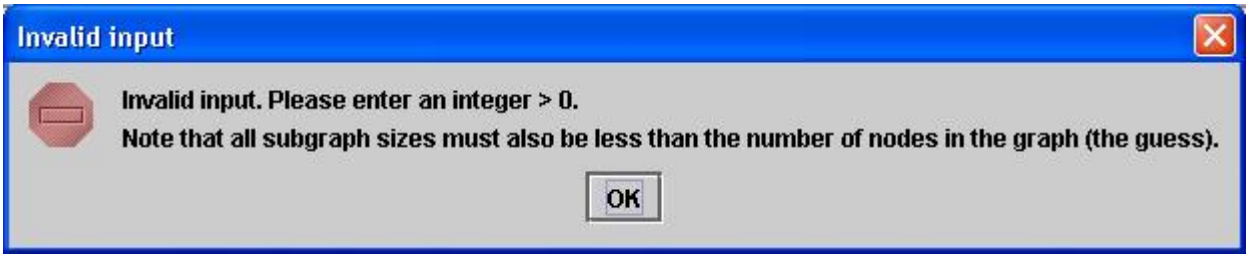


Figure 2: The error message that comes up if the user of the computational program puts in an invalid input. All inputs into that program have error catchers to ensure that only valid inputs are used in the program.

```
Program to Derive Ramsey Numbers
Created for the 2007 New Mexico Supercomputing Challenge.
(c) 2007 Punit Shah and Jack Ingalls. All rights reserved.
Team 7, Albuquerque Academy


BEGINNING OF CONSOLE I/O WINDOW:

We will be searching for subgraphs of size 3 and 3 in a map of size 5
More technically, we are calulating R(3, 3) with a guess of 5.

This graph of 5 nodes has a nonmonochromatic coloring. Ramsey number is definately greater than 5.
More technically: R(3, 3) > 5.

The Edge Sequence Key can be used to display a picture of the graph with no monochromatic subgraphs in it that the
program discovered. Enter this when requested in the appropriate program. The Edge Sequence Key is:
5X3X3X0100111100

END OF CONSOLE I/O WINDOW
```

Figure 3: The console output window for the inputs in Figure 1. As the table in Appendix C states, *R*(3, 3,) is > the guess, 5. The Edge Sequence Key, displayed because there is a graphing with no monochromatic colorings, is 5X3X3X0100111100.

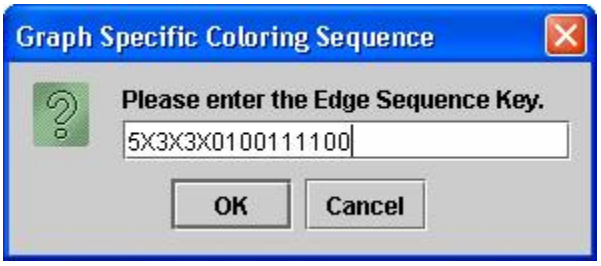


Figure 4: The input window for the second, graphing program. This input asks for the Edge Sequence Key from the calculations program. The one entered is the from the output in Figure 3.

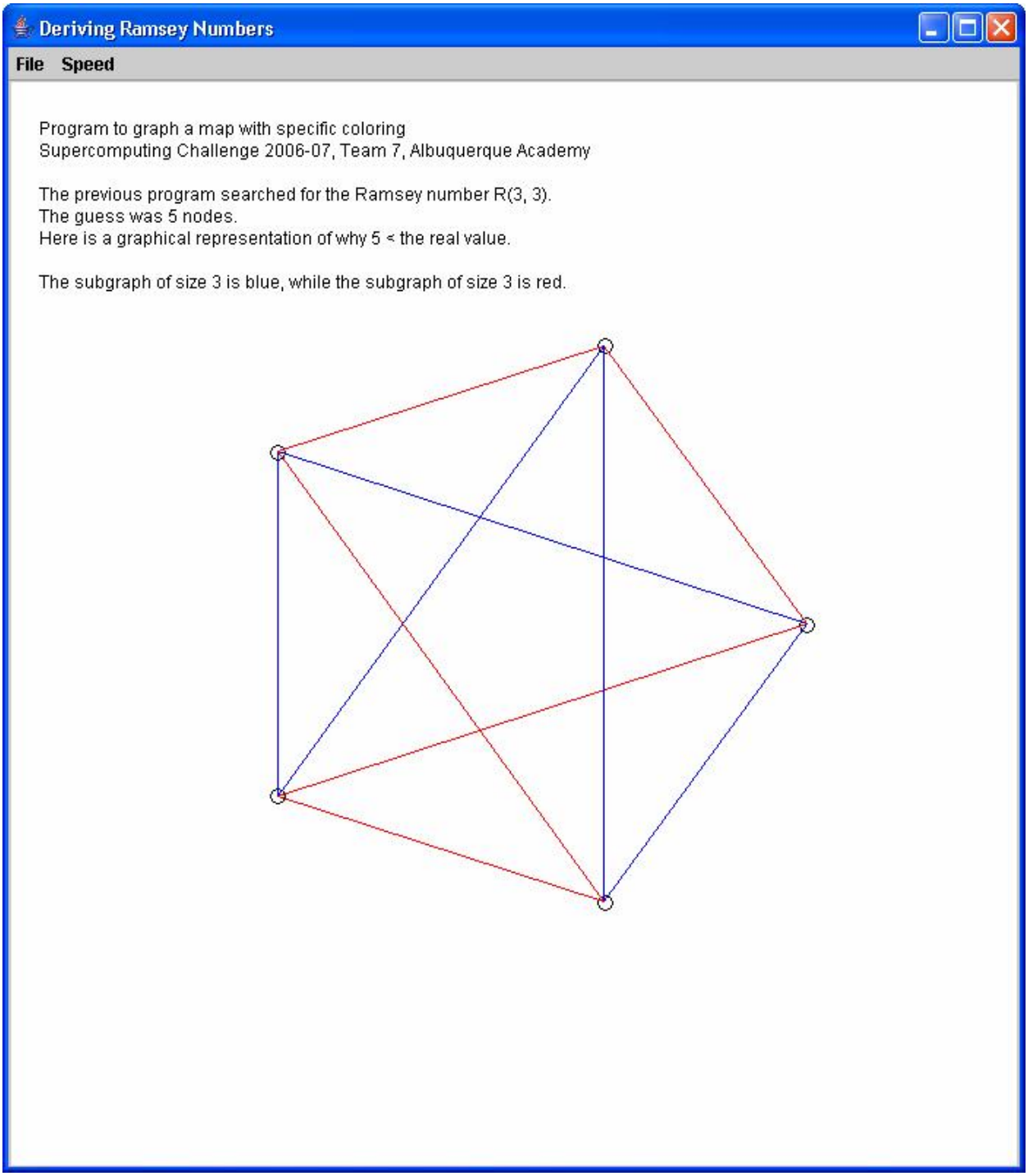


Figure 5: The graph printed from the Edge Sequence Key from Figure 4. Note that as the window's text states, there are no monochromatic k_3's in either colors.

```
Program to Derive Ramsey Numbers
Created for the 2007 New Mexico Supercomputing Challenge.
(c) 2007 Punit Shah and Jack Ingalls. All rights reserved.
Team 7, Albuquerque Academy


BEGINNING OF CONSOLE I/O WINDOW:

We will be searching for subgraphs of size 3 and 3 in a map of size 6
More technically, we are calulating R(3, 3) with a guess of 6.

This graph ONLY has monochromatic colorings. Ramsey number is this number of nodes (6) or lower.
More technically: R(3, 3) <= 6.

END OF CONSOLE I/O WINDOW
```

Figure 6: The console output for a $R(3, 3)$ with a guess of 6. The program does not find a graph without monochromatic colorings, so the Ramsey number is less than or equal to the guess. As Figures 1 and 3-5 showed that $R(3, 3)$ is > 5, and in this figure the outputs shows $R(3, 3) \leq$ the guess 6, so the answer must be six, which it is.

```
Program to Derive Ramsey Numbers
Created for the 2007 New Mexico Supercomputing Challenge.
(c) 2007 Punit Shah and Jack Ingalls. All rights reserved.
Team 7, Albuquerque Academy


BEGINNING OF CONSOLE I/O WINDOW:

We will be searching for subgraphs of size 4 and 3 in a map of size 7
More technically, we are calulating R(4, 3) with a guess of 7.

This graph of 7 nodes has a nonmonochromatic coloring. Ramsey number is definately greater than 7.
More technically: R(4, 3) > 7.

The Edge Sequence Key can be used to display a picture of the graph with no monochromatic subgraphs in it that the
program discovered. Enter this when requested in the appropriate program. The Edge Sequence Key is:
7X4X3X10100110011001111111000

END OF CONSOLE I/O WINDOW
```

Figure 7: The console output with Edge Sequence Key. The inputs were an R(3, 4) with guess 7. The total processing power time with all enhancement algorithms was 9 seconds compared to an unimproved algorithm that ran for at least 13.25 seconds. The Edge Sequence Key is: 7X4X3X101001100110011111000.

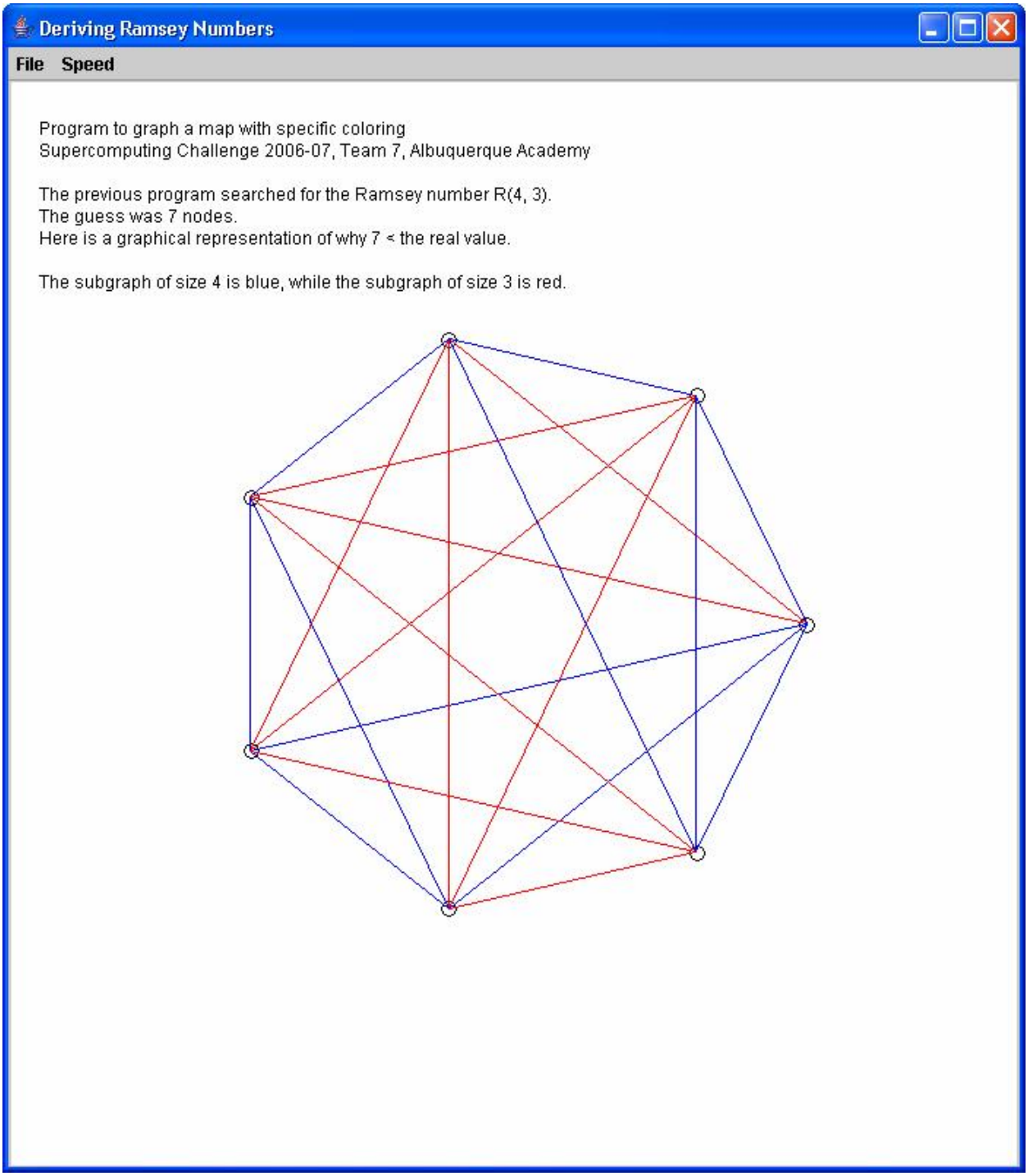


Figure 8: The graphing for the output from Figure 7. In this case, the color of the edges is important to take note of because, here, there may be as many blue k_3 as we need because its subgraph size is 4. There may be no red k_3's, however, as 3 is the size of the subgraph assigned to red.

APPENDIX C: STATED VALUES OF RAMSEY NUMBERS

The following table lists the stated values of Ramsey numbers. The inputs r and b are inputted into function $R(r, b) = n$. The values of n in the form $[a, b]$ are endpoint inclusive ranges of possible values for that Ramsey number.

Source: Wolfram Math World: http://mathworld.wolfram.com/RamseyNumber.html.

r	***b***	***n***
3	3	6
3	4	9
3	5	14
3	6	18
3	7	23
3	8	28
3	9	36
3	10	[40, 43]
3	11	[46, 51]
3	12	[52, 59]
3	13	[59, 69]
3	14	[66, 78]
3	15	[73, 88]
3	16	[79, 135]
3	17	[92, 152]
3	18	[98, 170]
3	19	[106, 189]
3	20	[109, 209]
3	21	[122, 230]
3	22	[125, 252]
3	23	[136, 275]
4	4	18
4	5	25
4	6	[35, 41]
4	7	[49, 61]
4	8	[56, 84]
4	9	[69, 115]
4	10	[92, 149]
4	11	[97, 191]
4	12	[128, 238]
4	13	[133, 291]

4	14	[141, 349]
4	15	[153, 417]
4	16	[153, 815]
4	17	[182, 968]
4	18	[182, 1139]
4	19	[198, 1329]
4	20	[230, 1539]
4	21	[242, 1770]
4	22	[282, 2023]
5	5	[43, 49]
5	6	[58, 87]
5	7	[80, 143]
5	8	[101, 216]
5	9	[121, 316]
5	10	[141, 442]
5	11	[157, 1000]
5	12	[181, 1364]
5	13	[205, 1819]
5	14	[233, 2379]
5	15	[261, 3059]
5	16	[278, 3875]
5	17	[284, 4844]
5	18	[284, 5984]
5	19	[338, 7314]
5	20	[380, 8854]
5	21	[380, 10625]
5	22	[422, 12649]
5	23	[434, 14949]
5	24	[434, 17549]
5	25	[434, 20474]
5	26	[464, 23750]
6	6	[102, 165]
6	7	[111, 298]
6	8	[127, 495]
6	9	[169, 780]
6	10	[178, 1171]
6	11	[253, 3002]
6	12	[262, 4367]
6	13	[317, 6187]
6	14	[317, 8567]
6	15	[401, 11627]
6	16	[434, 15503]
6	17	[548, 20348]
6	18	[614, 26333]
6	19	[710, 33648]

6	20	[878, 42503]
6	21	[878, 53129]
6	22	[1070, 65779]
7	7	[205, 540]
7	8	[216, 1031]
7	9	[232, 1713]
7	10	[232, 2826]
7	11	[405, 8007]
7	12	[416, 12375]
7	13	[511, 18563]
7	14	[511, 27131]
7	15	[511, 38759]
7	16	[511, 54263]
7	17	[628, 74612]
7	18	[722, 100946]
7	19	[908, 134595]
7	20	[908, 177099]
7	21	[1214, 230229]
8	8	[282, 1870]
8	9	[317, 3583]
8	10	[377, 6090]
8	11	[377, 19447]
8	12	[377, 31823]
8	13	[817, 50387]
8	14	[817, 77519]
8	15	[861, 116279]
8	16	[861, 170543]
8	17	[861, 245156]
8	18	[871, 346103]
8	19	[1054, 480699]
8	20	[1094, 657799]
8	21	[1328, 888029]
9	9	[565, 6588]
9	10	[580, 12677]
10	10	[798, 23556]
11	11	[1597, 184755]
12	12	[1637, 705431]
13	13	[2557, 2704155]
14	14	[2989, 10400599]
15	15	[5485, 40116599]
16	16	[5605, 155117519]
17	17	[8917, 601080389]
18	18	[11005, 2333606219]
19	19	[17885, 9075135299]

APPENDIX D: CLASS *RAMSEY*

//Please note that there is a flowchart of the algorithms in this code in Appendix A, Figure 1

```
/*
 Punit Shah and Jack Ingalls
 2007 New Mexico Supercomputing Challenge
 School: Albuquerque Academy
 Team: 7
 Project Name: Deriving Ramsey Numbers
*/

//---------------------------------------------------------------------------------
// This class contains all the essential functions and methods to calculate Ramsey numbers
//---------------------------------------------------------------------------------


import java.util.*;
import javax.swing.*;
import java.math.*;

class Ramsey
{

  private static Debugging debugger = new Debugging();  // object to call debugging
methods/functions


  //---------------------------------------------------------------------------
  // Purpose: to initiate and manage the core processes to calculate Ramsey numbers
  //---------------------------------------------------------------------------
  public static void main(String[] args)
  {
    /* Purpose of program:
     * This program calculates Ramsey numbers in ideally the least amount of time possible
     */

    // initial actions before we start computational processes
    initialAcations();

    // define user-inputted variables

    // number of points in the map; we will be searching for submaps within the map of this size
    int ptsInMap;
```

```
   // variables for holding the sizes of the subgraphs that we will be searching for
   int submapSize1;
   int submapSize2;


   //Get the inputs
   ptsInMap = getInput("How many nodes would you like in the graph?\nThis is essentially a
hypothesis for what you expect this Ramsey number to be.",
                       -1);
   submapSize1 = getInput("We are calculating R(r,b). What would you like the value of r to
be?\n(This is the size of the first subgraph size for which we will search.)",
                         ptsInMap);
   submapSize2 = getInput("We are calculating R(r,b). What would you like the value of b to
be?\n(This is the size of the second subgraph size for which we will search.)",
                         ptsInMap);


   // Set submapSize1 to be the larger of the submaps to improve performance later during the
while loops
   if (submapSize2 > submapSize1)
   {
     submapSize2 = submapSize2 - submapSize1;
     submapSize1 += submapSize2;
     submapSize2 = submapSize1 - submapSize2;
   }


   // report to user the inputted values
   System.out.println("We will be searching for subgraphs of size " + submapSize1 + " and " +
submapSize2 + " in a map of size " + ptsInMap);
   System.out.println("More technically, we are calculating R(" + submapSize1 +", " +
submapSize2 + ") with a guess of " + ptsInMap + ".");



   /************************************************///end of user inputs, start of
setting up essential variables for calculations

   // Create the array representing the points themselves
   boolean theMap[][];
   theMap = new boolean[ptsInMap][];

   // Create the individual boolean arrays that represent the type of connection with other points
(true/false);
   // the arrays will have as many cells as its index so there is not repeated
   // connection information between the two cells involved
```

```
   // NOTE: this will be creating unbalanced arrays

   for (int i = 0; i < ptsInMap; i++)
   {
    theMap[i] = new boolean[i];
   }

   // we are setting all of the connections to last node (index value: ptsInMap -1) to false
   // we can specially rotate the colors of these edges
   for (int i = 0; i < ptsInMap - 1; i++)
   {
    theMap[ptsInMap - 1][i] = false;
   }

   // initialize the object that we will use to get colorings; need to associate it with theMap
   ColoringGenerator colorGen = new ColoringGenerator(theMap, submapSize1, submapSize2);

   // variable to tell us if whether the current coloring has monochromatic graphs or not
   boolean mapColoringHasNoMonochromatics = false;

   // boolean to tell if we have reached the last coloring and need to quit
   boolean notFinished = true;



   /*********************************************************/// start of
computation


   // This is the outermost while loop of the program. This loop cycles the different colorings of
theMap
   while (notFinished)
   {

    // Fill the map with a possible combination; these commands are put into a
    // method so we can easily change the way we fill the map if we choose to use
    // a different algorithm
    notFinished = fillTheMap(theMap, colorGen);


    if (testColoringForASize(ptsInMap, submapSize2, false, theMap) &&
testColoringForASize(ptsInMap, submapSize1, true, theMap))
    {
     mapColoringHasNoMonochromatics = true;
     break;
    }
```

```
    }// while loop to go through all the different colorings


    /*********************************************************///end of computation,
determine findings

    // pass necessary parameters to determine the results of our searching and print the necessary
statements
    printFindings(mapColoringHasNoMonochromatics, ptsInMap, submapSize1, submapSize2,
theMap, colorGen);

  }


//*************************************************************************
**********************//
  //********************************** Other methods and functions
**********************************//

//*************************************************************************
**********************//


  //-------------------------------------------------------------------------------------------------------
--
  // Purpose: to print the essential information about the program and indicate the beginning of
the console i/o window
  //-------------------------------------------------------------------------------------------------------
--
  private static void initialAcations()
  {
    // this method displays essential information about the program to the user
    System.out.println("\n");
    System.out.println("Program to Derive Ramsey Numbers");
    System.out.println("Created for the 2007 New Mexico Supercomputing Challenge.");
    System.out.println("(c) 2007 Punit Shah and Jack Ingalls. All rights reserved.");
    System.out.println("Team 7, Albuquerque Academy");
    System.out.println("\n");
    System.out.println("BEGINNING OF CONSOLE I/O WINDOW:");
    System.out.println("");
  }


  //-----------------------------------------------------------------------------------------
```

```
  // Purpose: to get the users inputs while ensuring that they are legal in the context of the
program
  //-----------------------------------------------------------------------------------------
  private static int getInput(String requestMessage, int maxValue)
  {
    int input;
    input = 0;

    String temp;
    boolean errorMessageDisplayed;

    // have a while loop so we keep asking for value until we get a valid response
    while (input <= 0)
    {
      try
      {
        errorMessageDisplayed = false;

        // Get the input
        temp = JOptionPane.showInputDialog(null, requestMessage, "Deriving Ramsey Numbers ::
Input", JOptionPane.QUESTION_MESSAGE );

        // Following expression will throw a NumberFormatException if temp is does not convert to
an integer (i.e. temp doesn't have just an integer in it)
        input = Integer.parseInt(temp);
      }
      catch (NumberFormatException error)
      {
        reportInvalidInput (error);
        errorMessageDisplayed = true;
      }

      // integers <= 0 will pass the above test, so just check and catch if the input is valid
      if (input <=0 && !errorMessageDisplayed)
      {
        reportInvalidInput (new NumberFormatException());
      }

      // some of the inputs have a max value (like the submaps cannot be larger than the map
itself); this expression ensures this isn't true
      // note: if maxValue = -1, then there is no max value
      if(maxValue != -1 && input >= maxValue)
      {
        reportInvalidInput (new NumberFormatException());
        input = 0;  // resets input so we go continue to go through the while loop
      }
```

```
   }

   return input;
  }


  //-------------------------------------------------------------------------
  // Purpose: to report to the user that they have entered an invalid input
  //-------------------------------------------------------------------------
  private static void reportInvalidInput(NumberFormatException error)
  {

    JOptionPane.showMessageDialog(null, "Invalid input. Please enter an integer > 0. \nNote that
all subgraph sizes must also be less than the number of nodes in the graph (the guess).", "Invalid
input", JOptionPane.ERROR_MESSAGE);

  }


  //-------------------------------------------------------------------------------------------------------
  // Purpose: fills the given map with the next coloring and returns whether we have reached the
last coloring
  //-------------------------------------------------------------------------------------------------------
  private static boolean fillTheMap (boolean theMap[][], ColoringGenerator colorGen)
  {
    if (colorGen.hasMore())
    {

      // This statement will place the next coloring into theMap
      colorGen.nextColoring();

      return true; //was false; not all possibilities have been found
     }
     else if (theMap[theMap.length - 1][theMap.length - 2] == false)
     {
      int con = 0;  // con is short for connection number

      // This while loop will find the leftmost false so that we can change it to a true.
      while (theMap[theMap.length - 1][con])
      {
        con++;
      }
      theMap[theMap.length-1][ con ]=true;
```

```
      // You must also reset colorGen so that it goes through all of its combinations between the
other nodes
       colorGen.reset();

       return true; // Not all possibilities have been found
      }
      else
      {
       return false; // All possibilities have been found
      }

  }


  //-----------------------------------------------------------------------
  // Purpose: to test a coloring to see whether it is monochromatic or not
  //-----------------------------------------------------------------------
  private static boolean testColoringForASize (int ptsInMap, int size, boolean color, boolean
theMap[][] )
  {
    int twoPoints[] = new int[2];
    int twoPointsLocation[];
    CombinationGenerator twoPointSelector;
    CombinationGenerator comboOfPointsGen = new CombinationGenerator(ptsInMap, size);
    int pointCombination[];



    while (comboOfPointsGen.hasMore())
    {
     // get the combination
     pointCombination = comboOfPointsGen.getNext();

     //setup the twoPointSelector CombinationGenerator to check all edges in the submap
     twoPointSelector = new CombinationGenerator(pointCombination.length, 2);



     // RESET submapIsMonochromatic variables for next set of points/submap
     boolean submapIsMonochromatic = true;

     while (twoPointSelector.hasMore())
     {
      // find the position of the two points' indices in the current pointCombination, find
      // the actual indices and save those to a separate array, and then check whether the
      // line between the indices is false
```

```
        // Note that we use different array for the position of the indices in pointCombination
        // and the actual points because twoPointLocation is referenced to the private array
twoPointSelector.a

        twoPointsLocation = twoPointSelector.getNext();
        twoPoints[0] = pointCombination[twoPointsLocation[0]];
        twoPoints[1] = pointCombination[twoPointsLocation[1]];


        // if we find edge that is false record that the submap we are analyzing
        // is not monochromatic with trues; do the opposite if edge is true
        // we can skip the rest of this submap's analysis if we already know that the
        // subgraph is not monochromatic; knowing of more edges will not help us
        if (theMap[ twoPoints[1] ][ twoPoints[0] ] != color)
        {
          submapIsMonochromatic = false;
          break;
        }


      } // while (twoPointSelector.hasMore())


      // check to see if we found that the submap was monochromatic (either true OR false); if it
was,
      // then this coloration of theMap is not free of monochromatic submaps of the user's given
sizes
      if (submapIsMonochromatic == true)
      {
        return false;  // the function will only return false if there IS a monochromatic submap
      }

    } // while (comboOfPointsGen.hasMore())


    return true;  // the function will only return true if there IS NOT a monochromatic submap

  }


  //-----------------------------------------------------------------------------
  // Purpose: to print the findings and then indicate the end of the console i/o window
  //-----------------------------------------------------------------------------
  private static void printFindings(boolean mapColoringHasNoMonochromatics, int ptsInMap,
int submapSize1, int submapSize2, boolean theMap[][], ColoringGenerator colorGen)
  {
```

```
    System.out.println();

    if (mapColoringHasNoMonochromatics == true)
    {
     System.out.println("This graph of " + ptsInMap + " nodes has a coloring with no
monochromatic graphs. Ramsey number is definitely greater than " + ptsInMap + ".");
     System.out.println("More technically: R(" + submapSize1 +", " + submapSize2 + ") > " +
ptsInMap + ".");

     //call method to print the Edge Sequence Key
     printEdgeSequenceKey(ptsInMap, submapSize1, submapSize2, theMap, colorGen);

    }
    else
    {
     System.out.println("This graph ONLY has monochromatic colorings. Ramsey number is this
number of nodes (" + ptsInMap + ") or lower.");
     System.out.println("More technically: R(" + submapSize1 +", " + submapSize2 + ") <= " +
ptsInMap + ".");
    }

    System.out.println("\nEND OF CONSOLE I/O WINDOW");
  }


  //-----------------------------------------------------------------------------------------------------------
----------------
  // Purpose: prints  programmer defined code called the "Edge Sequence Key" for use in later
graphically displaying a map's coloring
  //-----------------------------------------------------------------------------------------------------------
----------------
  private static void printEdgeSequenceKey (int ptsInMap, int submapSize1, int submapSize2,
boolean theMap[][], ColoringGenerator colorGen)
  {
    int[] binarySequence = colorGen.getBinarySequence();

    System.out.println();
    System.out.println("The Edge Sequence Key can be used to display a picture of the graph with
no monochromatic subgraphs in it that the");
    System.out.println("program discovered. Enter this when requested in the appropriate
program. The Edge Sequence Key is:");
    System.out.print(ptsInMap + "X" + submapSize1 + "X" + submapSize2 + "X");

    for (int i = 0; i < binarySequence.length; i++)
    {
     System.out.print(binarySequence[i]);
```

```
    }


    // append the information for the last point as that has been left out of colorGen's
    // binarySequence array for performance/calculating time reductions
    for (int i = 0; i < ptsInMap - 1; i++)
    {
      if (theMap[ptsInMap - 1][i] == true)
      {
        System.out.print("1");
      }
      else
      {
        System.out.print("0");
      }

    }


    System.out.println();

  }


} // END OF CLASS RAMSEY
```

APPENDIX E: CLASS *COLORINGGENERATOR*

```
/*
 Punit Shah and Jack Ingalls
 2007 New Mexico Supercomputing Challenge
 School: Albuquerque Academy
 Team: 7
 Project Name: Deriving Ramsey Numbers
*/

/*
 * All code written by Team 7; algorithm from:
 * Donald E. Knuth, The Art of Computer Programming, Vol. 4, Fascicle 2: Generating All
Tuples and Permutations,
 * (c)2005, Addison-Wesley, p. 10, "Algorithm L"
 */

//---------------------------------------------
// This class generates colorings for the graph
//---------------------------------------------

import java.math.*;

public class ColoringGenerator
{
  private boolean theMap[][];  // once object created, this IS the exact same variable as theMap in
calling class Ramsey
  private long totalColorings;
  private long coloringsLeft;
  private int a[], f[];
  private int j;
  private int edges; // edges is the number of edges in a map with one less node
  private int realEdgeCount;
  private int submapSize1;
  private int submapSize2;
  private int ptsInMap; // this really equals (the real number of points) - 1

  private static Debugging debugger= new Debugging(); // for debugging


  //-------------------------------------
  // Constructor that sets up the object
  //-------------------------------------
  public ColoringGenerator(boolean map[][], int SubmapSize1, int SubmapSize2)
  {
```

```
    theMap = map;  // pass map here to ensure that the new coloring as being applied to same map
at all times and we don't get out of bound errors later
    submapSize1 = SubmapSize1;
    submapSize2 = SubmapSize2;
    ptsInMap = theMap.length - 1;

    //find number of edges
    for (int i = 0; i < ptsInMap; i++)
    {
      edges += theMap[i].length;
    }

    realEdgeCount = edges + ptsInMap;  // adds the number of edges that are connected to the
"hidden" node

    totalColorings = (long)Math.pow(2.0, (double)edges);

    a = new int[edges];
    f = new int[edges + 1];

    reset();

  }


  //------
  // Reset
  //------
  public void reset ()
  {
    for (int i = 0; i < edges; i++)
    {
      a[i] = 0;
      f[i] = i;
    }

    f[edges] = edges;

    coloringsLeft = totalColorings - 1; // subtract one because we always skip the first, all 0
coloring

  }


  //-------------------------------------------------
  // Generate next permutation (algorithm cited above)
```

```
  //--------------------------------------------------
  private int[] getNext ()
  {
    j = f[0];

    if (j == edges)  //breaks the function with last output for last possible and all subsequent
permutation requests
    {
      System.out.println("Last coloring being sent");
      return a;
    }

    //System.out.print("j: " + j + "    ");

    f[0] = 0;
    f[j] = f[j + 1];
    f[j + 1] = j + 1;

    a[j] = 1 - a[j];

    coloringsLeft--;

    //debugger.printInt1DArray(f);

    return a;
  }


  //-----------------------------------
  // Place the next coloring into theMap
  //-----------------------------------
  public void nextColoring()
  {
    //System.out.println("nextColoring called.");

    int coloringListIndex;
    coloringListIndex = 0;

    getNext();

    while (validateColoring())
    {
      getNext();
    }
```

```
   for (int i = 0; i < ptsInMap; i++)
   {
     for(int k = 0; k < theMap[i].length; k++)
     {

       if(a[coloringListIndex] == 1)
       {
         theMap[i][k] = true;
       }
       else
       {
         theMap[i][k] = false;
       }

       coloringListIndex++;
     }
   }

   //debugger.printBool2DArray(theMap);
   //System.out.println("\n\n");

  }


  //-------------------------------------------------------------------------------
  // Counts the number of 0s and 1s in the coloring to determine whether there is even
  // a change for the mapping to have no monochromatic submaps
  //-------------------------------------------------------------------------------
  private boolean validateColoring()
  {
    // returns true if the coloring will definitely have monochromatic submaps based on the
number of zeros and ones.
    // returns false if the coloring should be tested more thoroughly

    if (coloringsLeft == 0)
    {
      return false; // so if this is the last graph we do not get into an infinite loop from the calling
method
    }

    int num0;
    int num1;

    num0 = 0;
    num1 = 0;
```

```
    for (int i = 0; i < a.length; i++)
    {
      if (a[i] == 0)
      {
        num0++;
      }
    }

    num1 = edges - num0;  // calculate num1 this way to usually save time over counting the
number of 1s in a[]

    int min0;
    int min1;

    // submapSize1 is of color true/1; submapSize2 is of color false/0

    min0 = ptsInMap - submapSize1 + 2; // should be +1, but ptsInMap is one less than what it
really is
    min1 = ptsInMap - submapSize2 + 2;

    if (num0 >= min0 && num1 >= min1)
    {
      // both submaps have enough edges for this graph coloring to be one that we need to test
      return false;
    }

    // the coloring does not have the minimum number of 1s or 0s to be a coloring we need to test
more thoroughly
    return true;
  }

  //-----------------------------------------------
  // Indicates whether there are more coloring left
  //-----------------------------------------------
  public boolean hasMore ()
  {
    return coloringsLeft > 0;
  }

  //-----------------------------------------------
  // Return number of combinations not yet generated
  //-----------------------------------------------
  public long getNumLeft ()
  {
    return coloringsLeft;
  }
```

```
  //-----------------------------------
  // Return total number of combinations
  //-----------------------------------
  public long getTotal ()
  {
    return totalColorings;
  }

  //-----------------------------------------------------
  // Return the binary sequence representing the coloring;
  // useful for printing edge sequence key
  //-----------------------------------------------------
  public int[] getBinarySequence()
  {
    return a;
  }


  //code used to test the class

  /* public static void main(String[] args)
    {
      int ptsInMap = 5;

      boolean theMap[][];
      theMap = new boolean[ptsInMap][];

      // Create the individual boolean arrays that represent the type of connection with other points
(true/false);
      // the arrays will have as many cells as its index so there is not repeated
      // connection information between the two cells involved
      // NOTE: this will be creating unbalanced arrays

      for (int i = 0; i < ptsInMap; i++)
      {
        theMap[i] = new boolean[i];
      }

      ColoringGenerator x = new ColoringGenerator(theMap);

      for (int k = 0; k<x.totalColorings; k++)
      {
        //get output
        int output[];
        output = x.getNext();
```

```
      System.out.print("      " + (k+1) +"  ");


      //print output
      debugger.printInt1DArray(output);

      System.out.println();
     }

   }*/


}  // end of class
```

APPENDIX F: CLASS *COMBINATIONGENERATOR*

```
/*
 Punit Shah and Jack Ingalls
 2007 New Mexico Supercomputing Challenge
 School: Albuquerque Academy
 Team: 7
 Project Name: Deriving Ramsey Numbers
*/

/*
 * CombinationGenerator class adapted by Team 7 from:
 * Michael Gilleland, Merriam Park Software, http://www.merriampark.com/comb.htm
 * Permission to use code granted by statement on site:
 * "The source code is free for you to use in whatever way you wish."
 * Original algorithm from:
 * Kenneth H. Rosen, Discrete Mathematics and Its Applications, 2nd edition (NY: McGraw-
Hill, 1991), pp. 284-286.
 */



//-----------------------------------------------
//  Class to systematically generate combinations.
//-----------------------------------------------


import java.math.BigInteger;

public class CombinationGenerator
{
  //----------
  // Variables
  //----------
  private int[] a;
  private int n;
  private int r;
  private BigInteger numLeft;
  private BigInteger total;

  private static Debugging debugger= new Debugging(); // for debugging

  //------------
  // Constructor
  //------------
  public CombinationGenerator (int n, int r)
```

```
{
  // n is the total number of items
  // r is the number of items for included in each combination

  if (r > n)
  {
    throw new IllegalArgumentException ();
  }
  if (n < 1)
  {
    throw new IllegalArgumentException ();
  }

  this.n = n;
  this.r = r;
  a = new int[r];
  BigInteger nFact = getFactorial (n);
  BigInteger rFact = getFactorial (r);
  BigInteger nminusrFact = getFactorial (n - r);
  total = nFact.divide (rFact.multiply (nminusrFact));
  reset ();
}


//------
// Reset
//------
public void reset ()
{
  for (int i = 0; i < a.length; i++)
  {
    a[i] = i;
  }
  numLeft = new BigInteger (total.toString ());
}


//-----------------------------------------------
// Return number of combinations not yet generated
//-----------------------------------------------
public BigInteger getNumLeft ()
{
  return numLeft;
}
```

```
//-----------------------------
// Are there more combinations?
//-----------------------------
public boolean hasMore ()
{
  return numLeft.compareTo (BigInteger.ZERO) == 1;
}


//-----------------------------------
// Return total number of combinations
//-----------------------------------
public BigInteger getTotal ()
{
  return total;
}


//------------------
// Compute factorial
//------------------
private static BigInteger getFactorial (int n)
{
  BigInteger fact = BigInteger.ONE;
  for (int i = n; i > 1; i--)
  {
    fact = fact.multiply (new BigInteger (Integer.toString (i)));
  }
  return fact;
}


//-------------------------------------------------------
// Generate next combination (algorithm from Rosen p. 286)
//-------------------------------------------------------
public int[] getNext ()
{
  if (numLeft.equals (total))
  {
    numLeft = numLeft.subtract (BigInteger.ONE);
    return a;
  }

  int i = r - 1;

  while (a[i] == n - r + i)
```

```
    {
      i--;
    }

    a[i] = a[i] + 1;

    for (int j = i + 1; j < r; j++)
    {
      a[j] = a[i] + j - i;
    }

    numLeft = numLeft.subtract (BigInteger.ONE);

    return a;
  }

}
```

APPENDIX G: CLASS *DEBUGGING*

```
/*
 Punit Shah and Jack Ingalls
 2007 New Mexico Supercomputing Challenge
 School: Albuquerque Academy
 Team: 7
 Project Name: Deriving Ramsey Numbers
*/

//-----------------------------------------------------------------------------------------
// These methods help to debug aspects of the program by printing information to the console
// window or modifying variables to test cases
//-----------------------------------------------------------------------------------------

class Debugging
{
  //private debugging debugger = new debugging();;

  public Debugging()
  {

  }

  public void printInt1DArray (int array[])
  {
    for (int i = 0; i < array.length; i++)
    {
      System.out.print(array[i] + " ");
    }
    System.out.println();
  }

  public void printInt2DArray (int array[][])
  {
    for (int i = 0; i < array.length; i++)
    {
      printInt1DArray(array[i]);
    }
  }

  public void printBool1DArray (boolean array[])
  {
    for (int i = 0; i < array.length; i++)
    {
      System.out.print(array[i] + " ");
```

```
    }
    System.out.println();
  }

  public void printBool2DArray (boolean array[][])
  {
    for (int i = 0; i < array.length; i++)
    {
      printBool1DArray(array[i]);
    }
  }


  public void setWholeArrayToTrue (boolean theMap[][])
  {
    // this method sets all of the values of the array to true
    for (int i = 0; i < theMap.length; i++)
    {
      for (int k = 0; k < theMap[i].length; k++)
      {
        theMap[i][k] = true;
      }
    }
  }

  public void set5NodeMapToNonmonochromaticConfig(boolean theMap[][])
  {
    //all outside painted true; all inside painted false (this is a 5 node map with no monochromatic
3 node submaps
    theMap[1][0] = true;
    theMap[2][0] = false;
    theMap[2][1] = true;
    theMap[3][0] = false;
    theMap[3][2] = true;
    theMap[3][1] = false;
    theMap[3][0] = false;
    theMap[4][0] = true;
    theMap[4][1] = false;
    theMap[4][2] = false;
    theMap[4][3] = true;
  }

}
```

APPENDIX H: GRAPHICS PROGRAM: CLASS *GRAPHPLOTTER*

```
/*
 Punit Shah and Jack Ingalls
 2007 New Mexico Supercomputing Challenge
 School: Albuquerque Academy
 Team: 7
 Project Name: Deriving Ramsey Numbers

 This class is part of the graphing program
*/



//-----------------------------------------------------------------------------
// This class graphs to visually prove that the Ramsey Number is larger that the guess
// Uses the Edge Sequence Key in order to know how to color the map
//-----------------------------------------------------------------------------


import apcslib.*;
import java.awt.*;
import javax.swing.*;


class GraphPlotter

{
  public GraphPlotter() {
    try {
      jbInit();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void main(String[] args)

  {
    // Define main variables to be used through the
program************************************

    // tools for drawing
    DrawingTool pencil;
```

```
    SketchPad paper;
    Color red;
    Color blue;
    Color black;

    // essential information about the map we will graph
    int ptsInMap;
    int submapSize1;
    int submapSize2;
    int numEdges;
    //int binaryColoringSeq[];

    paper = new SketchPad(700,800);
    pencil = new DrawingTool(paper);
    red = new Color(255, 0, 0);
    blue = new Color(0, 0, 255);
    black = new Color(0, 0, 0);

    paper.setTitle("Deriving Ramsey Numbers ");
    setDrawSpeed(paper);

    // Get the Edge Sequence Key
    // NOTE: the key follows this format (no parenthesis in key):
    // (ptsInMap)X(submapSize1)X(submapSize2)X(binaryColoringSequence)
    String ESKey;
    ESKey = JOptionPane.showInputDialog(null, "Please enter the Edge Sequence Key.",
                                "Graph Specific Coloring Sequence",
JOptionPane.QUESTION_MESSAGE);


    // Get the values of the essential values about the graph*******************************

    int xLoc1;
    int xLoc2;
    String binaryColoringSequence; // actually used for this purpose after we have all the values


    // get ptsInMap
    xLoc2 = getNextXLoc(ESKey, 1);
    binaryColoringSequence = ESKey.substring(0, xLoc2);
    ptsInMap = Integer.parseInt(binaryColoringSequence);

    // get submapSize1
    xLoc1 = xLoc2 + 1;
    xLoc2 = getNextXLoc (ESKey, xLoc1);
    binaryColoringSequence = ESKey.substring(xLoc1, xLoc2);
```

```
submapSize1 = Integer.parseInt(binaryColoringSequence);

// get submapSize2
xLoc1 = xLoc2 + 1;
xLoc2 = getNextXLoc (ESKey, xLoc1);
binaryColoringSequence = ESKey.substring(xLoc1, xLoc2);
submapSize2 = Integer.parseInt(binaryColoringSequence);

// get binary coloring sequence
xLoc1 = xLoc2 + 1;
binaryColoringSequence = ESKey.substring(xLoc1);
numEdges = binaryColoringSequence.length();


// Print out some information about the key to the output console I/O window
// mostly printed for debugging purposes
System.out.println(ESKey);
System.out.println(binaryColoringSequence);
System.out.println(ptsInMap);
System.out.println(submapSize1 + "    " + submapSize2);
System.out.println(numEdges);

// Calculate the locations of the nodes*********************************


// first node is always at (1,0)
// location of subsequence nodes calculated by first (partially) getting their polar coordinates
// and then converting to Cartesian coordinates (note: radius = 150)
double coordinates[][];
coordinates = new double[ptsInMap][2];
double theta;

double radialDistance = (2 * Math.PI)/ (double)ptsInMap;

// polorCoordinates[node#][0] = x and polorCoordinates[node#][1] = y
for (int i = 0; i < ptsInMap; i++)
{
  theta = (double) i * radialDistance;

  // covert our polar information into Cartesian coordinates
  coordinates[i][0] = 200.0 * Math.cos(theta);
  coordinates[i][1] = 200.0 * Math.sin(theta);

}
```

```
// Now, actually start graphing************************************

// Plot all the points
for (int i = 0; i < ptsInMap; i++)
{
  pencil.up();
  pencil.move(coordinates[i][0], coordinates[i][1]);
  pencil.down();
  pencil.drawOval(10,10);
}

//draw the lines
// true: blue
// false: red


int index = 0;  // this is the index within the string
//int crntPt = 0; // this is the current node within the graph
int crntChar;


// in the loop, i is the current node within the graph
for (int i = 1; i < ptsInMap; i++)
{
  for(int k = 0; k < i; k++)
  {
    crntChar = binaryColoringSequence.charAt(index);
    if (crntChar == '0')
    {
      pencil.setColor(red);
      drawLine(coordinates[i][0], coordinates[i][1], coordinates[k][0],coordinates[k][1], pencil);
    }
    else // current char is '1'
    {
      pencil.setColor(blue);
      drawLine(coordinates[i][0], coordinates[i][1], coordinates[k][0],coordinates[k][1], pencil);
    }

    index++;
  }

}


// Print vital information about the sequence onto the screen
pencil.up();
```

```
    pencil.move(-325, 350);
    pencil.down();

    pencil.setColor(black);
    drawString("Program to graph a map with specific coloring", pencil);
    drawString("Supercomputing Challenge 2006-07, Team 7, Albuquerque Academy", pencil);
    drawString("", pencil);
    drawString("The previous program searched for the Ramsey number R(" +submapSize1 + ",
"+ submapSize2 + ").", pencil);
    drawString("The guess was " + ptsInMap + " nodes.", pencil);
    drawString("Here is a graphical representation of why " +ptsInMap + " < the real value.",
pencil);
    drawString("", pencil);
    drawString("The subgraph of size " + submapSize1 + " is blue, while the subgraph of size " +
submapSize2 + " is red.", pencil);


  }


  //-------------------------------------------------
  // Purpose: sets the drawing speed to instantaneous
  //-------------------------------------------------
  private static void setDrawSpeed(SketchPad paper)
  {
    // Gets to the speed button and changes the draw speed to "no delay"
    JMenuBar menu = paper.getJMenuBar();
    JMenu speed = menu.getMenu(1);
    JMenuItem noDelay = speed.getItem(3);
    noDelay.doClick();
  }


  //-----------------------------------------------------------------------------------------------------
  // Purpose: finds the index in the string where the next X is separating the values of the different
variables
  //-----------------------------------------------------------------------------------------------------
  private static int getNextXLoc (String ESKey, int startingXLoc)
  {
    while (ESKey.charAt(startingXLoc) != 'X')
    {
      startingXLoc++;
    }

    return startingXLoc;
  }
```

```
  //---------------------------------------------------------------
  // Purpose: draws a line between the two points passed to the method
  //---------------------------------------------------------------
  private static void drawLine(double x1, double y1, double x2, double y2, DrawingTool pencil)
  {
    pencil.up();
    pencil.move(x1, y1);
    pencil.down();
    pencil.move(x2,y2);

  }


  //-----------------------------------------------------------------------------------------------
  // Purpose: draws a string and then automatically moves the DrawingTool down 15 pixels after
every line
  //-----------------------------------------------------------------------------------------------
  private static void drawString (String theString, DrawingTool pencil)
  {
    pencil.up();
    pencil.setDirection(-90);
    pencil.forward(15);
    pencil.down();
    pencil.drawString(theString);

  }


  //---------------------
  // Method part of class
  //---------------------
  private void jbInit() throws Exception
  {
  }

}
```