2000

Graph coloring heuristics from investigation of smallest hard to color graphs

Andrew Radin

Graph Coloring Heuristics from Investigation of Smallest Hard to Color Graphs

MS Thesis

Andrew A. Radin

Rochester Institute of Technology

Computer Science Department

May 16, 2000

Advisor: Prof. Stanisław P. Radziszowski

Reader: Prof. Edith Hemaspaandra

Observer: Prof. Peter G. Anderson

External Reader: Prof. Marek Kubale
Technical University of Gdańsk, Poland

Copyright Statement

Andrew A. Radin 3/30/01
date

Abstract

Vertex coloring of graphs is an NP-complete problem. No polynomial time algorithm is known to color graphs optimally. The best we can do to handle vertex coloring of graphs is to create heuristics which provide a guess as to an optimal coloring.

This thesis examines a number of known vertex coloring heuristics, and compares their performance to a brute-force optimal coloring. These comparisons are made for relatively small graphs with low numbers of vertices.

The behaviors of the existing heuristics is examined to aid in the creation of new heuristics. The new heuristics are compared against the existing heuristics for both all small ($n \leq 12$) and relatively large random graphs.

The result of this thesis is two new graph coloring heuristics. The first heuristic, the so called *double interchange*, provides the best coloring performance of the heuristics studied for small, connected graphs. The second heuristic, the *annealing interchange*, provides the best coloring performance of the heuristics studied for larger, random graphs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A graph G consists of a non-empty set of elements, called vertices, and a list of unordered pairs of these elements, called edges. If v and w are vertices of G and the edge-set contains a vw or wv pair, then the vertices v and w are said to be adjacent [20].

An assignment of colors (or elements of some set) to the vertices of graph G, one color to each vertex is called coloring, if adjacent vertices are assigned different colors. Coloring with k colors is called k-coloring. A graph is said to be k-colorable if there exists an l-coloring of G such that $l \leq k$. If G is k-colorable but not $(k-1)$ colorable, we say the *chromatic number* of G is k and write $\chi(G) = k$ [11].

The problem of answering if a graph G is k-colorable is known to be an NP-complete problem. Problems that are NP-complete have no known polynomial time algorithm to solve them [5]. Currently, the best way to handle solving NP-complete problems is to approximate the solution using heuristics.

A proof that COLORABILITY problem is NP-complete is now given. A problem is considered to be NP-complete if the problem is in NP and there exists another problem known to be NP-complete that is polynomial time reducible to it. The 3-satisfiability problem (3-SAT) is known to be NP-complete [9]. The proof that 3-SAT is reducible to COLORABILITY taken from Aho [1] now follows.

Given an expression F in 3-CNF with n variables and t factors, we show how to construct, in time polynomial MAX(n, t), an undirected graph $G = (V, E)$ with $3n + t$ vertices, such that G can be colored with $n + 1$ colors if and only if F is satisfiable.

Let $x_1, x_2, ..., x_n$ and $F_1, F_2, ..., F_t$ be the variables and factors and F,

respectively. Let $v_1, v_2, ..., v_n$ be new symbols. Without loss of generality assume $n \geq 4$ since any expression in 3-CNF with three or fewer distinct variables can be tested directly for satisfiability in time linear in the length of the expression without use of the transformation to colorability. The vertices of G are:

1. $x_i, \bar{x_i}$, and v_i, for $1 \leq i \leq n$, and
2. F_i, for $1 \leq i \leq t$.

The edges of G are:

1. all (v_i, v_j) such that $i \neq j$,
2. all (v_i, x_j) and $(v_i, \bar{x_j})$ such that $i \neq j$,
3. $(x_i, \bar{x_i})$ for $1 \leq i \leq n$
4. (x_i, F_j) if x_i is not a term of factor F_j, and $(\bar{x_i}, F_j)$, if $\bar{x_i}$ is not a term of F_j.

The vertices $v_1, v_2, ..., v_n$ form a complete subgraph of n vertices and hence require n distinct colors. Each x_j and $\bar{x_j}$ is connected to each v_i, $i \neq j$, and hence x_j and $\bar{x_j}$ cannot be the same color as any of the v's, except possibly v_j. Since x_j and $\bar{x_j}$ are adjacent, they cannot be the same color, so G cannot be colored with $n+1$ colors unless one of x_j and $\bar{x_j}$ is the same color as v_j and the other is a new color which we refer to as the *special color.*

Think of that one of x_j or $\bar{x_j}$ colored with the special color as being assigned the value 0. Now consider the color assigned to the F_j vertices. F_j is adjacent to at least $2n-3$ of the $2n$ x_i's and $\bar{x_i}$'s. Since we assume $n \geq 4$, for each j there exists an i such that F_j is adjacent to both x_i and $\bar{x_i}$. Since one of x_i or $\bar{x_i}$ is colored with the special color, F_j cannot be colored with the special color.

If F_j contains some literal y, where $\bar{y}$ has been assigned the special color, then F_j is not adjacent to any vertex colored the same as y and hence may be assigned the same color as y. Otherwise, a new color is needed. Thus, all the F_j's can be colored with no additional colors if and only if there is an assignment of the special color to the literals such that each factor contains some literal y where $\bar{y}$ has been assigned the special color that is, if and only if one can assign values to the variables so that each factor contains a y assigned the value 1 ($\bar{y}$ assigned the value 0), i.e. if and only if F is satisfiable [1]. This completes the proof.

A number of approximation algorithms have been developed for coloring graphs [2, 5, 11, 18]. On the whole, the performance of graph coloring heuris-

tics is studied by giving asymptotic results. These are usually the worst-case performance guarantee and the worst-case complexity. Both functions tell us what one can expect as the number of vertices $n \to \infty$, but we don't know what is going on at the other end of the scale, say when $n \leq 10$. For these reasons, Hansen and Kuplinsky have introduced the concept of hard-to-color graphs [6, 8]. The study of such graphs makes it possible to design improved algorithms trying to avoid hard instances as far as possible [12].

Chapter 2

Heuristics Explained

In this chapter the graph coloring heuristics examined in this thesis are explained in detail. The heuristics are broken into two groups. The first group are the known heuristics. These heuristics existed before the creation of this thesis, and have been published elsewhere. The second group of heuristics are new heuristics that were created as a result of investigative work associated with this thesis.

The known heuristics that were chosen all share the common property that smallest hard-to-color graphs have been discovered for them. Hard-to-color graphs are graphs in which every application of a particular coloring algorithm (i.e. no matter what choice is made to break ties) results in a nonoptimal coloring. Hard-to-color graphs with relatively small values of n are the so called smallest hard-to-color graphs [11].

Other graph coloring heuristics exist, but were not examined because smallest hard-to-color graphs were not readily available for these heuristics [7].

2.1 Known Heuristics

The known heuristics were chosen largely in part from work done by Marek Kubale. Professor Kubale provides the smallest hard to color graphs for a number of known graph coloring heuristics. This catalog of smallest hard to color graphs was a corner stone in generating new heuristics [11, 12, 13].

```
procedure S(G);
begin
   order the vertices of G in some sequence v1, v2,... vn;
   for i := 1 to n do
      color vi with the least possible color
end
```

Figure 2.1: Sequential Coloring Pseudo-Code [11]

2.1.1 Sequential Algorithms

In a sequential algorithm the vertices of a graph are ordered in some way and then colored in the given order greedily. A greedy coloring is to assign the least possible color to each node. The coloring can be static or dynamic. With static coloring the ordering is done completely before the coloring. In dynamic coloring, the ordering can be changed during the coloring phase.

The general description of a sequential algorithm is described in Figure 2.1.

Unordered Sequential

The unordered sequential (US) algorithm relies on the graph generation tool to provide the order of vertices. Nodes are colored in the order that they appear in memory using the general sequential coloring algorithm. The US algorithm is static.

Random Sequential

The random sequential (S) algorithm provides a pseudo-random ordering of nodes and then colors greedily using the general sequential coloring algorithm. The S algorithm is static.

Largest First

The largest first (LF) method orders the vertices in a non-increasing order of their degree. This algorithm was first given by Welsh and Powell in a slightly different but equivalent form [19]. The LF algorithm is static.

Smallest Last

In the smallest last (SL) algorithm, vertices are ordered v_1, v_2, ..., v_n, so that v_i, $i = 1, 2, \ldots, n$ has the minimum degree in the subgraph of G

induced by vertices $v_1, v_2, \ldots, v_i$, in symbols $G(v_1, \ldots, v_i)$. The SL coloring algorithm and various refinements to it are from Matula et. al. [14]. The SL algorithm is static.

Saturation Largest First

The saturation largest first (SLF) algorithm repeatedly colors an uncolored vertex of the largest saturation degree with the smallest possible color. The saturation degree of a vertex is the number of adjacent distinctly colored vertices. Ties for coloring nodes are broken by choosing the vertex with the greatest degree. The SLF algorithm is dynamic.

The SLF algorithm was proposed by Brélaz as DSATUR [4].

2.1.2 Sequential Interchange Algorithms

A refinement to sequential algorithms is given by the interchange procedure (I). Whenever a new color is going to be used for a vertex v, consider all pairs of i and j of already colored vertices. Then, colors assigned to i, j pairs are interchanged in some way such that v can be colored without using a new color, if that is possible.

Although the I procedure is intended to improve sequential coloring, in some cases it may produce worse solutions. Nevertheless, extensive computer experiments have shown that, on the average, sequential interchange algorithms result in a reduction of the number of colors used [11].

This thesis provided interchange procedures for the S, LF, SL and SLF algorithms. The interchange versions are referred to as SI, LFI, SLI and SLFI, respectively.

2.1.3 Independent Set Algorithms

In an independent set algorithm, the nodes are considered in some order. Node v is colored with color i whenever v has no neighbors colored i. When no more nodes can be colored with i, a new color is chosen and the assignment continues until all nodes are colored.

The vertices of color i form an independent set in $G(V - V_1 - ... - V_{i-1})$, where $V_1, ..., V_{i-1}$ are sets of vertices with color $1, ..., i-1$, respectively.

The pseudo-code for the independent set algorithm is given in figure 2.2.

```
procedure IS(G);
begin
   order the vertices in a sequence v1, v2,... vn;
   i :=1;
   while not all vertices colored do
   begin
      for j := 1 to n do
         if vj is uncolored and not adjacent to any vertex colored i
         then color vj with i;
   end
   i := i +1
end
```

Figure 2.2: Independent Set Coloring Pseudo-Code [11]

Random Independent Set

The random independent set (IS) algorithm provides a pseudo-random ordering of nodes and then colors the graph using the general independent set coloring algorithm.

Greedy Independent Set

The greedy independent set (GIS) algorithm provides a method for choosing vertices for consecutive independent sets. Starting with $V_1 = \emptyset$, set V_1 is augmented at each step by a minimum degree vertex of the subgraph generated by the non-neighbors of the current members of V_1. The process is repeated for V_2 in subgraph $G(V - V_1)$, etc.

The GIS algorithm is attributed to Johnson [10].

2.2 New Heuristics

A method of tracing algorithm behavior against catalogs of smallest hard to color graphs was used to better understand the behavior of existing heuristics. This understanding was then used to devise experimental heuristics whose focus was to out-perform the existing heuristics in as many circumstances as possible. Focus was given to providing best performance for sets of all connected graphs for small values of n, and sets of random graphs for large values of n.

A combination of experimentation coupled with theory revealed that the LF and SLF algorithms tend to produce the best coloring results. Analytical focus was given to these two algorithms in hopes that they could be improved through augmentation of their basic behaviors.

Although a number of improvement methods were investigated and experimented with, this thesis will only describe two new heuristics. The double interchange heuristic is described because it provides (on the whole) the best coloring performance on small graphs. The annealing interchange algorithm provides the best performance for larger, random graphs.

2.2.1 Double Interchange

The double interchange procedure goes beyond the (single) interchange procedure by attempting two i, j pair color swaps if single i, j pair color swaps do not avoid adding a new color to the graph.

Double interchange was attempted with LF and SLF algorithms. The resulting algorithms are named largest-first double interchange (LFDI) and saturation largest-first double interchange (SLFDI), respectively.

2.2.2 Annealing Interchange

Recognizing that increasing the number of pair interchanges improved algorithm performance, a general case interchange procedure was created. This so called annealing interchange attempts provide c swap interchanges where c is defined by an annealing constant. The annealing constant is the number of iterations of the interchange that is determined to be feasible for the particular problem set.

The annealing interchange differs slightly from the general interchange case by attempting a color swap with a random neighbor, and then re-coloring the neighbor. Re-coloring is achieved by choosing the lowest ordered color that provides a valid coloring. This process repeats until no new color is required to color the neighbor or the number of iterations has reached the annealing constant.

As with double interchange, the LF and SLF algorithms were enhanced with an annealing interchange procedure. The resulting algorithms are named largest-first annealing interchange (LFAI) and saturation largest-first annealing interchange (SLFAI), respectively.

Figure 2.3 shows the pseudo code for the annealing interchange.

```
procedure AI(G, v);
begin
   for i := 1 to ANNEALING CONSTANT do
      choose a random neigbor of v, named r
      if assigning the color of r to v and uncoloring r produces no conflicts
      then begin
         if COLOR(r) requires no new color then exit
         v := r
   end
end
```

Figure 2.3: Annealing Interchange Procedure Pseudo-Code

Chapter 3

Experimental Method

We start this chapter with a discussion on how the two classes of graphs to be colored were generated. The two classes of graphs are all connected graphs of $n \leq 12$ and random graphs of varying edge densities for certain values of n.

This chapter also explains how both $\chi(G)$ and the estimates provided by the heuristics were calculated. The end of the chapter provides some detail on how the large number of calculations for $n = 12$ were performed.

3.1 All connected graph generation

Brendan McKay's *makeg* was used to create all connected graphs for $n \leq 12$. The *makeg* tool produces y-format graphs separated by line feeds. The *makeg* tool will only create graphs for a single value of n per invocation. As a result, *makeg* was run repeatedly with varying command line arguments [15].

The *makeg* tool also has the capability to create portions of a suite of connected graphs. This feature was used for handling the large number of graphs produced with $n = 12$.

An example of how *makeg* is used is given in figure 3.1.

3.2 Random graph generation

Random graphs are generated by first creating graphs of n nodes with no edges. Connection between i, j pairs of vertices are made randomly with probability p. This probability is the so called *edge density.*

```
>prompt% makeg -c 4
>A n=4 e=3:6 d=3 class=1/0
DG
DV
DW
D^
D_
D
>Z 6 graphs generated in 0.01 sec
```

Figure 3.1: Example Usage of *makeg*, all connected graphs, $n = 4$

When populating graphs with random edges, clumps of connected subgraphs tend to form. The higher the edge density the lower the number of these connected subgraphs. An edge density of 100% produces a single (completely) connected graph, where an edge density of 0% produces n connected subgraphs, each only containing one node [3].

For simplicity there was no attempt to connect subgraphs contained within the generated random graphs. Some random graphs generated may have been connected while others were unconnected.

3.3 $\chi(G)$ calculation

Calculation of the actual $\chi(G)$ is handled by a re-usable graph theory utility library provided by Stanisław Radziszowski. The calculation of $\chi(G)$ is performed as follows: First, all independent sets of the graph to be colored are generated, and a table of maximal sets is made. Now, graph G can be colored with k colors if and only if its vertices can be covered with k maximal independent sets (MIS). The latter property is tested by a cover function, which descends recursively, by fixing one MIS and calling cover with the appropriately modified arguments [17].

Section ?? details the implementation of this algorithm in C code.

3.4 Heuristic estimate calculation

All reasonable attempts were made to write C code that accurately reflects the behavior of the studied heuristic. However, due to the nature of combi-

natorics and the realities of translating theory into practice, some inconsistencies are present.

Some heuristics are non-deterministic in their execution. For example, if two nodes have the same degree, the ordering provided by LF can occur in more than one way. No attempt was made to examine all permutations of breaking these ties.

Some aspects of heuristic implementation were modified from the theoretical descriptions to aid in speed of calculations. One example of such a modification is the pre-calculation of vertex degrees. The pre-calculation loaded a table with vertex degree values. The use of this table decreased execution time while maintaining heuristic behavior.

To aid in validation of heuristics, check points were included in the code to verify that colorings produced by the heuristics were in fact valid. This validation process traversed the colored graphs and checked that all nodes were colored and that no adjacent nodes shared the same color. An additional safety check verified that the estimated $\chi(G)$ provided by each heuristic was equal to or greater than the calculated $\chi(G)$, if provided.

3.5 Software Usage and Interaction

The software execution environment consists of two executables, *makeg* and *chicheck*. The *makeg* tool is provided by Brandan McKay, and is described in section 3.1 in this document. The *chicheck* tool was created for this thesis.

3.5.1 Input Graphs

The *makeg* tool was designed to handle two types of graphs. The first type of graph is y-format graphs on standard input. In this form, *chicheck* uses a command line argument of '-' to indicate that y-format graphs should be read from standard input. Figure 3.2 provides an example. This figure also shows example output, which is discussed shortly.

The second form of input graphs used by *chicheck* is random graphs. A random graphs generator exists within *chicheck*. The internal random graph generator was chosen because significantly large graphs cannot be represented in y-format (there is a one-byte restriction on number of nodes). An internal random graph generator also provides a performance improvement by directly writing the random graphs to the same memory space that the heuristics use. This direct write reduces the overhead of reading from pipe and decoding the y-format data.

```
prompt% makeg -c 4 | chicheck -
>A n=4 e=3:6 d=3 class=1/0
>Z 6 graphs generated in 0.01 sec
100.00%    Chi 6/6
100.00%     LF 6/6
100.00%     SL 6/6
100.00%    SLF 6/6
100.00%     SI 6/6
100.00%    LFI 6/6
100.00%    SLI 6/6
100.00%   SLFI 6/6
100.00%   LFDI 6/6
100.00%  SLFDI 6/6
100.00%   LFAI 6/6
100.00%  SLFAI 6/6
```

Figure 3.2: Example Usage of *chicheck*; graphs from standard input

In random graph generation form, *chicheck* takes three command line arguments. The first argument is the number of random graphs to generate. The second is the number of nodes each graph should have. The last value is the edge density, given as a whole number percentage value.

Figure 3.3 shows an example of random graph usage.

3.5.2 Choosing Heuristics

The algorithms to be run by *chicheck* are controlled by environment variables. If the environment variable associated with the heuristic is not set it is assumed to be enabled. Disabling algorithms is achieved by setting the appropriate environment variable for the algorithm to the string 'off'. Table 3.1 provides detail on the environment variable names associated with algorithms.

For example, to disable $\chi(G)$ calculation under bourne shell:

```
prompt% Chi=off ; export Chi
```

```
prompt% chicheck 100 8 85
100.00%    Chi 100/100
100.00%     LF 100/100
 98.00%     SL 98/100
 99.00%    SLF 99/100
 97.00%     SI 97/100
100.00%    LFI 100/100
 98.00%    SLI 98/100
100.00%   SLFI 100/100
100.00%   LFDI 100/100
100.00%  SLFDI 100/100
100.00%   LFAI 100/100
 99.00%  SLFAI 99/100
```

Figure 3.3: Example Usage of *chicheck*; random graphs

enironment variable	algorithm
Chi	$\chi(G)$
US	Unordered Sequential
S	Random Sequential
SL	Smallest-Last
LF	Largest-First
SLF	Saturation Largest-First
SI	Random Sequential Interchange
SLI	Smallest-Last Interchange
LFI	Largest-First Interchange
SLFI	Saturation Largest-First Interchange
IS	Independent Set
GIS	Greedy Indepndent Set
SLFDI	Saturation Largest-First Double Interchange
LFDI	Largest-First Double Interchange
SLFAI	Saturation Largest-First Annealing Interchange
LFAI	Largest-First Annealing Interchange

Table 3.1: Algorithm disabling environment variables.

3.5.3 Reporting Options

The *chicheck* tool provides two types of run reports. The first type is a brief format. This report only shows the percentage and ratio of optimal colorings for each algorithm. A graph is considered to be optimally colored if the algorithm produces the lowest number of colors when compared to the other algorithms. A tie for lowest color is considered an optimal coloring. Figures 3.2 and 3.3 both show a brief report.

The second type of report is a full report. The full report shows the percentages and ratios just as the brief report. However, the full report adds a histogram of colorings as well as information on how accurate the colorings were.

The histogram records the number of graphs that were colored with a particular number of colors. The accuracy is described as the difference between the best $\chi(G)$ calculation and the estimate that each algorithm comes up with.

Figure 3.4 gives an example of a full report. In this example SI is compared to the actual $\chi(G)$. We can see that four graphs were not colored optimally, and the estimate provided by the SI algorithm was one off in each unoptimal coloring.

A full or brief report can be obtained from *chicheck* by setting the REPORT environment variable to 'brief' for a brief report and to 'full' for a full report.

3.6 Distributed Calculations

Special consideration was necessary for all connected graphs on 12 nodes. Current computing capabilities dictate that these calculations take approximately nine years on a single CPU computer.

Through the use of *autoson* [16] calculations for all connected graphs on 12 nodes were distributed to over one hundred Sun SPARC workstations in the computer science labs at the Rochester Institute of Technology. The *makeg* tool allows graph generation to be split via a modulus function. The *autoson* tool was used to drive ten thousand instances of *makeg* which in turn piped output to *chicheck*. One output file per instance was created, and all ten thousand output files were combined via a PERL script to produce the final results.

```
prompt% chicheck 100 8 85
100.00%    Chi 100/100
 96.00%     SI 96/100

       Chi:
 2 colorings:          0   1 off:          0
 3 colorings:          0   2 off:          0
 4 colorings:          6   3 off:          0
 5 colorings:         42   4 off:          0
 6 colorings:         43   5 off:          0
 7 colorings:          9   6 off:          0

        SI:
 2 colorings:          0   1 off:          4
 3 colorings:          0   2 off:          0
 4 colorings:          6   3 off:          0
 5 colorings:         39   4 off:          0
 6 colorings:         45   5 off:          0
 7 colorings:         10   6 off:          0
```

Figure 3.4: Example of full report from *chicheck*

Chapter 4

Experimental Results

4.1 Discussion

The raw data obtained from running *chicheck* is available in Appendix B. In the appendix, full reports are given in all of the data sets except all connected graphs on 12 nodes. This special case required distributed calculations to be made on ten thousand subsets of all connected graphs of 12 nodes. Combination of the results for this experiment was kept simple by only providing a brief report.

Connected graphs of $n \leq 4$ were optimally colored by all algorithms. No experimental data is provided for these graphs in the appendix. Connected graphs of $n = 13$ were too numerous to run an exhaustive enumeration of all. Graphs of $n \geq 14$ are not tractable with current computer technology.

All heuristics are run in each of the experiments. $\chi(G)$ is calculated for all of the experiments except for random graphs on 300 nodes. Also, the color distribution table on 300 nodes is not complete. Large graphs can produce data that is beyond the scope of the reporting mechanism.

Other experiments for random graphs using different input variables were run, but are not reported here. Random graphs provide an almost infinite number of experiments to be performed. The results for random graphs given in this thesis are representative of the information provided by other undocumented experiments.

4.2 Data Plots

Figure 4.1 shows a plotting of performance percentage against the size of small graphs. Since the interchange heuristics are all very close in their

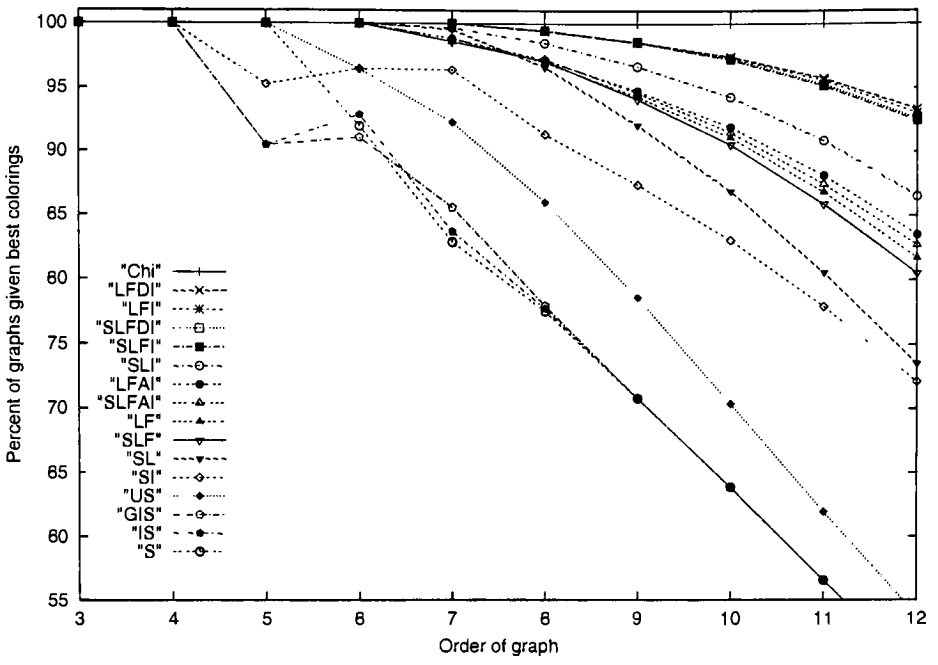

Figure 4.1: Performance comparison for all connected graphs

performance, a second plotting of just these heuristics is given in figure 4.2 to provide more detail.

Figures 4.3 and 4.4 plot the performance characteristics of 23 and 300 node random graphs, respectively. Note that with 300 nodes, $\chi(G)$ is not calculated and the percentage of best colorings is compared against the lowest coloring estimate, no the actual $\chi(G)$.

In figures 4.1 and 4.2 the key is ordered by best to worst performers for the largest graph plotted. The key in figures 4.3 and 4.4 is similarly arranged for the densest graphs plotted.

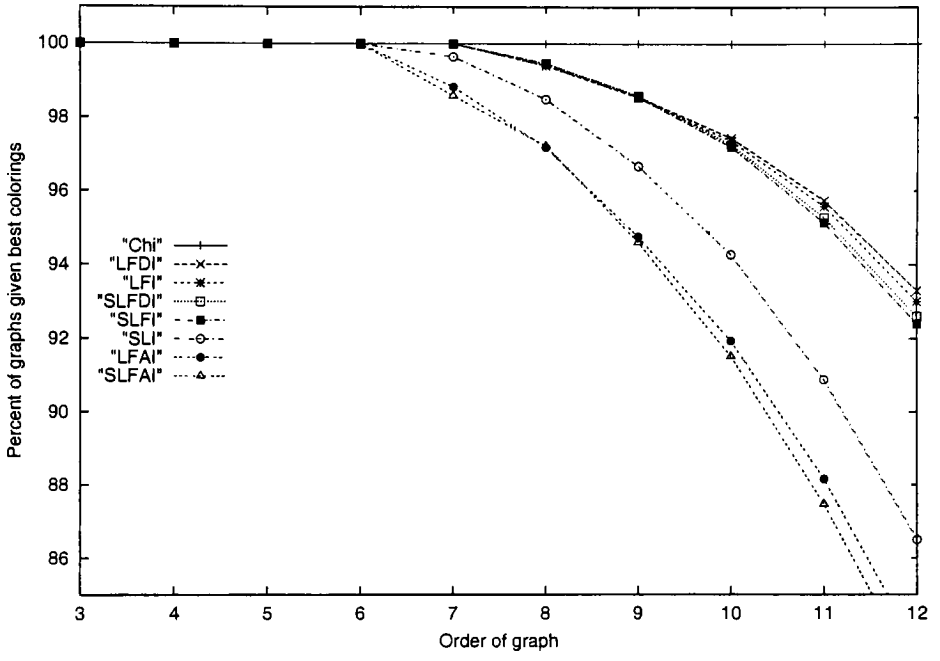

Figure 4.2: Performance comparison for all connected graphs, interchange only

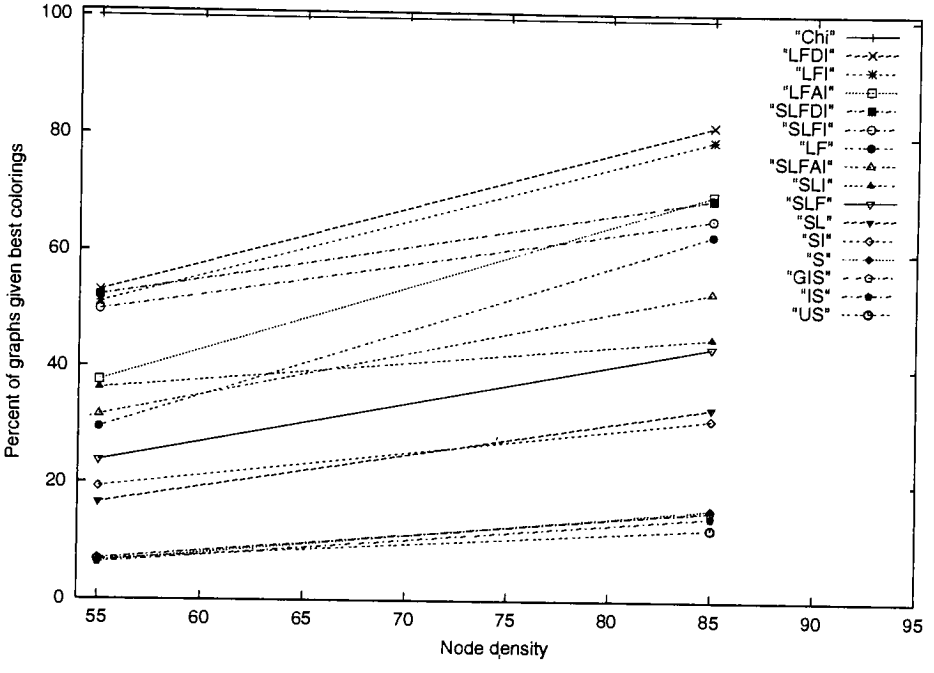

Figure 4.3: Performance comparison for 23 node random graphs

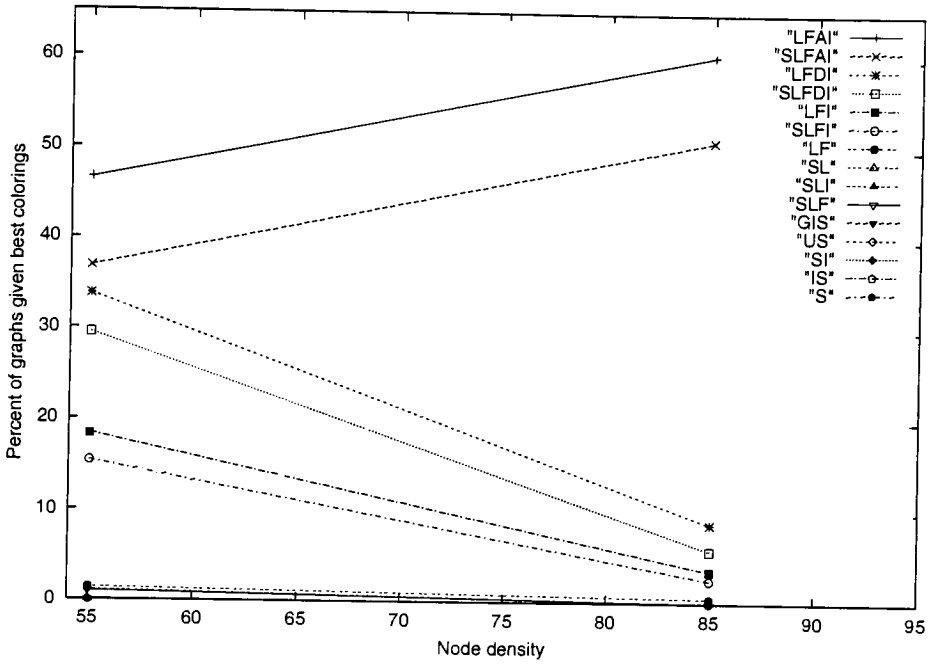

Figure 4.4: Performance comparison for 300 node random graphs

Chapter 5

Results Analysis

This chapter discusses the experimental results.

5.1 All connected Graphs, known χ(G)

Results are given for all connected graphs for graphs $5 \leq n \leq 12$. For each of these sets of graphs, $\chi(G)$ is compared to the upper bound for $\chi(G)$ provided by the heuristics.

5.1.1 Less than 5 nodes

In these experiments all connected graphs of $n \leq 4$ were colored optimally by all algorithms. No data is given for these graphs.

5.1.2 5 nodes

Connected graphs on 5 nodes show the first signs of unoptimal colorings. In table B.1 we see that SI, IS and GIS do not color optimally. All unoptimal colorings are only one less than optimal, as show in table B.3. Distributions of colors are shown in table B.2.

5.1.3 6 nodes

As seen in table B.4, with 6 nodes US and S join SI, IS and GIS in some unoptimal colorings. As with 5 nodes, all unoptimal colorings are only one less than optimal, as shown in table B.6. Table B.5 shows the distributions of colorings.

5.1.4 7 nodes

Now with 7 nodes only LFI, SLFI, LFDI and SLFDI color optimally, as seen in table B.7. However, unoptimal colorings are still only one off as seen in table B.9. Distributions are found in table B.8.

5.1.5 8 nodes

With 8 nodes we find the first set of connected graphs where not one heuristic provides an optimal coloring. Tables B.10 provides the details. These graphs are the first to produce more than one off errors in estimates, with S, IS and GIS, as seen in B.12. Table B.11 show the distributions.

In this experiment, SLFDI and SLFI are tied for the best colorings, followed by SLFDI and then LFI.

5.1.6 9-12 nodes

Connected graphs with 9 or more nodes provide more interesting data. With these graphs the accuracy and performance of the heuristics begin to break down. Tables B.13 through B.22 show the details.

With 9 nodes, SLFDI is the winner followed by LFDI. However, with 10, 11 and 12 nodes, SLFDI moves to third place behind LFI. In these graphs LFDI provides the best colorings.

Notice that with these graphs LFAI and SLFAI are midway in the field of heuristics.

5.2 Random Graphs, known $\chi(G)$

Calculating $\chi(G)$ for all graphs $n \geq 24$ proved to be intractable by simple means. As a result, maximal comparisons for heuristics against $\chi(G)$ were done for $n = 23$.

Experiments were run with a varying number of edge densities, but edge densities of 55% and 85% were arbitrarily chosen to be reported in this thesis.

5.2.1 23 nodes, 55% dense

This experiment saw similar characteristics to that of the all connected graphs. LFDI was the best performer with SLFDI beating LFI for second place. Table B.23 gives the details.

With 23 nodes we see that some heuristics are providing increasingly worse estimates. Four of the heuristics give estimates that are as much as four off the actual value of $\chi(G)$ as seen in table B.25.

5.2.2 23 nodes, 85% dense

Increasing the node density on 23 nodes produced some interesting results. LFDI is still the leader, but the LFAI annealing algorithm has moved from the mid-position of the previous experiments into third place. Table B.26 is worthy of study.

5.3 Random Graphs, unknown $\chi(G)$

The heuristics were run against larger, random graphs where calculation of the actual $\chi(G)$ is intractable. With these runs, heuristics were compared against the best performing heuristic for each graph. Accuracy and performance of estimates are now against the best performers, and not against the actual $\chi(G)$.

Graph size for these runs was restricted by the memory required to hold a n X n edge table. A graph size of $n = 300$ was chosen through experimentation as a reasonable limit for these larger graphs.

As with random graphs on 23 nodes, experiments were run with a varying number of edge densities, but edge densities of 55% and 85% were arbitrarily chosen to be reported in this thesis.

As stated, the actual $\chi(G)$ cannot be calculated for these large graphs. Clearly, a simple upper bound on the actual $\chi(G)$ is n, but a more refined estimate is needed.

Bollobás theorized that the greedy algorithm uses fewer than $\{1 + 5(\log\log n)/\log n\}n/\log_d n$ colors, where d is $1/1-p$ and p is the *edge density* [3]. This formula provides a practical upper bound for all of the sequential algorithms.

5.3.1 300 nodes, 55% dense

With these larger graphs the annealing algorithms are now the best performers. LFAI tops the list followed by SLFAI as seen in table B.29.

Color distribution goes beyond what *chicheck*'s build in reporting mechanism can handle, but results up to 50 colors are shown in table B.30.

The larger graphs also show bigger gaps between the best and worst estimates. By table B.31, these values differ by as much as 11 colors.

5.3.2 300 nodes, 85% dense

The dense, large graphs in this experiment exaggerate the ability of the annealing algorithms. LFAI tops the chart with the best colorings 60% of the time while SLFAI comes in second with 51%. The next closest algorithm, LFDI is only at 8% as seen in table B.32.

These denser graphs also exaggerate the gap between the best and worst estimates. Coloring estimates differ by as much as 19 colors, as seen in table B.34.

5.4 Best performing algorithms

For small, connected graphs, one of the double-interchange algorithms always provided the best colorings. At the larger end of small graphs where $10 \leq n \leq 12$, largest first interchange algorithms provided the top two colorings.

For families of algorithms such as LF, LFI, LFDI and LFDAI, the interchange versions always performed better than the base algorithm. Also, the double-interchange version consistently beat out the single interchange, and the annealing version varied in its ranking.

Random graphs saw the double interchange algorithms at the top of the list for small ($n = 23$) graphs, but the annealing algorithms grew in rank as the graphs became denser. The largest graphs run with $n = 300$ found the annealing algorithms at the top of the list. The denser graphs in this category further exaggerated the annealing ability with a sharp decline in performance of the non-annealing algorithms.

5.5 Worst performing algorithms

S, IS and GIS algorithms always were at the bottom of the lists. This behavior matches the theoretical estimates of their performance [11].

Chapter 6

Future Work

This chapter discusses future work that could be performed to extend this thesis.

6.1 Unoptimal characterization

Additional record keeping could be performed to note specific graphs that none of the heuristics colors optimally. These graphs could then be analyzed in an attempt to discover if any common properties for these types of graphs exist. Classification of these graphs could spawn additional heuristics to handle the specific characteristics these graphs have.

6.2 Multiple density analysis

For brevity, this thesis only ran experiments of 55% and 85% edge density. Investigation of heuristic behavior for eight or more intervals of edge density would provide a more detailed view of the relationship between edge density and heuristic performance.

6.3 Larger graphs

Experiments were restricted to graphs of size $n = 300$. Computer resources could be obtained with memory sizes capable of handling larger graphs.

Source code could be modified to only use one half of the n X n matrix, since no directed graphs have been evaluated.

Code could also pack graphs into bit fields instead of using a byte per edge to handle larger graphs.

In addition, much larger graphs of low edge density could also be examined using adjacency lists instead of adjacency matrices. The lists would decrease memory requirements but increase cpu times.

6.4 More connected graph analysis

Calculations for all connected graphs where $n = 12$ were performed in approximately forty days using approximately 150 computers in the computer science labs at Rochester Institute of Technology. These machines were only used to work on *chicheck* when they were not in use by students.

It seems reasonable that all connected graphs on $n = 13$ nodes could be accomplished using a similar setup in a lifetime using larger numbers of computers. It is likely that all connected graphs on $n = 14$ could not be accomplished in a reasonable amount of time.

6.5 Heuristic Refinements

Both the double and annealing interchanges were created by examining smallest hard to color graphs. Further refinements to the algorithms could be made by examining other properties of the heuristics through theory, observation and experimentation.

6.6 Enhancing other heuristics

The double and annealing interchange procedures were added to existing coloring heuristics to create the new heuristics. The only heuristics examined were those whose smallest hard to color graphs had been provided. Other graph coloring heuristics exist. The double and annealing interchange procedure may be able to enhance these algorithms as well.

6.7 Reporting Mechanism

The reporting mechanism in *chicheck* did not handle graphs sets that did not have $\chi(G)$ calculated, and for sets that were very large. A more robust and dynamic mechanism would provide more detailed data for analysis.

Chapter 7

Conclusions

The two new procedures introduced by this thesis improve graph coloring heuristic performance for two categories of graphs.

The double-interchange procedure always provides the best colorings when used with small, connected graphs.

The annealing-interchange procedure provides superior coloring ability for larger, random graphs. Graphs of higher densities are particularly improved in their colorings.

These heuristics were created by examining smallest hard to color graphs for known graph coloring heuristics.

Appendix A

Acronyms and Abbreviations

G	Graph
GIS	Greedy Independent Set
I	Interchange
IS	Independent Set
LF	Largest-First (Sequential)
LFAI	Largest-First (Sequential) Double Interchange
LFI	Largest-First (Sequential) Interchange
MIS	Maximal Independent Set
n	number of nodes in given graph
S	(Random) Sequential
SI	(Random) Sequential Interchange
SL	Smallest-Last (Sequential)
LFDI	Largest-First (Sequential) Double Interchange
SLF	Saturation Largest-First (Sequential)
SLFAI	Saturation Largest-First (Sequential) Annealing Interchange
SLFDI	Saturation Largest-First (Sequential) Annealing Interchange
SLFI	Saturation Largest-First (Sequential) Interchange
SLI	Smallest-Last (Sequential) Interchange
US	Unordered Sequential

Appendix B

Data Tables

B.1 All connected graphs 5 nodes

90.48%	IS	19/21
90.48%	GIS	19/21
95.24%	SI	20/21
100.00%	S	21/21
100.00%	LF	21/21
100.00%	SL	21/21
100.00%	US	21/21
100.00%	Chi	21/21
100.00%	LFI	21/21
100.00%	SLF	21/21
100.00%	SLI	21/21
100.00%	LFAI	21/21
100.00%	LFDI	21/21
100.00%	SLFI	21/21
100.00%	SLFAI	21/21
100.00%	SLFDI	21/21

Table B.1: Performance distribution all connected graphs 5 nodes

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	5	4	4	5	5	5
03	12	12	12	12	12	12
04	3	4	4	3	3	3
05	1	1	1	1	1	1
06	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	5	5	4	5	5	5
03	12	12	13	12	12	12
04	3	3	3	3	3	3
05	1	1	1	1	1	1
06	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	5	5	5	5
03	12	12	12	12
04	3	3	3	3
05	1	1	1	1
06	0	0	0	0

Table B.2: Color distribution all connected graphs 5 nodes

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	2	2	0	0	0
02	0	0	0	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	0	0	1	0	0	0
02	0	0	0	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	0	0	0	0
02	0	0	0	0
03	0	0	0	0
04	0	0	0	0
05	0	0	0	0

Table B.3: Accuracy distribution all connected graphs 5 nodes

B.2 All connected graphs 6 nodes

91.07%	GIS	102/112
91.96%	S	103/112
92.86%	IS	104/112
96.43%	SI	108/112
96.43%	US	108/112
100.00%	LF	112/112
100.00%	SL	112/112
100.00%	Chi	112/112
100.00%	LFI	112/112
100.00%	SLF	112/112
100.00%	SLI	112/112
100.00%	LFAI	112/112
100.00%	LFDI	112/112
100.00%	SLFI	112/112
100.00%	SLFAI	112/112
100.00%	SLFDI	112/112

Table B.4: Performance distribution all connected graphs 6 nodes

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	17	11	14	17	17	17
03	64	67	63	64	64	64
04	26	28	29	26	26	26
05	4	5	5	4	4	4
06	1	1	1	1	1	1
07	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	17	13	16	17	17	17
03	64	64	62	64	64	64
04	26	29	29	26	26	26
05	4	5	4	4	4	4
06	1	1	1	1	1	1
07	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	17	17	17	16
03	64	64	64	62
04	26	26	26	29
05	4	4	4	4
06	1	1	1	1
07	0	0	0	0

Table B.5: Color distribution all connected graphs 6 nodes

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	10	8	0	0	0
02	0	0	0	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	0	9	4	0	0	0
02	0	0	0	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	0	0	0	4
02	0	0	0	0
03	0	0	0	0
04	0	0	0	0
05	0	0	0	0
06	0	0	0	0

Table B.6: Accuracy distribution all connected graphs 6 nodes

B.3 All connected graphs 7 nodes

82.88%	S	707/853
83.70%	IS	714/853
85.58%	GIS	730/853
92.26%	US	787/853
96.37%	SI	822/853
98.59%	SLF	841/853
98.59%	SLFAI	841/853
98.83%	LF	843/853
98.83%	LFAI	843/853
99.53%	SL	849/853
99.65%	SLI	850/853
100.00%	Chi	853/853
100.00%	LFI	853/853
100.00%	LFDI	853/853
100.00%	SLFI	853/853
100.00%	SLFDI	853/853

Table B.7: Performance distribution all connected graphs 7 nodes

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	44	31	37	42	42	44
03	475	399	369	469	469	475
04	282	352	375	290	290	282
05	46	63	66	46	46	46
06	5	7	5	5	5	5
07	1	1	1	1	1	1
08	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	44	26	39	44	44	44
03	475	388	459	471	463	463
04	282	365	298	286	294	294
05	46	67	51	46	46	46
06	5	6	5	5	5	5
07	1	1	1	1	1	1
08	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	44	44	44	40
03	475	475	472	425
04	282	282	285	328
05	46	46	46	54
06	5	5	5	5
07	1	1	1	1
08	0	0	0	0

Table B.8: Color distribution all connected graphs 7 nodes

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	123	138	10	10	0
02	0	0	1	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	0	146	31	4	12	12
02	0	0	0	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	0	0	3	66
02	0	0	0	0
03	0	0	0	0
04	0	0	0	0
05	0	0	0	0
06	0	0	0	0
07	0	0	0	0

Table B.9: Accuracy distribution all connected graphs 7 nodes

B.4 All connected graphs 8 nodes

77.53%	S	8619/11117
77.75%	IS	8643/11117
77.96%	GIS	8667/11117
86.01%	US	9562/11117
91.32%	SI	10152/11117
96.59%	SL	10738/11117
96.97%	LF	10780/11117
97.08%	SLF	10792/11117
97.18%	LFAI	10803/11117
97.22%	SLFAI	10808/11117
98.49%	SLI	10949/11117
99.40%	LFI	11050/11117
99.41%	LFDI	11051/11117
99.47%	SLFI	11058/11117
99.47%	SLFDI	11058/11117
100.00%	Chi	11117/11117

Table B.10: Performance distribution all connected graphs 8 nodes

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	182	113	117	170	170	181
03	5036	3460	3437	4767	4786	4973
04	5009	5932	5923	5246	5231	5072
05	809	1480	1517	853	849	810
06	74	125	116	74	74	74
07	6	6	6	6	6	6
08	1	1	1	1	1	1
09	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	181	112	156	178	182	182
03	4972	3427	4367	4711	4772	4781
04	5073	5961	5469	5292	5212	5210
05	810	1486	1034	855	870	863
06	74	124	84	74	74	74
07	6	6	6	6	6	6
08	1	1	1	1	1	1
09	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	182	182	181	158
03	4981	4981	4888	3943
04	5060	5060	5140	5722
05	813	813	827	1194
06	74	74	74	93
07	6	6	6	6
08	1	1	1	1
09	0	0	0	0

Table B.11: Color distribution all connected graphs 8 nodes

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	2413	2427	337	314	66
02	0	37	47	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	67	2470	964	379	325	309
02	0	28	1	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	59	59	168	1546
02	0	0	0	9
03	0	0	0	0
04	0	0	0	0
05	0	0	0	0
06	0	0	0	0
07	0	0	0	0
08	0	0	0	0

Table B.12: Accuracy distribution all connected graphs 8 nodes

B.5 All connected graphs 9 nodes

70.78%	S	184790/261080
70.80%	GIS	184844/261080
70.83%	IS	184914/261080
78.61%	US	205241/261080
87.40%	SI	228178/261080
92.07%	SL	240378/261080
94.11%	SLF	245700/261080
94.32%	LF	246254/261080
94.61%	SLFAI	247020/261080
94.76%	LFAI	247399/261080
96.67%	SLI	252386/261080
98.53%	LFI	257240/261080
98.55%	SLFI	257301/261080
98.56%	LFDI	257319/261080
98.59%	SLFDI	257387/261080
100.00%	Chi	261080/261080

Table B.13: Performance distribution all connected graphs 9 nodes

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	730	399	407	675	675	722
03	80947	44011	43770	70764	71392	77940
04	149551	149559	150000	155351	155225	151830
05	27794	61636	61583	32132	31648	28528
06	1940	5271	5122	2040	2022	1942
07	110	194	190	110	110	110
08	7	9	7	7	7	7
09	1	1	1	1	1	1
10	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	722	392	577	678	726	726
03	77885	43977	62529	66580	71156	71698
04	151861	149634	154737	157829	153926	154106
05	28552	61558	40307	33830	33048	32384
06	1942	5326	2792	2045	2106	2048
07	110	184	130	110	110	110
08	7	8	7	7	7	7
09	1	1	1	1	1	1
10	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	729	729	723	620
03	78380	78310	74739	52246
04	151002	151056	153344	152526
05	28904	28920	30165	51821
06	1947	1947	1991	3714
07	110	110	110	145
08	7	7	7	7
09	1	1	1	1
10	0	0	0	0

Table B.14: Color distribution all connected graphs 9 nodes

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	74110	74117	14821	13679	3761
02	0	2126	2048	5	2	0
03	0	0	1	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	3840	74175	32803	20687	15375	14057
02	0	2112	99	15	5	3
03	0	3	0	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	3693	3779	8693	55077
02	0	0	1	762
03	0	0	0	0
04	0	0	0	0
05	0	0	0	0
06	0	0	0	0
07	0	0	0	0
08	0	0	0	0
09	0	0	0	0

Table B.15: Accuracy distribution all connected graphs 9 nodes

B.6 All connected graphs 10 nodes

63.82%	IS	7477782/11716571
63.85%	S	7480690/11716571
63.86%	GIS	7481867/11716571
70.37%	US	8245232/11716571
83.07%	SI	9733029/11716571
86.92%	SL	10184495/11716571
90.56%	SLF	10610661/11716571
91.12%	LF	10676094/11716571
91.52%	SLFAI	10722892/11716571
91.94%	LFAI	10771666/11716571
94.28%	SLI	11046611/11716571
97.19%	SLFI	11387228/11716571
97.26%	SLFDI	11395468/11716571
97.36%	LFI	11407705/11716571
97.44%	LFDI	11416361/11716571
100.00%	Chi	11716571/11716571

Table B.16: Performance distribution all connected graphs 10 nodes

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	4032	2055	2035	3659	3662	3959
03	2010328	840481	838714	1583461	1607083	1860320
04	7694428	6055779	6055398	7529273	7574022	7696980
05	1890221	4280712	4282585	2460539	2396284	2035167
06	113272	520769	521109	135169	131084	115853
07	4125	16448	16410	4305	4271	4127
08	156	316	308	156	156	156
09	8	10	11	8	8	8
10	1	1	1	1	1	1
11	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	3959	2072	3204	3412	3988	3988
03	1857469	841072	1342155	1380411	1611693	1634727
04	7694041	6053813	7155501	7452517	7417514	7475916
05	2040919	4281466	2982218	2729973	2532790	2459718
06	115891	521676	226482	145807	145901	137708
07	4127	16147	6810	4286	4520	4349
08	156	314	191	156	156	156
09	8	10	9	8	8	8
10	1	1	1	1	1	1
11	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	4017	4017	3932	3284
03	1878201	1874536	1721186	1057150
04	7643514	7642633	7616038	6335515
05	2067362	2071863	2244697	3924296
06	119177	119222	126353	385487
07	4135	4135	4200	10614
08	156	156	156	216
09	8	8	8	8
10	1	1	1	1
11	0	0	0	0

Table B.17: Color distribution all connected graphs 10 nodes

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	4052849	4056776	1038689	943700	300152
02	0	181509	181691	1788	1205	58
03	0	346	322	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	308785	4054165	1970639	1527690	1104085	992628
02	81	181376	12902	4386	1825	1051
03	0	340	1	0	0	0
04	0	0	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	321068	329292	669715	3389845
02	35	51	245	81441
03	0	0	0	53
04	0	0	0	0
05	0	0	0	0
06	0	0	0	0
07	0	0	0	0
08	0	0	0	0
09	0	0	0	0
10	0	0	0	0

Table B.18: Accuracy distribution all connected graphs 10 nodes

B.7 All connected graphs 11 nodes

56.55%	S	569259363/1006700565
56.55%	IS	569268358/1006700565
56.55%	GIS	569256501/1006700565
61.93%	US	623431101/1006700565
77.93%	SI	784477199/1006700565
80.54%	SL	810784255/1006700565
85.90%	SLF	864789952/1006700565
86.85%	LF	874328133/1006700565
87.47%	SLFAI	880558679/1006700565
88.16%	LFAI	887477223/1006700565
90.88%	SLI	914915215/1006700565
95.14%	SLFI	957735480/1006700565
95.28%	SLFDI	959204347/1006700565
95.58%	LFI	962206429/1006700565
95.74%	LFDI	963765362/1006700565
100.00%	Chi	1006700565/1006700565

Table B.19: Performance distribution all connected graphs 11 nodes

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	25598	12392	12180	23080	23111	25118
03	76115143	24440097	24442880	53334430	54461213	67048355
04	667036310	393034450	393041840	587786094	597228599	643543975
05	248580644	490336878	490338371	342669750	333708993	279824292
06	14545025	95005297	94993813	22422377	20829973	15853884
07	389583	3824795	3825079	456279	440173	396677
08	8040	46226	45955	8333	8281	8042
09	212	419	436	212	212	212
10	9	10	10	9	9	9
11	1	1	1	1	1	1
12	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	25118	12285	19641	19483	25259	25259
03	66897935	24442950	43497190	42849898	55562724	56758741
04	642331176	393039553	533816471	550073841	576864280	587777832
05	281137210	490330089	388846406	385172144	348118324	338737559
06	15904112	95000906	39439003	28085416	25609676	22925637
07	396750	3828311	1066905	491298	511192	466835
08	8042	46017	14687	8263	8888	8480
09	212	443	252	212	212	212
10	9	10	9	9	9	9
11	1	1	1	1	1	1
12	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	25477	25477	23872	20643
03	68063123	67868578	59357374	32686205
04	638100657	637068191	613705661	423680153
05	283116420	284293028	313687899	469507439
06	16978226	17028430	19490094	78163017
07	408388	408587	427302	2614357
08	8052	8052	8141	28436
09	212	212	212	305
10	9	9	9	9
11	1	1	1	1
12	0	0	0	0

Table B.20: Color distribution all connected graphs 11 nodes

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	410160179	410160953	131914056	118936805	42920002
02	0	27174182	27161039	458365	286533	15201
03	0	109698	110215	11	4	0
04	0	5	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	44473855	410163861	219704928	194576929	141232947	125775044
02	20281	27166610	2517903	1339278	677647	366835
03	0	110726	535	103	19	7
04	0	5	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	47481533	48947109	91698530	368206604
02	14685	17976	86815	15037186
03	0	0	5	25674
04	0	0	0	0
05	0	0	0	0
06	0	0	0	0
07	0	0	0	0
08	0	0	0	0
09	0	0	0	0
10	0	0	0	0
11	0	0	0	0

Table B.21: Accuracy distribution all connected graphs 11 nodes

B.8 All connected graphs 12 nodes

49.47	S	81155199554/164059132502
49.47	IS	81156762921/164059132502
49.47	GIS	81156310115/164059132502
53.87	US	88374597327/164059132502
72.17	SI	118399146620/164059132502
73.50	SL	120583780463/164059132502
80.50	SLF	132065375775/164059132502
81.71	LF	134052033113/164059132502
82.72	SLFAI	135703007967/164059132502
83.57	LFAI	137108580065/164059132502
86.59	SLI	142054125807/164059132502
92.42	SLFI	151625382032/164059132502
92.67	SLFDI	152036809469/164059132502
93.03	LFI	152616593499/164059132502
93.30	LFDI	153060930692/164059132502
100.00	Chi	164059132502/164059132502

Table B.22: Performance distribution all connected graphs 12 nodes

A detailed table of colorings is not provided for all connected graphs on 12 nodes. This special case required distributed calculations to be made on ten thousand subsets of all connected graphs of 12 nodes. Combination of the results for this experiment was kept simple by only providing a brief report.

B.9 Random graphs on 23 nodes, 55% dense

6.30%	IS 63/1000
6.50%	S 65/1000
6.80%	US 68/1000
7.00%	GIS 70/1000
16.60%	SL 166/1000
19.30%	SI 193/1000
23.80%	SLF 238/1000
29.50%	LF 295/1000
31.60%	SLFAI 316/1000
36.20%	SLI 362/1000
37.60%	LFAI 376/1000
49.80%	SLFI 498/1000
51.10%	LFI 511/1000
52.30%	SLFDI 523/1000
53.10%	LFDI 531/1000
100.00%	Chi 1000/1000

Table B.23: Performance distribution random graphs on 23 nodes, 55 dense

colors	Chi	GIS	IS	LF	LFAI	LFDI
02	0	0	0	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	6	0	0	0	0	2
06	361	8	11	53	68	112
07	584	185	175	439	508	616
08	49	483	478	452	394	262
09	0	271	292	55	29	8
10	0	51	42	1	1	0
11	0	2	2	0	0	0
12	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
02	0	0	0	0	0	0
03	0	0	0	0	0	0
04	0	0	0	0	0	0
05	1	0	0	1	0	0
06	108	7	29	18	28	42
07	602	155	370	315	410	494
08	278	504	498	508	456	405
09	11	273	99	147	105	59
10	0	59	4	11	1	0
11	0	2	0	0	0	0
12	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
02	0	0	0	0
03	0	0	0	0
04	0	0	0	0
05	1	1	1	0
06	105	96	72	9
07	626	615	502	167
08	254	273	370	480
09	14	15	55	309
10	0	0	0	30
11	0	0	0	5
12	0	0	0	0

Table B.24: Color distribution random graphs on 23 nodes, 55 dense.

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	443	456	580	539	452
02	0	407	392	119	83	17
03	0	75	87	6	2	0
04	0	5	2	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	464	413	619	552	569	568
02	25	431	180	259	183	111
03	0	87	8	23	10	5
04	0	4	0	0	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	455	476	549	427
02	22	25	86	420
03	0	1	3	84
04	0	0	0	1
05	0	0	0	0
06	0	0	0	0
07	0	0	0	0
08	0	0	0	0
09	0	0	0	0
10	0	0	0	0
11	0	0	0	0

Table B.25: Accuracy distribution random graphs on 23 nodes, 55 dense

B.10 Random graphs on 23 nodes, 85% dense

12.40%	US 124/1000
14.40%	IS 144/1000
15.40%	GIS 154/1000
15.80%	S 158/1000
31.20%	SI 312/1000
33.20%	SL 332/1000
43.70%	SLF 437/1000
45.30%	SLI 453/1000
53.20%	SLFAI 532/1000
63.10%	LF 631/1000
65.80%	SLFI 658/1000
69.30%	SLFDI 693/1000
70.00%	LFAI 700/1000
79.30%	LFI 793/1000
81.80%	LFDI 818/1000
100.00%	Chi 1000/1000

Table B.26: Performance distribution random graphs on 23 nodes, 85 dense

colors	Chi	GIS	IS	LF	LFAI	LFDI
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	2	0	0	0	0	1
10	97	9	9	39	47	56
11	383	71	62	249	278	331
12	362	249	275	444	435	420
13	128	379	360	227	201	157
14	25	223	223	34	33	29
15	3	63	61	7	6	6
16	0	5	10	0	0	0
17	0	1	0	0	0	0
18	0	0	0	0	0	0

colors	LFI	S	SI	SL	SLF	SLFAI
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	1	0	0	0	1	1
10	52	4	15	18	28	33
11	322	64	155	144	175	219
12	426	278	350	361	406	415
13	163	361	313	341	305	264
14	30	228	136	117	72	57
15	6	57	27	19	13	11
16	0	8	4	0	0	0
17	0	0	0	0	0	0
18	0	0	0	0	0	0

colors	SLFDI	SLFI	SLI	US
07	0	0	0	0
08	0	0	0	0
09	1	1	2	0
10	49	45	24	6
11	281	269	207	70
12	422	423	382	256
13	203	213	286	382
14	34	37	88	203
15	10	12	11	74
16	0	0	0	9
17	0	0	0	0
18	0	0	0	0

Table B.27: Color distribution random graphs on 23 nodes, 85 dense.

off	Chi	GIS	IS	LF	LFAI	LFDI
01	0	414	456	353	291	181
02	0	369	318	16	9	1
03	0	58	76	0	0	0
04	0	5	5	0	0	0
05	0	0	1	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0

off	LFI	S	SI	SL	SLF	SLFAI
01	206	433	495	504	479	418
02	1	322	181	150	81	49
03	0	81	12	12	3	1
04	0	6	0	2	0	0
05	0	0	0	0	0	0
06	0	0	0	0	0	0
07	0	0	0	0	0	0
08	0	0	0	0	0	0
09	0	0	0	0	0	0
10	0	0	0	0	0	0

off	SLFDI	SLFI	SLI	US
01	299	327	469	484
02	8	15	74	311
03	0	0	3	70
04	0	0	1	11
05	0	0	0	0
06	0	0	0	0
07	0	0	0	0
08	0	0	0	0
09	0	0	0	0
10	0	0	0	0

Table B.28: Accuracy distribution random graphs on 23 nodes, 85 dense

B.11 Random graphs on 300 nodes, 55% dense

0.00%	S 0/1000
0.00%	IS 0/1000
0.00%	SL 0/1000
0.00%	US 0/1000
0.00%	GIS 0/1000
0.10%	SI 1/1000
1.00%	SLF 10/1000
1.10%	SLI 11/1000
1.40%	LF 14/1000
15.40%	SLFI 154/1000
18.40%	LFI 184/1000
29.50%	SLFDI 295/1000
33.80%	LFDI 338/1000
36.90%	SLFAI 369/1000
46.60%	LFAI 466/1000

Table B.29: Performance distribution random graphs on 300 nodes, 55 dense

colors	GIS	IS	LF	LFAI	LFDI	LFI
44	0	0	0	0	0	0
45	0	0	0	0	0	0
46	0	0	0	3	0	1
47	0	0	0	50	11	6
48	0	0	4	222	163	69
49	0	0	34	411	431	285
50	1	1	190	239	318	402
51	20	19	324	71	75	199
52	97	100	306	4	2	38
53	259	266	128	0	0	0
54	330	335	11	0	0	0
55	195	200	2	0	0	0
56	75	58	1	0	0	0
57	21	21	0	0	0	0
58	2	0	0	0	0	0
59	0	0	0	0	0	0

colors	S	SI	SL	SLF	SLFAI	SLFDI
44	0	0	0	0	0	0
45	0	0	0	0	0	0
46	0	0	0	0	0	0
47	0	0	0	0	35	14
48	0	0	0	4	172	130
49	0	6	0	16	403	403
50	1	59	6	102	260	350
51	16	233	75	290	105	94
52	108	386	230	319	24	8
53	248	246	332	200	1	1
54	337	62	249	62	0	0
55	211	7	87	6	0	0
56	62	1	16	1	0	0
57	14	0	5	0	0	0
58	3	0	0	0	0	0
59	0	0	0	0	0	0

colors	SLFI	SLI	US
44	0	0	0
45	0	0	0
46	0	0	0
47	4	0	0
48	63	3	0
49	253	30	0
50	401	206	3
51	230	370	19
52	49	293	84
53	0	88	251
54	0	10	338
55	0	0	219
56	0	0	69
57	0	0	15
58	0	0	2
59	0	0	0

Table B.30: Color distribution random graphs on 300 nodes, 55 dense.

off	GIS	IS	LF	LFAI	LFDI	LFI
01	0	0	87	324	368	303
02	6	7	220	160	228	320
03	47	42	304	41	57	149
04	144	166	259	8	9	38
05	270	269	94	1	0	6
06	284	261	20	0	0	0
07	155	171	1	0	0	0
08	74	74	1	0	0	0
09	19	10	0	0	0	0
10	1	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0

off	S	SI	SL	SLF	SLFAI	SLFDI
01	1	26	3	39	323	365
02	5	119	24	161	226	252
03	49	259	126	277	60	73
04	144	336	258	297	19	13
05	260	186	299	155	3	2
06	312	59	180	53	0	0
07	148	13	84	7	0	0
08	59	1	23	1	0	0
09	19	0	3	0	0	0
10	3	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0

off	SLFI	SLI	US
01	295	76	2
02	328	258	7
03	172	347	45
04	44	218	123
05	6	79	279
06	1	11	285
07	0	0	176
08	0	0	68
09	0	0	11
10	0	0	3
11	0	0	1
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0

Table B.31: Accuracy distribution random graphs on 300 nodes, 55 dense

B.12 Random graphs on 300 nodes, 85% dense

0.00%	S 0/1000
0.00%	IS 0/1000
0.00%	SI 0/1000
0.00%	US 0/1000
0.00%	GIS 0/1000
0.00%	SLF 0/1000
0.00%	SLI 0/1000
0.10%	SL 1/1000
0.50%	LF 5/1000
2.50%	SLFI 25/1000
3.50%	LFI 35/1000
5.80%	SLFDI 58/1000
8.70%	LFDI 87/1000
51.00%	SLFAI 510/1000
60.40%	LFAI 604/1000

Table B.32: Performance distribution random graphs on 300 nodes, 85 dense

colors	GIS	IS	LF	LFAI	LFDI	LFI
82	0	0	0	0	0	0
83	0	0	0	0	0	0
84	0	0	0	0	0	0
85	0	0	0	0	0	0
86	0	0	0	1	0	0
87	0	0	0	12	0	0
88	0	0	0	30	1	0
89	0	0	0	95	3	2
90	0	0	1	196	26	4
91	0	0	1	269	67	40
92	0	0	11	212	175	97
93	0	0	39	118	259	186
94	2	0	93	55	245	254
95	7	1	178	8	152	220
96	19	6	209	4	59	123
97	30	46	204	0	11	50
98	86	101	143	0	1	19
99	149	168	76	0	1	4
100	207	188	32	0	0	1
101	207	187	11	0	0	0
102	141	146	2	0	0	0
103	91	91	0	0	0	0
104	42	38	0	0	0	0
105	11	20	0	0	0	0
106	7	6	0	0	0	0
107	1	1	0	0	0	0
108	0	1	0	0	0	0
109	0	0	0	0	0	0

colors	S	SI	SL	SLF	SLFAI	SLFDI
82	0	0	0	0	0	0
83	0	0	0	0	0	0
84	0	0	0	0	0	0
85	0	0	0	0	0	0
86	0	0	0	0	0	0
87	0	0	0	0	4	0
88	0	0	0	0	23	0
89	0	0	0	0	72	0
90	0	0	0	0	201	16
91	0	1	0	2	234	51
92	0	1	1	2	229	152
93	0	5	0	15	141	230
94	1	23	1	51	68	254
95	1	76	12	81	25	192
96	17	141	38	183	3	81
97	38	191	88	204	0	21
98	100	218	147	196	0	3
99	136	183	198	152	0	0
100	196	99	218	73	0	0
101	214	44	156	33	0	0
102	130	15	88	5	0	0
103	98	3	40	3	0	0
104	47	0	9	0	0	0
105	17	0	3	0	0	0
106	5	0	1	0	0	0
107	0	0	0	0	0	0
108	0	0	0	0	0	0
109	0	0	0	0	0	0

colors	SLFI	SLI	US
82	0	0	0
83	0	0	0
84	0	0	0
85	0	0	0
86	0	0	0
87	0	0	0
88	0	0	0
89	0	0	0
90	5	0	0
91	28	1	0
92	72	6	0
93	142	27	0
94	220	55	2
95	240	151	3
96	163	191	5
97	95	228	42
98	32	182	111
99	3	101	153
100	0	43	188
101	0	12	187
102	0	2	150
103	0	1	90
104	0	0	42
105	0	0	20
106	0	0	5
107	0	0	1
108	0	0	1
109	0	0	0

Table B.33: Color distribution random graphs on 300 nodes, 85 dense.

off	GIS	IS	LF	LFAI	LFDI	LFI
01	0	0	7	171	109	57
02	1	1	24	130	213	145
03	2	1	86	67	225	211
04	5	1	126	20	178	217
05	9	12	165	7	101	164
06	24	34	179	1	56	95
07	67	64	159	0	21	49
08	112	114	135	0	6	18
09	173	170	71	0	4	7
10	181	173	30	0	0	1
12	126	109	3	0	0	0
13	69	61	0	0	0	0
14	29	51	0	0	0	0
15	21	14	0	0	0	0
16	7	8	0	0	0	0
17	2	4	0	0	0	0
18	0	0	0	0	0	0
19	0	1	0	0	0	0
20	0	0	0	0	0	0
21	0	0	0	0	0	0
22	0	0	0	0	0	0
23	0	0	0	0	0	0
24	0	0	0	0	0	0
25	0	0	0	0	0	0

off	S	SI	SL	SLF	SLFAI	SLFDI
01	0	0	0	5	186	112
02	0	3	0	12	164	184
03	1	21	2	37	80	191
04	2	57	10	75	41	203
05	14	113	37	110	16	133
06	26	152	49	163	2	79
07	72	185	118	200	1	31
08	123	183	160	179	0	7
09	141	132	193	117	0	2
10	186	81	179	59	0	0
12	118	18	77	11	0	0
13	84	6	35	2	0	0
14	42	2	14	0	0	0
15	21	1	9	0	0	0
16	1	0	0	0	0	0
17	2	0	0	0	0	0
18	1	0	0	0	0	0
19	0	0	0	0	0	0
20	0	0	0	0	0	0
21	0	0	0	0	0	0
22	0	0	0	0	0	0
23	0	0	0	0	0	0
24	0	0	0	0	0	0
25	0	0	0	0	0	0

off	SLFI	SLI	US
01	45	5	0
02	113	27	0
03	177	53	2
04	195	96	2
05	196	151	12
06	138	189	25
07	68	184	81
08	33	137	126
09	9	92	138
10	1	45	167
12	0	4	133
13	0	2	70
14	0	0	38
15	0	0	15
16	0	0	9
17	0	0	2
18	0	0	1
19	0	0	0
20	0	0	0
21	0	0	0
22	0	0	0
23	0	0	0
24	0	0	0
25	0	0	0

Table B.34: Accuracy distribution random graphs on 300 nodes, 85 dense

Appendix C

Source Code

C.1 Discussion

This appendix contains listings of the new source code written for this thesis. Specifically, this is the code that when compiled produces the *chicheck* executable.

This appendix does not contain the source code of the tools used in conjunction with *chicheck* to complete the experiments. Specifically, there is no code provided for the *makeg*[15] or *autoson*[16] tools.

Brief descriptions of the source code files are provided in table C.1. Detailed explanation of the behavior of these files can be extrapolated by examining the actual source code and comments contained within.

File	Description
Makefile	Makefile to build chicheck binary from dependencies
algotypes.h	Macro definitions for heuristic algorithms
chicheck.h	Definitions of external variables to describe graph
colors.h	Definition of external array to describe coloring
enabled.h	Function prototypes for algorithm enabler disabler
graphmacro.h	Macro definitions of algorithmic constants
makerg.h	Function prototypes for random graph generation
order.h	Defintions of external variables to describe order of nodes
algo_ai.c	Annealing interchange algorithms LFAI and SLFAI
algo_chi.c	Brute-force coloring algorithm, Chi
algo_di.c	Double interchange algorithms LFDI and SLFDI
algo_gis.c	Greedy independent set algorithm, GIS
algo_is.c	Random independent set algorith, IS
algo_lf.c	Largest-First sequential algorithms, LF and LFI
algo_s.c	Random sequential algorithms, S and SI
algo_sl.c	Smallest-Last sequential algorithms, SL and SLI
algo_slf.c	Saturation Largest-first algorithms, SLF and SLFI
algo_us.c	Unordered sequential algorithm, US
algotypes.c	Utility functions to convert algorithm types with strings
chicheck.c	Main program collects parameters and runs algorithms
colors.c	Utility functions for graph colors
enabled.c	Utility functions enable and disable running of algorithms
graphlib.c	Functions for calculating properties of graphs (from Dr. Radziszowski)
graphlib2.c	Functions for calculating properties of graphs (from the author)
interchange.c	Interchange portion of algorithm for LFI, SI, SLI and SLFI
makerg.c	Functions to generate a random graph
order.c	Functions to track ordering while coloring nodes
stats.c	Functions to keep and report statistics on algorithms
verify.c	Utility function to verify valid colorings of graphs are produced

Table C.1: Source Code Descriptions

Bibliography

[1] A. V. Aho et. al., The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, Reading, Massachusetts. 1974.

[2] A. Blum, *New approximation algorithms for graph coloring*, J. ACM 41, 3 (May. 1994), 470 - 516

[3] B. Bollobás, Random Graphs, Academic Press, London. 1985.

[4] D. Brélaz, *New methods to color the vertices of a graph*, Comm. ACM 22 (1979), 251 256.

[5] T. H. Cormen, et. al., Introduction to Algorithms, The MIT Press. 1999.

[6] P. Hansen and J. Kuplinsky, *Slightly hard-to-color graphs*, Congressus Numerantium, 78 (1990), 81-98.

[7] M. M. Halldorsson, *A Still Better Performance Guarantee for Approximate Graph Coloring*, Information Processing Letters 45, (1993), 19-23.

[8] P. Hansen and J. Kuplinsky, *The smallest hard-to-color graph*, Discrete Mathematics, 96 (1991), 199-212.

[9] D. S. Hochbaum, Approximation Algorithms for NP-hard problems PWS Publishing Company, Boston, MA. 1997.

[10] D. S. Johnson, *Worst-Case behavior of graph coloring algorithms*, Proc. 5th Southeastern Conf. on Combinatorics, Graph Theory and Computing, Congr. Numer. 10, (1974) 513-527.

[11] M. Kubale, Introduction to Computational Complexity and Algorithmic Graph Coloring, Gdańskie Towarzystwo Naukowe. 1998.

[12] M. Kubale, *The quest for small benchmarks for the chromatic sum problem*, Archives of Control Sciences, Volume 6, No. 3-4 (1997); 249 - 260.

[13] M. Kubale, *The smallest hard-to-color graphs for the classical, total and strong colorings of vertices*, Control and Cybernetics, Volume 28, No. 2 (1999).

[14] D. W. Matula et. al., *Graph Coloring Algorithms*, Graph Theory and Computing, Academic Press, New York. (1972), 109-122.

[15] B. McKay, *nauty: a set of procedures for determining the automorphism group of a graph, and optionally for canonically labeling it*, http://cs.anu.edu.au/~bdm/nauty/

[16] B. McKay, *autoson: a tool for scheduling independent processes across a network of UNIX workstations*, http://cs.anu.edu.au/~bdm/autoson/

[17] S. P. Radziszowski, Rochester Institute of Technology, Rochester, NY. http://www.cs.rit.edu/~spr

[18] S. R. Vegdahl, *Using node merging to enhance graph coloring*, Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation, 1999, 150 154.

[19] P. J. Welsh and M. B. Powell, *An upper bound for the chromatic number of graph and its applications to timetabling problem*, Comput. J. 10, 85-86, 1967.

[20] R. J. Wilson and J. J. Watkins, Graphs: an Introductory Approach, John Wiley and Sons, Inc. 1990.

Makefile


```
#
#  Makefile for chicheck
#
#  Author:  Andrew Radin
#  DAte:    01/18/00
#

OBJS = chicheck.o algotypes.o graphlib.o graphlib2.o colors.o stats.o \
interchange.o order.o algo_chi.o algo_us.o algo_s.o algo_sl.o algo_lf.o \
algo_slf.o algo_is.o algo_gis.o algo_di.o verify.o enabled.o makerg.o \
algo_ai.o



HEAD = algotypes.h chicheck.h colors.h graphmacro.h order.h thresholds.h

# tracing options
# -DdebugSI
# -DdebugSLF
# -DdebugSL
# -DdebugINT
# -DdebugIS
# -DdebugMR
# -DdebugAI
# -DVERIFY
#TRACE = -DVERIFY -DdebugAI
TRACE =

# debug mode
# ARGS = -g -pg
# LINK = -g -pg

# performance mode ( >3x faster than debug )
ARGS = -O
LINK =

CC = gcc
#CC = cc
LIBS = -lm

all: chicheck

install: all
        rm -f ../bin/chicheck
        cp chicheck ../bin

new: realclean all

chicheck: $(OBJS)
        $(CC)  $(LINK) $(OBJS) $(LIBS) -o chicheck

.c.o: $(HEAD)
        $(CC) -c $(ARGS) $(TRACE) $*.c

clean:
        rm -f $(OBJS) core

realclean: clean
        rm -f chicheck
```

algotypes.h


```
/*
 *    Name:     algotypes.h
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Definitions for algo types
 *
 *
 */

#define UNDEF    -1
#define BEST      0
#define FIRSTA    1

#define CHI       1   /* chromatic number */
#define US        2   /* unordered sequential -- use order from makeg */
#define S         3   /* sequential random ordering */
#define LF        4   /* sequential largest first */
#define SL        5   /* sequential smallest last ordering */
#define SLF       6

#define SI        7   /* same as above but with interchange procedure */
#define LFI       8
#define SLI       9
#define SLFI     10

#define IS       11   /* independet set        */
#define GIS      12   /* greedy independet set */

#define LFDI     13   /* double interchange on lfi  */
#define SLFDI    14   /* double interchange on slfi */

#define LFAI     15   /* annealing interchange on lfi  */
#define SLFAI    16   /* annealing interchange on slfi */

#define LASTA    16

#define STR_CHI       "Chi"
#define STR_US        "US"
#define STR_S         "S"
#define STR_SL        "SL"
#define STR_LF        "LF"
#define STR_SLF       "SLF"
#define STR_SI        "SI"
#define STR_SLI       "SLI"
#define STR_LFI       "LFI"
#define STR_SLFI      "SLFI"
#define STR_IS        "IS"
#define STR_GIS       "GIS"
#define STR_SLFDI     "SLFDI"
#define STR_LFDI      "LFDI"
#define STR_SLFAI     "SLFAI"
#define STR_LFAI      "LFAI"


int algoS2I(char * algo);
char * algoI2S(int algo);
int runalgo(int algo);
```

chicheck.h


```
/*
 *    Name:      chicheck.h
 *    Author:    Andrew Radin
 *    Date:      01/18/00
 *    Purpose:
 *
 *
 */

extern int G[][MAXVERT];
extern int n;
```

colors.h


```
/*
 *    Name:      colors.h
 *    Author:    Andrew Radin
 *    Date:      01/18/00
 *    Purpose:
 *
 *
 */

extern int c[MAXVERT];
```

enabled.h

```
void enabled_init();
int enabled(int algo);
```

graphmacro.h


```
#include <stdio.h>

#define min(x,y) ((x<y) ? (x) : (y))
#define max(x,y) ((x<y) ? (y) : (x))
/* #define SIZE 16 */

/* some strange bug requires SIZE and MAXVERT to be the same */
/* also, get much bigger than 450 and it core dumps.  Hmm... */
/* size is size of max graph to calculate actual chi */
#define SIZE 450
/* maxvert is size of max graph */
#define MAXVERT 450

/* much bigger than 25 and it's never gonna finish for a single graph */
/* in your lifetime                                                   */
#define CHIMAX 23

#define clMAX 4400
```

makerg.h

```
void makerg_init(int d, int n);
int makerg();
```

order.h


```
/*
 *    Name:      order.h
 *    Author:    Andrew Radin
 *    Date:      01/18/00
 *    Purpose:
 *
 *
 */

extern int pointer;
extern int order[MAXVERT+1];
```

algo_ai.c

```
/*
 *    Name:     algo_ai.c
 *    Author:   Andrew Radin
 *    Date:     2/2/00
 *    Purpose:  annealing interchange with SL and L algorithms
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

#define ANNEAL 100000

int saveC[MAXVERT];

int lfai() {

        int chi=1, v;

        pointer=0;
        clear_colors();

        orderLF(G);

        while ((v = nextVertex()) != -1) {
                c[v] = 1;
                while ( conflict(v) ) {

                        if ( c[v] == chi) {
                                if (! (annealinter(v, chi)) ) {
                                        c[v]++;
                                        if (c[v] > chi) chi = c[v];
                                }
                        } else {
                                c[v]++;
                                if (c[v] > chi) chi = c[v];
                        }
                }
        }

        chi = 0;
        for (v = 0 ; v < n ; v++)
                if (c[v] > chi) chi = c[v];
        return chi;
}

int slfai() {

        int chi=1, v;
        int i;

        pointer=0;
        clear_colors();

        for (i = 0 ; i < n ; i++) {
                v = nextlSatV(G);
                c[v] = 1;

                while ( conflict(v) ) {
                        if ( c[v] == chi) {
                                if (! (annealinter(v, chi)) ) {
                                        c[v]++;
```



```
                                        if (c[v] > chi) chi = c[v];
                                }
                        } else {
                                c[v]++;
                                if (c[v] > chi) chi = c[v];
                        }
                }
        }

        chi = 0;
        for (i = 0 ; i < n , i++)
                if (c[i] > chi) chi = c[i];

        return chi;
}

int annealinter(int v, int chi) {

        int more=ANNEAL;
        int neighbor;

        save_colors();

        while (more) {
                if ( (neighbor = random_neighbor(v)) == -1) {
                        restore_colors();
                        return 0;
                }
#ifdef debugAI
        printf("vertex %d, neighbor %d\n", v, neighbor);
#endif
                c[v] = c[neighbor];
                c[neighbor] = 0;
                if (conflict(v)) {
                        c[neighbor] = c[v];
                        /* greedy_color(v); */
                        c[v]=chi;
                } else {
                        greedy_color(neighbor);
#ifdef debugAI
                        if (c[neighbor] < chi) printf("***Got one!\n");
#endif
                        if (c[neighbor] < chi) return 1;
                        v = neighbor;
                }
                more--;
        }
        restore_colors();
        return 0;
}

int random_neighbor(int v) {

        int i;
        int count=0;
        int num, this;

        num =0;
        for (i=0 ; i < n ; i ++) {
                if (G[v][i] && c[i])
                        num++;
        }

        this = rand() % num;

        for (i=0 , i < n , i ++) {
```

```
                if (G[v][i] && c[i]) {
                        count++;
#ifdef debugAI
        printf("Neighbor %d of %d of vertex %d is color %d, (%d)\n"
                        i, num, v, c[i], c[v]);
#endif
                        if (count == this) return i;
                }
        }

        return -1;

}

int greedy_color(int v) {

        c[v] = 1;
        while (conflict(v))
                c[v]++;

}

save_colors() {

        int i;

        for (i = 0 ; i < n ; i ++)
                saveC[i] = c[i];
}

restore_colors() {

        int i;

        for (i = 0 ; i < n ; i ++)
                c[i] = saveC[i];
}
```

```
/*
 *    Name:     algo_chi.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Brute-force coloring algorithm
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"

int chrome() {

        int i, j;

        if (n > CHIMAX) {
                printf("Calculating chi for a graph on %d vertices will take too long.
\n"  n);
                printf("Exiting...\n"  n);
                exit(1);
        }

        for (i=0 ; i < n ; i++) {
                for (j=0 ; j < n ; j++) {
                        if (G[i][j]) return testchi(G, n, 0);
                }
        }

        /* graph with no edges */
        return 1;

}
```

algo_di.c


```
/*
 *    Name:     algo_dilfia.c
 *    Author:   Andrew Radin
 *    Date:     2/2/00
 *    Purpose:  double interchange with SL and L algorithms
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int lfdi() [

        int chi=1, v;

        int oc, i;

        pointer=0;
        clear_colors();

        orderLF(G);

        while ((v = nextVertex()) != -1) {
                c[v] = 1;
                while ( conflict(v) ) {

                        if ( c[v] == chi) {
                                if (! (doubleinterc(v)) ) {
                                        c[v]++;
                                        if (c[v] > chi) chi = c[v];
                                }
                        } else {
                                c[v]++;
                                if (c[v] > chi) chi = c[v];
                        }
                }
        }

#ifdef VERIFY
        oc = 0;
        for (i = 0 ; i < n ; i++)
                if (c[i] > oc) oc = c[i];

        if (oc != chi)
                printf("Calculation error in Chi for lfdi!\n");
#endif

        return chi;
}

int slfdi() {

        int chi=1, v;
        int i;

        int oc;

        pointer=0;
        clear_colors();

        for (i = 0 ; i < n ; i++) {
                v = nextLargeV(G);
                while ( conflict(v) ) {
                        if ( c[v] == chi) {
                                if (! (doubleinterc(v)) ) {
                                        c[v]++;
                                        if (c[v] > chi) chi = c[v];
                                }
                        } else {
                                c[v]++;
                                if (c[v] > chi) chi = c[v];
                        }
                }
        }
#ifdef VERIFY
        oc = 0;
        for (i = 0 ; i < n , i++)
                if (c[i] > oc) oc = c[i];

        if (oc != chi)
                printf("Calculation error in Chi for slfdi!\n");
#endif

        return chi;
}

int doubleinterc(int v) {

        int i;
        int tmp, who;

        /* if we can't interchange this vertex with neigbors */
        if (! interc(v)) {

                /* for each vertex in the graph */
                for (i = 0 , i < n  i++) {

                        /* attempt an interchange with its neighbors */
                        if ( who = interc(i) ) {

                                /* if we can now interchange v then good */
                                if (interc(v)) {
                                        return 1;
                                /* otherwise, undo and try the next one */
                                } else {
                                  tmp = c[i];
                                  c[i] = c[who-1];
                                  c[who-1] = tmp;
                                }
                        }
                }
                return 0;
        }

        return 1;
}
```

algo_gis.c

```
/*
 *    Name:     algo_gis.c
 *    Author:   Andrew Radin
 *    Date:     1/23/00
 *    Purpose:  greedy independent set graph algorithm
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int gis() {

        int chi=1, i, v;

        pointer=0;
        clear_colors();

        orderRan();

#ifdef debugGIS
        printf("Welcome to GIS\n");
        showgraph(G,n);
        fflush(stdout);
#endif
        while (! all_colored() ) {
#ifdef debugGIS
        dump_colors();
        fflush(stdout);
        sleep(1);
#endif
                pointer=0;
                /* for (v = 0 , v < n ; v++) { */
                while ((v = nextVertex()) != -1) {
                        if ((! is_colored(v)) && (nonadjacentcolor(v, chi)))
                                c[v] = chi;
                }
                chi++;
        }
        /*


        while ((v = nextVertex()) != -1) {
                c[v] = 1;
                while ( conflict(v) ) {
                        c[v]++;
                        if (c[v] > chi) chi = c[v];
                }
        }
        */

        return chi -1;
}
```

algo_is.c

```
/*
 *    Name:      algo_is.c
 *    Author:    Andrew Radin
 *    Date:      12/8/99
 *    Purpose:   random independent set graph algorithm
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int is() {

        int chi=1, v;

        pointer=0;
        clear_colors();

        orderRan(G);

#ifdef debugIS
        printf("Welcome to IS\n");
        showgraph(G,n);
        fflush(stdout);
#endif
        while (! all_colored() ) {
#ifdef debugIS
        dump_colors();
        fflush(stdout);
        sleep(1);
#endif
                pointer=0;
                /* for (v = 0 , v < n ; v++) { */
                while ((v = nextVertex()) != -1) {
                        if ((! is_colored(v)) && (nonadjacentcolor(v, chi)))
                                c[v] = chi;
                }
                chi++;
        }
        /*


        while ((v = nextVertex()) != -1) {
                c[v] = 1;
                while ( conflict(v) ) {
                        c[v]++;
                        if (c[v] > chi) chi = c[v];
                }
        }
        */

        return chi -1;
}


int nonadjacentcolor(int v, int chi) {

        /* return true if v is nonadjacent to a vertex colored chi */

        int i;

        for (i=0 ; i < n ; i++) {
```

Page 1

```
#ifdef debugIS
                        printf("Vertex %d has neighbor %d colored %d\n", v, i, chi);
#endif
                        return 0;
                }
        }

#ifdef debugIS
        printf("Vertex %d has no neighbors colored %d\n", v, chi);
#endif
        return 1;

}
```

algo_lf.c


```
/*
 *    Name:     algo_lf.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  largest-first sequential graph algorithms
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int lf() {

        int chi=1, v;

        pointer=0;
        clear_colors();

        orderLF(G);

        while ((v = nextVertex()) != -1) {
                c(v) = 1;
                while ( conflict(v) ) {
                        c(v)++;
                        if (c(v) > chi) chi = c(v);
                }
        }

        return chi;
}

int lfi() {

        int chi=1, v;

        pointer=0;
        clear_colors();

        orderLF(G);

        while ((v = nextVertex()) != -1) {
                c(v) = 1;
                while ( conflict(v) ) {
                        if ( c(v) == chi) {
                                if (! (interc(v)) ) {
                                        c(v)++;
                                        if (c(v) > chi) chi = c(v);
                                }
                        } else {
                                c(v)++;
                                if (c(v) > chi) chi = c(v);
                        }
                }
        }

        return chi;
}

orderLF() {

        int i, v;

        for (i=n-1 , i>=0   i--) {
                for (v=0 ; v<n ; v++) {
                        if (degree(v) == i) {
                                order(o++) = v;
                        }
                }
        }
        order(n) = -1;
}
```

algo_s.c

```
/*
 *    Name:     algo_s.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  random sequential graph algorithms
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int s() {

        int chi=1, v;

        pointer=0;
        clear_colors();

        orderRan(G);

        while ((v = nextVertex()) != -1) {
                c[v] = 1;
                while ( conflict(v) ) {
                        c[v]++;
                        if (c[v] > chi) chi = c[v];
                }
        }

        return chi;
}


int si() {

        int chi=1, v;

        pointer=0;

#ifdef debugSI
        printf("Welcome to si!\n");
#endif

        clear_colors();

        orderRan(G);

        while ((v = nextVertex()) != -1) {
#ifdef debugSI
        printf("vertex %d\n ", v);
#endif
                c[v] = 1;
                while ( conflict(v) ) {
#ifdef debugSI
        dump_colors();
        showgraph(G,n);
#endif
                        if ( c[v] == chi) {
                                if (! (interc(v)) ) {
#ifdef debugSI
        printf("Interchange failed \n");
#endif
                                        c[v]++;
```

Page 1

```
                                }
#ifdef debugSI
                        else
                                printf("Interchange succeeded! \n");
#endif
                                } else {
                                        c[v]++;
                                        if (c[v] > chi) chi = c[v];
                                }
                        }
                }

                return chi;
}

orderRan() {

        int i;
        int ran;

        /* clear out ordering */

        for (i=0 ; i <n   i++) {
                order[i] = -1;
        }

        for (i=0 ; i <n ; i++) {
                ran = rand() % n;
                while (order[ran] != -1) {
                        ran++;
                        if (ran >= n) ran = 0;
                }
                order[ran] = i;
        }
        order[n] = -1;


}
```

algo_sl.c

```
/*
 *    Name:     algo_sl.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  smallest last sequential graph algorithms
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int sg[MAXVERT];

int sl() {

        int chi=1, v;

        pointer=0;
        clear_colors();

        orderSL(G);

#ifdef debugSL
        dump_order();
#endif

        while ((v = nextVertex()) != -1) {
                c[v] = 1;
                while ( conflict(v) ) {
                        c[v]++;
                        if (c[v] > chi) chi = c[v];
                }
        }

        return chi;
}

int sli() {

        int chi=1, v;

        pointer=0;

        clear_colors();

        orderSL(G);

        while ((v = nextVertex()) != -1) {
                c[v] = 1;
                while ( conflict(v) ) {
                        if ( c[v] == chi) {
                                if (! (interc(v)) ) {
                                        c[v]++;
                                        if (c[v] > chi) chi = c[v];
                                }
                        } else {
                                c[v]++;
                                if (c[v] > chi) chi = c[v];
                        }
                }
        }
```

Page 1

```
}

orderSL() {

        int i, v;
        int o;

        if (n > MAXVERT) exit(1);

#ifdef debugSL
        showgraph(G, n);
#endif
        o = n -1;

        /* clear out ordering */
        for (i=0 ; i <n ; i++) {
                order[i] = -1;
        }

        /* make subgraph equal to graph */
        for (i = 0 ; i < n   i++)  {
                sg[i] = 1;
        }

        for (o = n-1 ; o >= 0   o--)  {
                o = doSL(o);
        }

}

doSL(int o) {

        int i, v;

        if (o <= -1) return;

        /* for each type of degree */
        for (i=0 ; i < n , i++) {
                /* for every vertex that might match degree */
                for (v=0 ; v<n ; v++) {
                        /* if vertex is in subgraph */
                        if (sg[v]) {
                                /* and is the degree we want */
                                if (subdegree(v, sg) == i) {
#ifdef debugSL
        printf("%d>%d-%d:  ". o, i, v);
        dumpsg(sg);
#endif

if (order == 0) {
printf("Argh!\n");
exit(1);
}
                                                order[o--] = v;
                                                sg[v] = 0;
                                }
                        }
                }
        }

        return o+1;

}


dumpsg(int sg[]) {
```

```
        for (i = 0 ; i < n ; i++)
                printf("%d ", sg[i]);
        printf("\n");
}
```

algo_slf.c


```
/*
 *    Name:      algo_slf.c
 *    Author:    Andrew Radin
 *    Date:      12/8/99
 *    Purpose:   saturation largest rirst graph algorithms
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int slf() {

        int chi=1, v;
        int i;

        /*
        // color one vertex
        // while there are vertecies
        // choose next vertex w/ greatest colored neighbors.
        */


        pointer=0;
        clear_colors();

#ifdef debugSLF
        showgraph(G, n);
#endif

        for (i = 0 , i < n ; i++) {
                v = nextlSatV(G);
#ifdef debugSLF
        printf("largest saturation vertex is %d\n", v);
#endif
                c[v] = 1;
                /* while there is a conflict find best alternative */
                while ( conflict(v) ) {
#ifdef debugSLF
        dump_colors();
#endif
                        c[v]++;
#ifdef debugSLF
        printf("vertex %d gets color %d\n", v, c[v]);
#endif
                        if (c[v] > chi) chi = c[v];
                }
        }

        return chi;
}

int slfi() {

        int chi=1, v;
        int i;

        pointer=0;
        clear_colors();

        for (i = 0 ; i < n ; i++) {
                v = nextlSatV(G);
                while ( conflict(v) ) {
                        if ( c[v] == chi) {
                                if (! (interc(v)) ) {
                                        c[v]++;
                                        if (c[v] > chi) chi = c[v];
                                }
                        } else {
                                c[v]++;
                                if (c[v] > chi) chi = c[v];
                        }
                }
        }

        return chi;
}

int nextlSatV() {

        /*
        // for each vertex
        // if not colored
        // sum up colored neighbors
        // return max
        */

        int v; /* each vertex in graph */
        int nei; /* neigbors of vertex */
        int cn[MAXVERT]; /* sum of colored neighbors */
        int ret=-1;
        int deg=-1;
        int maxdeg=-1;
        int maxc=-1;


        /* for each vertex in graph */
        for (v = 0 ; v < n , v++) {

#ifdef debugSLF
        printf("v:%d ", v);
#endif
                cn[v] = 0;
                /* if the vertex is not colored */
                if (c[v] == 0) {
#ifdef debugSLF
        printf("nc ");
#endif
                        /* for each of its neighbors */
                        for (nei = 0 ; nei < n ; nei++) {
                                if (G[v][nei] == 1) {
#ifdef debugSLF
        printf("%d:", nei);
#endif
                                        /* if neighbor colored then increment cn total
                                         */
                                        if (c[nei]) {
                                                cn[v]++;
#ifdef debugSLF
        printf("x ");
#endif
                                        }
                                }
                        }
                } else {
#ifdef debugSLF
                printf("colored");
```

```
                }
#ifdef debugSLF
        printf("\n");
#endif
        }


        for (v = 0 , v < n ; v++) {
                deg = degree(v);
                if ((cn[v] > maxc) && (deg > maxdeg) && (c[v] == 0)) [
                        maxc = cn[v];
                        maxdeg = deg;
                        ret = v;
                }
        }
        return ret;

}
```

algo_us.c


```
/*
 *    Name:     algo_us.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  unordered sequential graph algorithms
 *
 *
 */

#include <stdio.h>
#include "algotypes.h"
#include "graphmacro.h"
#include "chicheck.h"
#include "order.h"
#include "colors.h"

int us() {

        int chi=1, v;

        pointer=0;
        clear_colors();

        for (v =  0 ; v < n ; v++ ) {
                c[v] = 1;
                while ( conflict(v) ) {
                        c[v]++;
                        if (c[v] > chi) chi = c[v];
                }
        }

        return chi;
}
```

algotypes.c

```
/*
 *    Name:     algotypes.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Convert strings to algo types and algotypes to strings
 *
 *
 */
#include "algotypes.h"
#include "graphmacro.h"

extern int G[][MAXVERT];
extern int n;

int algoS2I(char * algo) {


        if (strcmp (algo, STR_CHI) == 0)
                return CHI;

        if (strcmp (algo, STR_S) == 0)
                return S;

        if (strcmp (algo, STR_US) == 0)
                return US;

        if (strcmp (algo, STR_SL) == 0)
                return SL;

        if (strcmp (algo, STR_LF) == 0)
                return LF;

        if (strcmp (algo, STR_SLF) == 0)
                return SLF;

        if (strcmp (algo, STR_SI) == 0)
                return SI;

        if (strcmp (algo, STR_SLI) == 0)
                return SLI;

        if (strcmp (algo, STR_LFI) == 0)
                return LFI;

        if (strcmp (algo, STR_SLFI) == 0)
                return SLFI;

        if (strcmp (algo, STR_IS) == 0]
                return IS;

        if (strcmp (algo, STR_GIS) == 0)
                return GIS;

        if (strcmp (algo, STR_SLFDI) == 0)
                return SLFDI;

        if (strcmp (algo, STR_LFDI) == 0)
                return LFDI;

        if (strcmp (algo, STR_SLFAI) == 0)
                return SLFDI;

        if (strcmp (algo, STR_LFAI) == 0)
                return LFDI;

        return UNDEF;
}
```



```
char * algoI2S(int algo) {

        switch(algo) {
                case CHI:
                        return STR_CHI;
                        break;
                case US:
                        return STR_US;
                        break;
                case S:
                        return STR_S;
                        break;
                case SL:
                        return STR_SL;
                        break;
                case LF:
                        return STR_LF;
                        break;
                case SLF:
                        return STR_SLF;
                        break;
                case SI:
                        return STR_SI;
                        break;
                case SLI:
                        return STR_SLI;
                        break;
                case LFI:
                        return STR_LFI;
                        break;
                case SLFI:
                        return STR_SLFI;
                        break;
                case IS:
                        return STR_IS;
                        break;
                case GIS:
                        return STR_GIS;
                        break;
                case SLFDI:
                        return STR_SLFDI;
                        break;
                case LFDI:
                        return STR_LFDI;
                        break;
                case SLFAI:
                        return STR_SLFAI;
                        break;
                case LFAI:
                        return STR_LFAI;
                        break;
                default:
                        return (char *) 0;
        }
}

int runalgo(int algo) {

        switch(algo) {
                case CHI:
                        return chrome(G, n);
                        break;
                case US:
                        return us();
                        break;
                case S:
                        return s();
                        break;
```

```
                case LF:
                        return lf();
                        break;
                case SLF:
                        return slf();
                        break;
                case SI:
                        return si();
                        break;
                case LFI:
                        return lfi();
                        break;
                case SLFI:
                        return slfi();
                        break;
                case SL:
                        return sl();
                        break;
                case SLI:
                        return sli();
                        break;
                case IS:
                        return is();
                        break;
                case GIS:
                        return gis();
                        break;
                case SLFDI:
                        return slfdi();
                        break;
                case LFDI:
                        return lfdi();
                        break;
                case SLFAI:
                        return slfai();
                        break;
                case LFAI:
                        return lfai();
                        break;
                default:
                        return 0;
        }
}
```

chicheck.c

```
/*
 *    Name:     chicheck.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Mainprogram
 *
 *
 */

#include <stdio.h>
#include "graphmacro.h"
#include "algotypes.h"
#include "enabled.h"
#include "makerg.h"

#define EXTERNAL 0
#define INTERNAL 1

int debug0=0;
int debug1=0;

/* the graph in matrix form */
int G[MAXVERT][MAXVERT];
int n;

int checkargs(int argc) {

        if (argc == 2)
                return EXTERNAL;
        else if (argc == 4)
                return INTERNAL;
        else
                error("usage: chicheck in.y\n       chicheck number size density\n");
        return -1;
}

int nextgraph(int runtype, FILE *in) {
        if (runtype == EXTERNAL)
                return ingraph(in, G, &n);
        else if (runtype == INTERNAL)
                return makerg();
        else
                error("should never get here!\n");
}



main(int argc, char *argv[]) {


        int chi;                /* the real chromatic number */
        int estChi[MAXVERT];    /* chromatic estimates by algorithms */
        FILE *in;               /* input file */
        int algo, best;

        int runtype;

        /* check command line and get input file */


        runtype = checkargs(argc);

        if (runtype == EXTERNAL) {
                if (strcmp (argv[1], "-") == 0)
                        in = stdin;
                else
                        in = fopen(argv[1], "r");
                if (!in)
                        error("Couldn't open input file\n");
        }
        else if (runtype == INTERNAL) {
                makerg_init( atoi(argv[3]), atoi(argv[1]));
                n = atoi(argv[2]);
        }
        else
                error("should never get here!\n");


        enabled_init();

        /*while (ingraph(in, G, &n)) { */
        while (nextgraph(runtype, in)) {

                if (n > MAXVERT) {
                        printf("MAXVERT exceeded.  Exiting...\n");
                        exit(1);
                }

                init_degree();
                best = -1;

                /* run algo, save estimated chi and best chi */
                for (algo=FIRSTA ; algo <= LASTA ; algo++) {
                        if (enabled(algo)) {
                                estChi[algo] = runalgo(algo);
                                if (best == -1)
                                        best = estChi[algo];
                                else if (estChi[algo] < best)
                                        best = estChi[algo];

#ifdef VERIFY
                                verify(algo);
#endif
                        }
                }

                /* peg statistics */
                for (algo=FIRSTA ; algo <= LASTA ; algo++) {
                        if (enabled(algo)) {
                                statpeg(algo, estChi[algo] - best);
                                distpeg(algo, estChi[algo]);
                        }
                }
                distpeg(BEST, best);
                statpeg(BEST, 0);
        }
/*******************************************************
        while (ingraph(in, G, &n)) {

                init_degree();

                best = 1;

                chi = runalgo(CHI);
                best = chi;

                distpeg(CHI, chi);
                statpeg(CHI, 0);

                for (algo=FIRSTA+1 ; algo <= LASTA ; algo++) {
                        estChi = runalgo(algo);
#ifdef VERIFY
                        verify(algo);
#endif
```

```
                        distpeg(algo, estChi);
                        if (estChi < chi) {
                                printf("Error: %s is less than Chi (%d, %d)\n",
                                                algoI2S(algo), estChi, chi);
                                showgraph(G, n);
                        } else {
                                statpeg(algo, estChi - chi);
                        }

                }
        }
*****************************************************/
        report();
}
```

```
/*
 *    Name:      colors.c
 *    Author:    Andrew Radin
 *    Date:      12/8/99
 *    Purpose:   operations for handline graph colors
 *
 *
 */

#include <stdio.h>
#include "graphmacro.h"
#include "chicheck.h"

/* c is coloring of vericies */
int c[MAXVERT];

clear_colors() {

        int v;

        for (v =  0 ; v < n ; v++ ) {
                c[v] = 0;
        }
}

int all_colored() {

        int v;

        for (v =  0 ; v < n ; v++ ) {
                if (c[v] == 0) return 0;
        }

        return 1;

}

int is_colored(int v) {

        if (c[v] == 0) {
                return 0;
        } else {
                return 1;
        }
}

int conflict(int v) {

        int a;

        for (a = 0 ;  a < n ; a ++) {
                /* if adjacent with same color and is not itself */
                if ((G[v][a] == 1) && (c[a] == c[v]) && (a != v) )
                        return 1;
        }
        return 0;
}

dump_colors() {
        int i;

        printf("colors:");
        for (i = 0 ; i < n ; i++)
                printf(" %d"  c[i]);
        printf("\n:");
}
```

enabled.c

```
#include "graphmacro.h"
#include "algotypes.h"
#include "enabled.h"
#include <stdlib.h>

#include <stdio.h>

int e[MAXVERT];

void toggle(int algo, char *evalue) {

        if (evalue == 0)
                return;

        if (strcmp("off", evalue) == 0)
                e[algo] = 0;
        else if (strcmp("OFF", evalue) == 0)
                e[algo] = 0;
        else if (strcmp("on", evalue) == 0)
                e[algo] = 1;
        else if (strcmp("ON"  evalue) == 0)
                e[algo] = 1;
        else
                printf("Unknown command: %s\n", evalue);

}

void enabled_init() {

        int algo;
        char *evalue;

        for (algo = FIRSTA , algo <= LASTA , algo++)
                e[algo] = 1;

        for (algo = FIRSTA ; algo <= LASTA ; algo++) {
                evalue = getenv(algoI2S(algo));
                toggle(algo, evalue);
        }


}


int enabled(int algo) {

        return e[algo];
}
```

graphlib.c


```
/*************************************************************************

                        graphlib.c

                        um-modified from spr except:

                        MAXVERT used in showgraph, ingraph,
                        outgraph,

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

#include "graphmacro.h"

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

showgraph(T,n)
/*int T[][SIZE], n;*/
int T[][MAXVERT], n;
{
        int i, j;
        printf("Graph on %d vertices\n",n);
        for (i=0;i<n;i++) {
                for (j=0;j<n;j++) printf("%d",T[i][j]);
                printf("\n");
                }
}

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

ingraph(f,T,pnum)               /* read y-format to matrix */
FILE *f;
/*int T[][SIZE], *pnum;*/
int T[][MAXVERT], *pnum;
{
        int p, q, j, points, bit, e;
        char c;
        j = fscanf(f,"%c",&c);
        if (j!=1) return(0);
        points = 077&c;
        c = ' '; e = 0;
        p = -1; q = 1;
        while (c!='\n') {
                if (fscanf(f,"%c",&c)!=1) return(0);
                bit = 64;
                for (j=0;j<6;j++) {
                        p++; bit /=2;
                        if (p==q) {
                                T[p][p] = 0;
                                p = 0; q++;
                                }
                        if (q>=points) break;
                        T[p][q] = T[q][p] = (c&bit) ? 1 : 0;
                        e += T[p][q];
                        }
                }
        *pnum = points;
        return(e+1);
}

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

outgraph(f,T,num)               /* write y-format to file from matrix */
FILE *f;
/*int T[][SIZE], num;*/
int T[][MAXVERT], num;
{
        int bit, pat, i, j, hibit;
        hibit = 0100;
        fprintf(f,"%c",hibit|num);
        bit = 32; pat = hibit;
        for (j=1;j<num;j++)
        for (i=0;i<j;i++) {
                if (T[i][j]) pat |= bit;
                bit /= 2;
                if (!bit) {
                        fprintf(f,"%c",pat);
                        pat = hibit;
                        bit = 32;
                        }
                }
        if (bit!=32) fprintf(f,"%c",pat);
        fprintf(f,"\n");
        fflush(f);
}

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

compl(g,gbar,n)                 /* complement 0-1 graph matrix */
int g[][SIZE], gbar[][SIZE], n;
{
        int i, j;
        for (i=0;i<n;i++) {
                gbar[i][i] = 0;
                for (j=i+1;j<n;j++)
                gbar[i][j] = gbar[j][i] = 1 - g[i][j];
                }
}

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

int look[33] = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,
                1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5};

ones(x)
int x;
{
        int i;
        i = 0;
        while (x) {
                i += look[x&037];
                x >>= 5;
                }
        return(i);
}

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

getmis(G,n,size,indset)
int G[][SIZE], n, size, *indset;
{
        int list[SIZE];
        return(recindset(G,n,size,indset,list,0,0));
}

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

recindset(G,n,size,indset,list,points,pat)
int G[][SIZE], n, size;
int *indset, *list, points, pat;
{
        int bit, i, j, top, locpat, num, *locindset;

        size--;
        num = 0;
        top = n - size;
        i   = (!points) ? 0 : list[points-1] + 1;
```

```
        if (top<=i) return(num);
        bit = (1<<i);

        for (;i<top;i++,bit<<=1) {
                for (j=0;j<points;j++)
                if (G[list[j]][i]) break;
                if (j<points) continue;
                list[points] = i;
                locpat = pat|bit;
                if (!size) {
                        num++;
                        *indset = locpat;
                        indset++;
                        } else {
                        locindset = indset + num;
                        num +=
                        recindset(G,n,size,locindset,list,points+1,locpat);
                        }
                if (num>=clMAX) error("getmis exceeds clMAX");
                }

        return(num);
}

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

outset(set)
int set;
{
        int i, j;
        i = 1;
        fflush(stderr);
        for (j=0;j<32;j++) {
                if (set&i) printf(" %2d",j);
                i <<= 1;
                }
        printf("\n");
        fflush(stdout);
}

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

outtab(tab,n)
int *tab, n;
{
        int i;
        for (i=0;i<n;i++) printf("%2d ",tab[i]);
        printf("\n");
        fflush(stdout);
}

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

outbit(set)
int set;
{
        int i, j;
        i = 1;
        for (j=0;j<32;j++) {
                printf("%c",(set&i)?'x':' ');
                i <<= 1;
                }
        printf("\n");
}

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

error(message)
```



```
char *message;
{
        printf("%s\n",message);
        exit(1);
}

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

choose(n,k)
int n,k;
{
        int i, t;
        t = n;
        if (n<=0 || k>n || k<0) {
                printf("choose error %d %d\n",n,k);
                exit(1);
                }
        if (k>n/2) k = n-k;
        if (k==1) return(n);
        if (!k) return(1);
        for (i=2;i<=k;i++) t = t*(n-i+1)/i;
        return(t);
}

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

fbit(x)
int x;
{
        int bit;
        if (!x) return(0);
        bit = 1;
        while (!(bit&x)) bit<<=1;
        return(bit);
}

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

hibit(x)
int x;
{
        int bit;
        if (!x) return(0);
        bit = 1;
        while (bit<=x) bit<<=1;
        return(bit>>1);
}
/*****************************************************************************/

outtabnl(tab,n)
int tab[], n;
{
        int i;
        for (i=0;i<n;i++) printf(" %5d",tab[i]);
        fflush(stdout);
}

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

outseq(tab,n)
int tab[], n;
{
        int i;
        for (i=0;i<n;i++) printf("%d",tab[i]);
        printf("\n");
        fflush(stdout);
}
```

```
/*************************************************************************/

outbitnnl(set)
int set;
{
        int i, j;
        i = 1;
        for (j=0;j<32;j++) {
                printf("%c",(set&i)?'x':' ');
                i <<= 1;
                }
}

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

swap(x,y)
int *x, *y;
{
        int i;
        i = *x;
        *x = *y;
        *y = i;
}

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

criticalK4(g,n)                         /* 1 iff g is +e K4 critical */
int g[][SIZE], n;
{
        int i, j, p, q, makesk4;

        for (i=0;i<n-1;i++)
        for (j=i+1;j<n;j++)
        if (!g[i][j]) {
                makesk4 = 0;
                for (p=0;(p<n-1) && !makesk4;p++)
                if (p!=i && p!=j && g[p][i] && g[p][j])
                for (q=p+1;(q<n) && !makesk4;q++)
                if (q!=i && q!=j && g[p][q] && g[q][i] && g[q][j])
                makesk4 = 1;
                if (!makesk4) return(0);
                }
        return(1);
}

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

criticalK5(g,n)                         /* 1 iff g is +e K5 critical */
int g[][SIZE], n;
{
        int i, j, p, q, r, makesk5;

        for (i=0;i<n-1;i++)
        for (j=i+1;j<n;j++)
        if (!g[i][j]) {
                makesk5 = 0;

                for (p=0;(p<n-2) && !makesk5;p++)
                if (p!=i && p!=j && g[p][i] && g[p][j])
                for (q=p+1;(q<n-1) && !makesk5;q++)
                if (q!=i && q!=j && g[p][q] && g[q][i] && g[q][j])
                for (r=q+1;(r<n) && !makesk5;r++)
                if (r!=i && r!=j && g[p][r] && g(q][r] && g[r)[i) && g[r)[j])
                makesk5 = 1;

                if (!makesk5) return(0);
                }
        return(1);
}
```



```
}

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

testchi(G,n,chibound)           /* if chibd>0 returns truth of chi<=chibd */
int G[][SIZE], n, chibound;     /* if chibd=0 returns chi                 */
{
        int indtab[SIZE][clMAX], indcount[SIZE], miscount[SIZE];
        int i, j, k, p, q, x, maxind, allmis[clMAX], allcount;

/* getting independent sets */
        for (i=1;i<n;i++) {
                indcount[i] = getmis(G,n,i,indtab[i]);
                if (!indcount[i]) break;
                }
        maxind = i - 1;

/* getting mis */
        miscount[maxind] = indcount[maxind];
        for (i=maxind-1;i;i--) {
                k = 0;
                for (j=0;j<indcount[i];j++) {
                        x = indtab[i][j];
                        for (p=i+1;p<=maxind;p++)
                        for (q=0;q<miscount[p];q++)
                        if ((x&indtab[p][q])==x) {
                                p = SIZE;
                                break;
                                }
                        if (p>=SIZE) continue;
                        indtab[i][k] = x;
                        k++;
                        }
                miscount[i] = k;
                }

/* info and packing */
        allcount = 0;
        for (i=1;i<=maxind;i++)
        if (indcount[i])
        for (j=0;j<miscount[i];j++)
        allmis[allcount++] = indtab[i][j];

/* painting */
        k = (1<<n)-1;
        if (chibound) return(cover(k,allmis,allcount,chibound));
        j = 2;
        while (!cover(k,allmis,allcount,j)) j++;
        return(j);
}

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

cover(pat,mis,m,size)
int pat, *mis, m, size;
{
        int i, j, locpat, mlow;

        size--;
        mlow = m;
        for (i=0;i<m;i++) {
                j = *mis;
                mis++; mlow--;
                locpat = pat&~j;
                if (!locpat) return(1);
                if (!size || !mlow) continue;
                if (cover(locpat,mis,mlow,size)) return(1);
                }
```

```
        return(0);
}

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

vramsey33(g,n)                          /* 1 iff g is (3,3)v-Ramsey */
int g[][SIZE], n;
{
        int i, j, p, q, H[SIZE][SIZE];
        int ntr, triangles[clMAX];

        q = 1<<(n-1);
        p = (q<<1) + 1;
        compl(g,H,n);
        ntr = getmis(H,n,3,triangles);

        for (i=1;i<q;i++) {
                for (j=0;j<ntr;j++) {
                        p = ones(i&triangles[j]);
                        if (!p || p>2) break;
                        }
                if (j==ntr)
{
outset(i);
return(0);
}
                }

        return(1);
}

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

getallmis(G,n,tab)                      /* getting all independent sets */
int G[][SIZE], n, *tab;
{
        int i, all, num;
        all = 0;
        for (i=1;i<n;i++) {
                num = getmis(G,n,i,tab);
                if (!num) break;
/*
                printf("size %2d indsets %3d\n",i,num);
*/
                all += num;
                tab += num;
                }
        return(all);
}

/*****************************************************************************/
```

graphlib2.c


```
/*
 *    Name:     graphlib2.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Other useful graph operations
 *
 *
 */

#include <stdio.h>
#include "graphmacro.h"
#include "chicheck.h"
int deg[MAXVERT];

int init_degree() {

        int i,v;

        for (v = 0 ; v < n · v++)
                deg[v] = 0;

        for (v = 0 ; v < n , v++) {
                for (i = 0 ; i < n ; i++) {
                        deg[v] += G[v][i];
                }
        }

}

int degree(int v) {
        if ((v >= 0) && (v < MAXVERT))
                return deg[v];
        else
                return -1;
}

int subdegree(int v, int bm[]) {

        int i;
        int deg=0;

        for (i = 0 ; i < n ; i++)  {
                if (bm[i]) {
                        deg += G[v][i];
                }
        ]

        return deg;
}
```

interchange.c

```
/*
 *    Name:     graphlib2.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Other useful graph operations
 *
 *
 */

#include <stdio.h>
#include "graphmacro.h"
#include "chicheck.h"
#include "colors.h"

interc(int v) {
        /*
        for each colored neighbor of v
        if swap is no conflict then swap and return 1
        else return 0
        */

        int i, tmp;

        /* make sure were swapping a colored vertex */
        if (c[v] == 0) return 0;

        for (i=0; i < n ; i++) {
                if (G[v][i] && c[i]) {
                        tmp = c[v];
                        c[v] = c[i];
                        c[i] = tmp;
                        if (conflict(v) || conflict(i)) {
                                tmp = c[v];
                                c[v] = c[i];
                                c[i] = tmp;

                        } else {
#ifdef debugINT
        printf("exchanged %d (%d) and %d (%d)\n", v, c[v], i, c[i]);
#endif
                                return i+1;
                        }
                }
        }
        return 0;
}
```

```
#include <stdio.h>
#include "graphmacro.h"
#include "makerg.h"

#include "chicheck.h"

int density;
int howmany;
int this;

void makerg_init(int d, int n) {
        density = d;
        howmany = n;
        this = 1;
}

int makerg() {

        int i, j;

        if (this > howmany)
                return 0;

#ifdef debugMR
        printf("random graph %d of %d, %d node, %d dense\n", this, howmany, n, density
);
#endif

        for ( i=0 ; i < n ; i++) {
                for ( j=0 ; j < n ; j++) {
                        if ( (rand() % 100) <= density -1) {
                                G[i][j] = 1;
                                G[j][i] = 1;
                        } else {
                                G[i][j] = 0;
                                G[j][i] = 0;
                        }
                }
        }


        /* make sure diagonal is 0 */
        for (i=0 ; i< n ; i++)
                G[i][i] = 0;

        this++;
        return 1;

}
```

order.c


```
/*
 *    Name:     order.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Other useful graph operations
 *
 *
 */

#include <stdio.h>
#include "graphmacro.h"
#include "chicheck.h"

/* order to attempt coloring */
int pointer=0;
int order[MAXVERT+1];

int nextVertex() {

        if (pointer >= MAXVERT) return -1;

        return order[pointer++];
}

int thisVertex(int v) {

        return order[v];
]


dump_order() {
        int i;

        printf("order:");
        for (i = 0 ; i < n ; i++)
                printf(" %d", order[i]);
        printf("\n:");
]
```

stats.c

```
/*
 *    Name:     stats.c
 *    Author:   Andrew Radin
 *    Date:     12/8/99
 *    Purpose:  Keep and report on statistics
 *
 *
 */

#include "algotypes.h"
#include "graphmacro.h"
#include "enabled.h"

/* number of correct estimates by algorithms */
int stat[LASTA+1][MAXVERT+1];

/* actual histogram of colorings */
int dist[LASTA+1][MAXVERT+1];

void initStats(void) {

        int i,j;

        for (i = BEST ; i <= LASTA ; i++ ) {
                for (j = 0 ; j < MAXVERT +1 ; j++ ) {
                        dist[i][j] = 0;
                        stat[i][j] = 0;
                }
        }
}

int distpeg(int algo, int color) {


        if ((algo < BEST) || (algo > LASTA))
                return 0;

        if (color > MAXVERT) {
                printf("Stat handling color overrun (%d)\n", color);
                return 0;
        }

        dist[algo][color]++;
        return 1;

}

int statpeg(int algo, int diff) {

        if ((algo < BEST) || (algo > LASTA))
                return 0;

        if (diff > MAXVERT) {
                printf("Stat handling diff overrun (%d)\n", diff);
                return 0;
        }

        stat[algo][diff]++;
        return 1;

}

void report() {

        char *report;

        report = (char *) getenv("REPORT");
```

Page 1

```
        if ((report) && (strcmp("brief", report) == 0)) {
                summary_report();
        } else {
                summary_report();
                printf("\n");
                detailed_report();
        }
}

summary_report() {

        int i,j;

        int chitot, thistot;

        chitot = 0;
        for (j = 0 ; j < MAXVERT +1; j++) {
                chitot += dist[BEST][j];
        }

        for (i=FIRSTA ; i <= LASTA ; i++)  {

                if (enabled(i)) {
                        printf("%6.2f%% %6s %d/%d\n",
                        ( (float) stat[i][0] / (float) chitot) * 100,
                        algoI2S(i), stat[i][0], chitot );
                }
        }
}

detailed_report() {

        int i,j;

        int lowbound ,


        for (j = 1 ; j < MAXVERT +1 , j++) {
                for (i = BEST , i <= LASTA ; i++ ) {
                        if (dist[i][j])
                                lowbound = j+2;
                }
        }

        if (lowbound > MAXVERT +1) lowbound = MAXVERT +1;

        for (i=FIRSTA ; i <= LASTA , i++)  {

                if (enabled(i)) {
                        printf("%10s:\n",algoI2S(i));
                        for (j = 2 ; j < lowbound ; j++) {
                                printf("%2d colorings: %10d  ", j, dist[i][j]);
                                printf("%2d off: %10d\n", j-1, stat[i][j-1]);
                        }
                        printf("\n");
                }

        }
}
```

```
#include "graphmacro.h"
#include "chicheck.h"
#include "colors.h"
#include "algotypes.h"

verify(int algo) {


        int i,j;

        /* chi does not use the color array, so we don't check it */
        if (algo == CHI)
                return;

        /* check for all vertex colored */
        for (i=0 ; i < n , i++) [
                if (c[i] == 0) {
                        printf("Error: %s vertex %d not colored!\n", algoI2S(algo), i)
;
                }
        }

        /* check for conflicts in coloring */
        for (i=0 , i < n ; i++) {
                for (j=0 ; j < n ; j++) {
                        if ( (G[i][j] == 1) && (c[i] == c[j]) ) {
                                printf("Error: %s vertex %d, %d; adjacent and same col
or %d!\n",
                                                algoI2S(algo), i, j, c[i]);
                        }
                }
        }
}
```