Missouri University of Science and Technology
Scholars' Mine

Computer Science Technical Reports
Computer Science

01 Dec 1993

The Difficulty of Approximating the Chromatic Number for Random Composite Graphs

Jeffrey Wayne Jenness

Billy E. Gillett
Missouri University of Science and Technology

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_techreports

Part of the Computer Sciences Commons

Recommended Citation

Jenness, Jeffrey Wayne and Gillett, Billy E., "The Difficulty of Approximating the Chromatic Number for Random Composite Graphs" (1993). *Computer Science Technical Reports*. 56.
https://scholarsmine.mst.edu/comsci_techreports/56

This Technical Report is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

The Difficulty of Approximating the Chromatic Number for Random Composite Graphs

J. Jenness* and B. Gillett

CSC-93-24

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401

*This report is substantially the Ph. D. dissertation of the first author, completed December 1993

c 1993 ©
Jeffrey Wayne Jenness
all rights reserved

ABSTRACT

Combinatorial Optimization is an important class of techniques for solving Combinatorial Problems. Many practical problems are Combinatorial Problems, such as the Traveling Salesman Problem (TSP) and Composite Graph Coloring Problem (CGCP). Unfortunately, both of these problems are $\mathcal{NP}$-complete and it is not known if efficient algorithms exist to solve these problems. Even approximation with guaranteed results can be just as difficult. Recently, many generalized search techniques have been developed to improve upon the solutions found by the heuristic algorithms.

This paper presents results for CGCP. In particular, exact and heuristic algorithms are presented and analyzed. This study is made to show empirically that CGCP cannot provide guarantees on the approximation using these heuristic methods. In addition, an improvement is presented on the interchange method by Clementson and Elphick that is used with vertex sequential algorithms. This improvement allows graphs of up to 1000 vertices to be colored in considerably less time than previous studies. The study also shows that CDSaturI heuristic does not compete as well with CDSatur as expected for large graphs with edge density of 0.2.

Several $\mathcal{NP}$-completeness theorems are presented and proved. Approximation of CGCP is shown to be as difficult as finding exact solutions if we expect the approximate solutions to fall within a specified bound. These bounds on approximate solutions are shown to be directly related to the bounds that have been proved to exist for the Standard Graph Coloring Problem (SGCP).

Finally, a model of CGCP is developed so that the Tabu Search technique can be applied. Several neighborhoods are developed and tested on 50 and 100 vertex graphs. Timing and performance is analyzed against the heuristics in the previous study. Instances of larger order graphs are used to test the best neighborhood searches with Tabu Search.

ACKNOWLEDGEMENTS

First and foremost I would like to thank God for his rich mercy and understanding in my life and the grace in placing me in a position to do this work. More importantly, I thank God for giving me the strength to finish.

In all thy ways acknowledge him, and he shall direct thy paths. [1]

This document is the collaboration of many persons. I mean this in the sense that I have not worked alone. I have received a great deal of assistance over the years by some very talented and generous people. If it were not for them this work would have not been completed. I would like to first thank my advisor Dr. Billy Gillett for his encouragement and understanding. Also the members of my committee were a great deal of help and support in and out of the classroom. I would also like to thank Dr. Larry Martin, Mrs. Mary Elick and Dr. Joe Shields of Missouri Southern State College who inspired me to take the first step.

I wish to also thank my wife for her patience and endurance. She was also a great support during this work. And of course, my children provided a great many wonderful distractions from my work and I could not have done without them. My parents are also important to me and I thank them for their support.

Dr. Jerry Linnstaedter of Arkansas State University provided encouragement when there was not much to be encouraged about.

Finally, I would like to thank the University of Missouri—Rolla for the Chancellor's Fellowship that I received during the course of this work.

[1] *Proverbs 3:6, King James Bible*

TABLE OF CONTENTS

ABSTRACT iii
ACKNOWLEDGEMENTS iv
LIST OF FIGURES viii
LIST OF TABLES x
I. INTRODUCTION 1
II. PRELIMINARIES 6
A. SET NOTATION AND DEFINITIONS 6
B. GRAPH THEORY 7
C. COMBINATORIAL OPTIMIZATION 13
D. COMPLEXITY THEORY 14
1. Problem Representation 15
2. Measuring Complexity 16
E. THEORY OF *NP*-COMPLETENESS 19
1. *P* and *NP* problems 20
2. *NP*-complete problems 20
3. *NP*-hard problems 21
III. REVIEW OF COMPOSITE GRAPH COLORING 23
A. GRAPH COLORING PROBLEMS 23
B. EXACT ALGORITHMS 25
1. Vertex Sequential 25
2. Color Sequential 29
C. HEURISTIC ALGORITHMS 31
1. Vertex Sequential 31
2. Color Sequential 35
IV. IMPLEMENTATION OF HEURISTIC ALGORITHMS AND RESULTS 37
A. PROBLEM GENERATION 37
B. IMPLEMENTATION OF THE INTERCHANGE METHOD . 38

1. Description of the Implementation 39
2. Timings and Analysis 40
C. EXPERIMENTATION AND RESULTS 44
1. Related Studies 44
2. Testing and Analysis 49
D. IMPLEMENTATION SPECIFICS 63
V. THEORETICAL RESULTS 64
A. THE DECISION PROBLEM IS *NP* 64
B. COMPOSITE GRAPH COLORING IS *NP*-HARD 64
C. STRONGLY *NP*-COMPLETE RESULTS 67
D. COMPLEXITY OF APPROXIMATION 70
VI. TABU SEARCH 78
A. COMPONENTS OF LOCAL SEARCH 78
B. COMPONENTS OF TABU SEARCH 80
1. Simple Tabu Search 80
2. Aspiration Levels 82
3. Implementing Tabu Lists and Aspiration Levels 83
C. TABU SEARCH ALGORITHMS FOR CGCP 84
1. Related Research 84
2. Configuration Space 87
3. Implementation of Components 88
4. Neighborhoods 89
D. ANALYSIS OF ALGORITHMS 93
E. EXPERIMENTATION 96
1. Preliminary Testing 96
2. Testing Parameters 100
3. Testing Larger Graphs 101
F. CONCLUSIONS 104
VII. TABU SEARCH VERSUS HEURISTICS 107

VIII. SUMMARY AND FURTHER RESEARCH 111
BIBLIOGRAPHY 114
APPENDIX 117
VITA 229

LIST OF FIGURES

Figure Page

1. Diagram of graph G_1 . . . 8
2. Diagram of graph G_2 . . . 10
3. Diagram of the graph G_2^c, the complement of G_2 . . . 11
4. Tom, Dick and Harry's scheduling problem . . . 12
5. Coloring a standard graph with the decision problem . . . 22
6. Composite Graph G_c . . . 24
7. Composite graph with static vertex measures . . . 32
8. Reimplementation of the interchange algorithm . . . 41
9. Number of wins from Oakes for graphs $(*, 0.2, \text{TPOI}(1), 25)$ . . . 47
10. Number of wins from Oakes for graphs $(*, 0.5, \text{TPOI}(1), 25)$ . . . 48
11. Coloring results for CDSaturI and CRLF $(*, 0.2, \text{TPOI}(1), 25)$ . . . 50
12. Coloring results for graphs $(*, 0.2, \text{TPOI}(1), 25)$ . . . 51
13. Coloring results for CDSaturI and CRLF $(*, 0.5, \text{TPOI}(1), 25)$ . . . 52
14. Coloring results for graphs $(*, 0.5, \text{TPOI}(1), 25)$ . . . 53
15. Comparing wins for CDS and CDSI heuristics $(*, 0.2, \text{TPOI}(1), 25)$ . . . 55
16. Number of wins for heuristics $(*, 0.2, \text{TPOI}(1), 25)$ . . . 56
17. Number of wins for heuristics $(*, 0.5, \text{TPOI}(1), 25)$ . . . 56
18. Absolute error estimate in heuristics $(*, 0.2, \text{TPOI}(1), 25)$ . . . 59
19. Absolute error estimate in heuristics $(*, 0.5, \text{TPOI}(1), 25)$ . . . 60
20. Relative deviation from the upper bound $(*, 0.2, \text{TPOI}(1), 25)$ . . . 61
21. Relative deviation from the upper bound $(*, 0.5, \text{TPOI}(1), 25)$ . . . 62
22. Algorithm for validating a coloring . . . 65
23. Composite Graph constructed using a Standard Graph . . . 66
24. Coloring a composite graph using the decision problem . . . 67
25. Composite Graph constructed using 3 isomorphic copies of G_c . . . 72
26. Polynomial algorithm for coloring composite graphs . . . 73
27. ϵ-relative algorithm for SGCP . . . 75

28. Local search algorithm 80
29. Simple Tabu Search algorithm 81
30. Function for finding the minimum color for vertex v_i 88
31. Approximations and upper bounds for graphs with $e = 0.2$ 108
32. Approximations and upper bounds for graphs with $e = 0.5$ 109

LIST OF TABLES

Table Page

I. Polynomial versus Exponential Complexity 18

II. Stage 1 of Composite Vertex Sequential Algorithm 28

III. Composite Color Sequential Algorithm 31

IV. Vertex orders for static heuristics 33

V. CDSatur heuristic for composite graph coloring 34

VI. CRLF heuristic for composite graph coloring 36

VII. TPOI(1) Probability Distribution 38

VIII. Timing results for graphs with $e = 0.2$ 42

IX. Timing results for graphs with $e = 0.5$ 42

X. Actual and adjusted times for the reimplementation ($e = 0.2$) . . . 43

XI. Actual and adjusted times for the reimplementation ($e = 0.5$) . . . 43

XII. Interchange for graphs $(1000, *, \mathrm{TPOI}(1), 25)$ 44

XIII. Coloring results from Roberts for graphs $(*, 0.2, \mathrm{TPOI}(1), 25)$ . . . 45

XIV. Number of Wins reported by Roberts $(*, 0.2, \mathrm{TPOI}(1), 25)$ 46

XV. Coloring results from Oakes for graphs $(*, 0.2, \mathrm{TPOI}(1), 25)$ 47

XVI. Coloring results from Oakes for graphs $(*, 0.5, \mathrm{TPOI}(1), 25)$ 48

XVII. Coloring results for graphs $(*, 0.2, \mathrm{TPOI}(1), 25)$ 50

XVIII. Coloring results for graphs $(*, 0.5, \mathrm{TPOI}(1), 25)$ 52

XIX. Probabilistic bounds for graphs with $e = 0.2$ 57

XX. Probabilistic bounds for graphs with $e = 0.5$ 57

XXI. Absolute error estimate in heuristics $(*.0.2, \mathrm{TPOI}(1), 25)$ 59

XXII. Absolute error estimate in heuristics $(*, 0.5, \mathrm{TPOI}(1), 25)$ 60

XXIII. Relative deviation from the upper bound $(*.0.2, \mathrm{TPOI}(1), 25)$ . . . 61

XXIV. Relative deviation from the upper bound $(*, 0.5, \mathrm{TPOI}(1), 25)$ . . . 62

XXV. Results from Jenness and Gillett for CGCP using Tabu Search . . 87

XXVI. Complexity of the Tabu Search algorithms for CGCP 95

XXVII. Results for Tabu Algorithms on graphs $(50, 0.2, \mathrm{TPOI}(1), 25)$ . . . 98

XXVIII. Results for Tabu Algorithms on graphs (50, 0.5, TPOI(1), 25) . . . 98
XXIX. Results for FH moves on graphs (50, 0.2, TPOI(1), 25) 99
XXX. Results for FH moves on graphs (50, 0.5, TPOI(1), 25) 99
XXXI. Number of wins for Tabu Search algorithms 100
XXXII. Results from variations of starting configurations for $e = 0.2$ 102
XXXIII. Results from variations of starting configurations for $e = 0.5$ 102
XXXIV. Results from neighborhood sampling for $e = 0.2$ 103
XXXV. Results from neighborhood sampling for $e = 0.5$ 103
XXXVI. Results of Tabu Search for graphs (100, *, TPOI(1), 25) 105
XXXVII. Results of Tabu Search for graphs (200, *, TPOI(1), 25) 105
XXXVIII. Results of Tabu Search for graphs (300, *, TPOI(1), 2) 106
XXXIX. Best approximations and probabilistic upper bounds for $\chi(G_c)$ . . 108
XL. Best approximations and probabilistic upper bounds for $\chi(G_c)$ . . 109

I. INTRODUCTION

This paper is an examination of some of the techniques used in Combinatorial Optimization. Specifically, the problem of coloring Random Composite Graphs will be examined.

Combinatorial Optimization provides some of the most challenging problems in research. Although progress has been made, especially in the area of classifying problems, the task of finding solutions can still be extremely difficult. In general, a solution is sought that provides the best value of the objective function among *all* possible solutions to the problem. An algorithm that provides solutions of this type is referred to as an *exact algorithm.* For Combinatorial Optimization problems, exact algorithms are simple to construct. As long as the solution set is finite, one would simply write a procedure to generate, in some order, the solutions and exhaustively search the list for the "best" solution. Since the solutions space is well-defined then the writing of the generation routine should not be an issue. Then what is the issue?

Imagine you were a traveling encyclopedia salesman. Your territory consists of 20 different cities. As you travel you realize that you constantly retrace your steps when going from city to city. In order to reduce the time and money you spend, you would like to find a route in which you visit each city only once and you return home when you have completed the route. You find a map and begin to trace the shortest path connecting each pair of cities that you visit on your route. This, you discover, is no easy task since there are 190 pairs of cities that you must find paths between. Once complete, you then realize that the combinations are too many to examine in a reasonable time so you contact a friend who works in the Computer Science Department at a local university. Convinced that this friend can provide services that can easily solve the problem by computer, you explain to him your problem. His reply surprises you. In short, your friend explains that to find the exact solution by exhaustive means you would need to examine 2,432,902,008,176,640,000 possibilities and would take 771.5 years on a computer that could construct and examine 100,000,000 routes each second!

The issue then is to find a "reasonable" solution using a "reasonable" amount of resources for problems such as the Traveling Salesman Problem. Although it is easy to theoretically determine the *existence* of an optimal solution and some of its properties it is practically impossible to *exhibit* such a solution. Of course, reasonable is defined by the problem and its application, but the quest for what seems reasonable leads us to the use of what are called *approximation algorithms.* Approximation algorithms are procedures for solving Combinatorial Optimization problems that will find *near*-optimal solutions, that is, solutions that may not be the best but are reasonably good and can be found using a relatively small amount of resources when compared to exact algorithms.

While contemplating your situation as a salesman, you determine that you just wish to find a good route, one that would reduce your expenses, so you again contact your friend. Upon explaining your request of him, he asks what is meant by "good". You state that if a route could be found that would be within 100 miles of the best route then you would be satisfied. Again you receive a disappointing response. He conveys to you that it has been proven that there is no "easy" procedure which can guarantee such an approximation for your problem. In fact, it is just as easy to solve it exactly!

Such results for the Traveling Salesman Problem (and many other of the $\mathcal{NP}$-complete problems) are well known. Unfortunate as this may seem, we still need to solve such problems. The simplest technique for finding solutions involves choosing any initial configuration that satisfies the constraints of the problem and attempting to improve this configuration by iteration. At each iteration the "neighborhood" of the current configuration is examined. Neighbors (configurations from the neighborhood) are candidates for improving the current configuration. Once a candidate is chosen, this process is repeated on this new configuration. Several issues arise when using this process.

1. How will the process know when the good solution has been found?

   Typically, the process will halt when there are no neighbors that improve the

current configuration. The search process above is referred to as local optimization because of this behavior.

2. How will the process choose the initial configuration?

   Because of the local behavior of the search, the initial configuration is an important one. A poorly chosen initial configuration can result in a poor solution and poor running times.

3. How will the process choose neighborhoods?

   Along with the initial configuration, neighborhood size can affect the quality of the solution. Obviously if the initial neighborhood is the whole of the configuration space the process will return the optimum to the problem. The process will require only one iteration but the examination of the neighborhood will be prohibitively expensive. Likewise if the neighborhood is very small then the process will halt very quickly on a local optimum. If this occurs it is highly unlikely that it will be a good solution.

Examining the above issues more closely reveals the dual nature of search algorithms. Often one must compromise the quality of the solution in order to reduce the cost in time for finding the solution. The better solutions take more time in search. The key to any search algorithm is to examine only as many configurations as needed to find a good solution. Much of the research in this area involves finding "tailored" heuristics for taking advantage of peculiarities of the problem.

More recently, research in this area has been interested in randomized search methods for finding good solutions. These methods can be applied to all Combinatorial Optimization problems and under specified conditions can guarantee that the probability of finding the optimum approaches 1 as the number of iterations approaches infinity. Of course the number of iterations must be finite but convergence will provide us with some assurance that we are asymptotically approaching the optimum for the problem. In addition, these methods help to alleviate some of the problems encountered in local search.

This paper examines Tabu Search [Gl89a, Gl90] as applied to the Composite Graph Coloring Problem. The aim of this research is threefold:

1. Examine the current heuristics associated with Composite Graph Coloring and show empirically that the heuristics perform only moderately well by using the probabilistic bounds provided by Oakes [Oa90].

2. Provide the basic theory to show the difficulty in constructing "good" algorithms for approximating the chromatic number in composite graphs. This includes $\mathcal{NP}$-completeness results.

3. Model the Composite Graph Coloring Problem for use with Tabu Search. Show that Tabu Search can be used to improve the results provided by the heuristics.

The next chapter provides the basic definitions required for understanding the work in subsequent chapters.

Chapter III. introduces the exact and heuristic algorithms that have been developed for the Composite Graph Coloring Problem. The results of two other studies [Ro87, Oa90] as well as an extended study are provided in Chapter IV. This chapter also provides a reimplementation of the interchange method [CE83] used in vertex sequential heuristics. Graphs of order to 1000 are colored using this interchange method and the results are analyzed.

The theory developed in Chapter V. provides results for showing that the Composite Graph Coloring Problem is $\mathcal{NP}$-hard. There are also results to show that approximation is as difficult as finding an exact solution if the approximate solution is guaranteed to be within a specified bound. This chapter ends by showing that the bounds on approximation algorithms for the Standard Graph Coloring Problem are directly related.

Tabu Search is introduced in Chapter VI. A model is developed for the Composite Graph Coloring Problem so that Tabu Search can be applied to coloring random composite graphs. Several neighborhoods are implemented and examined for 50 and

100 vertex graphs. The chapter concludes by using Tabu Search to color graphs of order 200 and 300.

Results from Chapters IV. and VI. are compared and summarized in Chapter VII. The focus of this chapter is to show Tabu Search can be effective at approximating the chromatic number of composite graphs. This, of course, has its cost in run-time but is much less than run-times associated with exact methods.

II. PRELIMINARIES

This chapter provides most of the background material as well as the notation used in subsequent chapters. Words that are being defined will be in *italics.* Set names will be in upper-case, such as S or A, except where defined otherwise. Elements of sets and functions will be in lower-case except where defined otherwise.

A. SET NOTATION AND DEFINITIONS

The natural numbers will be the set $\{0, 1, 2, 3, \ldots\}$ and will be represented by $\mathbb{N}$. The set of integers will be denoted by $\mathbb{Z}$ with the positive integers being $\mathbb{Z}^+$. The set of real numbers will be denoted by $\mathbb{R}$.

The *power set* of a set S is the set of all subsets of S and is represented by $\mathfrak{P}(\mathsf{S})$.

The *cross product* of two sets A and B is the set $\mathsf{C} = \{(a, b) \mid a \in \mathsf{A}\ \mathit{and}\ b \in \mathsf{B}\}$. Where the set A and B are explicit, the cross product is written $\mathsf{A} \times \mathsf{B}$. The elements of C are called *ordered pairs.*

A *relation* ρ on the set $\mathsf{A} \times \mathsf{B}$ is a subset of $\mathsf{A} \times \mathsf{B}$. We say $a \in \mathsf{A}$ is related to $b \in \mathsf{B}$ if and only if $(a, b) \in \mathsf{A} \times \mathsf{B}$ and write $a\rho b$. If σ is a relation on the product $\mathsf{A} \times \mathsf{A}$ then we say σ is a relation on the set A. Given relation σ on the set A, we say σ is *reflexive* if and only if for every $a \in \mathsf{A}$, $a\sigma a$. We say σ is *symmetric* if and only if for every $a, b \in \mathsf{A}$, $a\sigma b$ implies $b\sigma a$. σ is said to be *transitive* if and only if for every $a, b, c \in \mathsf{A}$, $a\sigma b$ and $b\sigma c$ implies $a\sigma c$.

An *order* on a set A is a relation R that is that R is anti-symmetric (for every $a, b \in \mathsf{A}$ such that $a \neq b$ and aRb implies a is not related to b, $a \not R b$) and transitive. An order is a *total order* if for every $a, b \in \mathsf{A}$, aRb or bRa.

A *function* f on the set A into B, denoted $f : \mathsf{A} \to \mathsf{B}$, is a relation of $\mathsf{A} \times \mathsf{B}$ with the condition that if $(a_1, b_1) \in \mathsf{A} \times \mathsf{B}$ and $(a_2, b_2) \in \mathsf{A} \times \mathsf{B}$ and $a_1 = a_2$ then $b_1 = b_2$. The set A is called the *domain* of f and the set B is called the *co-domain* of f. For some element $(a, b) \in f$, a is called the *pre-image* of b and b is called the *image* of a. The image of a can also be written as $f(a)$.

A function is said to be an *injection* if and only if for each $b \in \mathsf{B}$ there is at

most one pre-image in A. A function is said to be a *surjection* if and only if for each $b \in B$ there is at least one pre-image in A. A function is a *bijection* if and only if the function is an injection and a surjection.

A set S is said to be *countable* if and only if S has a finite number of elements or there exists a bijection f such that f maps S onto $\mathbb{N}$.

The *cardinality* of a set S is the number of elements in the set S if S is finite. If S is countably infinite then the cardinality is given to be the same as the cardinality of $\mathbb{N}$ and is represented by $\aleph$. The cardinality for an arbitrary set S will be written using the notation $|S|$.

B. GRAPH THEORY

This section explains the rudiments of Graph Theory which is studied in detail in other texts [Be85, BM76].

Definition 2.1 A *directed graph* G is the tuple (V, E, ψ) where V is a finite set of *vertices* and E is a set of edges and ψ is a function with domain E and co-domain $V \times V$. ∎

The *order* of a graph G, written as $|G|$, is defined to be $|V|$.

Definition 2.2 An *undirected graph* is a graph where the edges are not ordered pairs. An edge is a set represented by $\{v_i, v_j\}$ where $v_i \neq v_j$. Thus $\{v_i, v_j\}$ is equivalent to the edge $\{v_j, v_i\}$. ∎

For the edge (v_i, v_j), v_i is called the *initial endpoint* and v_j is called the *terminal endpoint*. If e is undirected the v_i and v_j are referred to as *endpoints*. An edge is called a *loop* if and only if the endpoints are the same vertex. The degree of some vertex v_i, denoted $\deg(v_i)$, is the number of edges for which v_i is an endpoint.

The above are standard definitions for the directed and undirected graphs. A simpler definition can be used if multiple edges between vertices are not necessary. This simplification will be used for the specification of graphs in this paper. A *graph* G is the tuple (V, E) where V is a finite set of vertices and $E \subseteq V \times V$ is the set of

edges. In not dealing unnecessarily with the function ψ, this will simplify many of the following definitions to set theoretic definitions.

When convenient, the vertex set and edge set of some arbitrary graph G will be denoted as V(G) and E(G) respectively.

A graph $G = (V, E)$, is called a *simple graph* if and only if E has no edges that are loops. A simple graph G is called a *complete graph* if and only if for each pair of distinct vertices, v_i and v_j in V, the edge (v_i, v_j) is in E.

Example 2.1 Let the graph G_1 be defined as follows:

$$V(G_1) = \{0, 1, 2, 3, 4\}$$
$$E(G_1) = \{(0,1), (0,3), (1,1), (1,4), (2,4), (3,2), (4,2)\}$$

For the edge $(2, 4)$, 2 is the initial endpoint and 4 is the terminal endpoint. Also since this edge is present, we say that 2 is adjacent to 4 and 4 is adjacent to 2. Because edge $(1, 1)$ is a loop, G_1 is not simple. Figure 1 is a diagram of G_1.

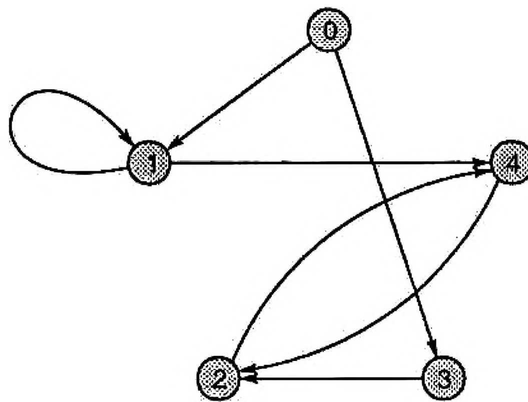


Figure 1. Diagram of graph G_1

Two vertices, v_i and v_j, are said to be *adjacent* if and only if there is some $e \in E$ for which v_i and v_j are endpoints. The adjacency matrix for a simple graph G is a matrix, $\mathcal{M}$, defined over the set $\{0, 1\}$ where the value in row i and column j of $\mathcal{M}$ is 1 if $(v_i, v_j) \in E$ and 0 otherwise.

Example 2.2 From the previous example the adjacency matrix is as follows:

$$\mathcal{M} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

An adjacency list for a graph is the function $f : V \to \mathfrak{P}(V)$ where for all $v \in V$, $f(v) = \{u \in V | \{v, u\} \in E\}$.

An *independent set* of a simple graph is a set $S \subseteq V(G)$ such that for each pair of vertices, $v_i, v_j \in S$ then v_i and v_j are not adjacent. A *maximally independent set* of a simple graph G is the set M such that if the set S is any other independent set then $M \not\subset S$. Note that the definition of maximally independent sets is not what is defined as the maximal independent set in other texts. The *maximal independent set* for a simple graph G is an independent set with the largest cardinality of all independent sets of G. Since both definitions are important and will be used in this paper, it is important that the distinction be clear.

Example 2.3 The simple undirected graph G_2 given as the diagram in Figure 2 has the following maximally independent sets.

$$\{0,1,3,4\},\ \{0,3,5\},\ \{1,2\},\ \{2,5\},\ \{6\}$$

In this example, the set $\{0,1,3\}$ is an independent set for graph G_2 but it is not maximally independent since it is a subset of another independent set of G_2. Clearly, the maximal independent set is the independent set $\{0,1,3,4\}$.

A complementary notion of an independent set of a graph is a clique. A *clique* of a simple graph G is a subset of vertices $S \subseteq V(G)$ such that for each pair of distinct vertices, say v_i and v_j, in S then v_i and v_j are adjacent in G. The *clique number* of a graph is the cardinality of the largest clique of G.

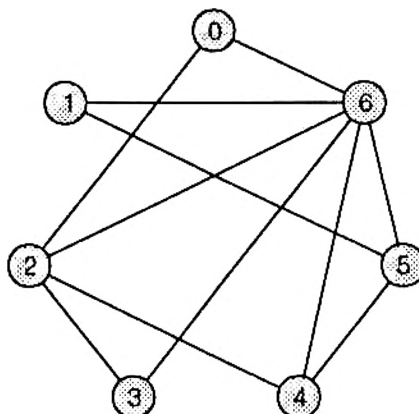


Figure 2. Diagram of graph G_2

A graph H is a *subgraph* of the simple graph G if and only if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ with the requirement that for each edge in E(H) both endpoints must be in V(H). For some subset S of V(G), the *induced subgraph*, written G[S], has the edge set E_S with elements (v_i, v_j) for which both v_i and v_j are in S. Thus the subgraph of G induced by S is given as $G[S] = (S, E_S)$.

Using the above definition of induced subgraph, it is possible to describe a clique in another way. A clique C of a simple graph G is a subset of V(G) such that the graph G[C] is a complete graph.

The *complement* of a simple graph G is the simple graph G^c such that for each pair of distinct vertices v_i and v_j, then $(v_i, v_j) \in E(G^c)$ if and only if $(v_i, v_j) \notin E(G)$. The complement of graph G_2 is in Figure 3.

There are some interesting relationships of the graph G and its complement G^c that are worth mentioning. The following theorem can be easily shown.

Theorem 2.1 Given a simple graph G, $S \subseteq V(G)$ is an independent set of G if and only if S is a clique of G^c.

A *path* from vertex v_i and v_j in a simple graph G, is a sequence of vertices beginning with v_i and ending with v_j and each pair of juxtaposed vertices in the sequence are adjacent in G. A *connected component* of a simple graph G is the induced subgraph

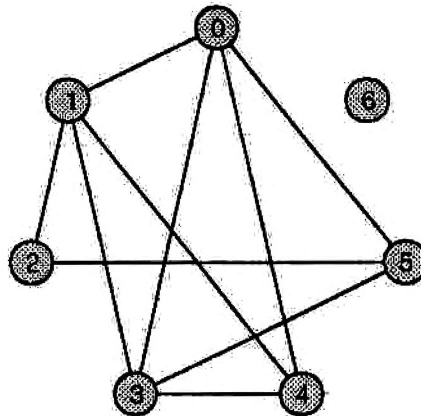


Figure 3. Diagram of the graph G_2^c, the complement of G_2

G[S] where there exists a path between each pair of distinct vertices in S. A graph is said to be connected if it is itself a connected component.

Another useful idea is that of combining two graphs to form another graph. The *union* of two disjoint graphs G and H, written G + H, is the graph with vertex set $V(G) \cup V(H)$ and edge set $E(G) \cup E(H)$. The *join* of two disjoint graphs, denoted by $G \oplus H$, is the union of G and H with the addition of all edges such that one endpoint is in V(G) and the other is in V(H).

This section will end with an introduction to the graph coloring problems. These formalizations model such real world problems as time tabling, the register allocation problem in compiler optimization, computer CPU scheduling problems and job shop scheduling.

Given a simple undirected graph G, a *coloring* of G is a function $\kappa : V(G) \rightarrow \mathbb{Z}^+$ such that if $(v_i, v_j) \in E(G)$ then $\kappa(v_i) \neq \kappa(v_j)$. In other words, each pair of adjacent nodes in G must be assigned a different integer (or color). To see how this might model a time scheduling problem consider the following example.

Example 2.4 Suppose there is a group of speakers that are intending to make a series of presentations. In fact, there are a total of eight one-hour presentations and three speakers. Tom, one of the speakers, will make four of the eight presentations while both of the other speakers, Dick and Harry, will present two of the talks. Also

suppose that Tom's first talk and Harry's second talk cannot coincide because of required resources. There has also been a request not to schedule Tom's third talk at the same time as Harry's second talk or Dick's first talk. The graph representing this problem is diagrammed in Figure 4. The nodes represent talks that are to be scheduled and each edge denotes a scheduling conflict between the two talks that act as endpoints for that edge.

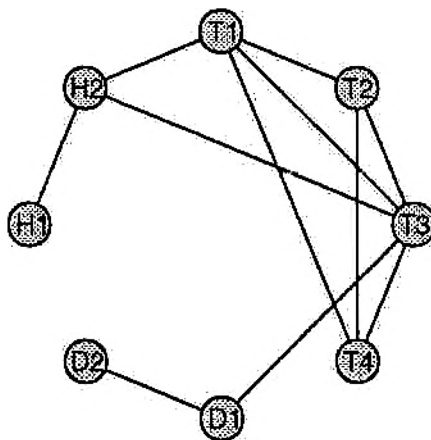


Figure 4. Tom, Dick and Harry's scheduling problem

In order to solve the problem, time slots must be given to each one-hour talk. The time slots become the colors with which the above graph is "painted". It is easy to see that a valid coloring for the graph would provide a time schedule for the presentations that would avoid the conflicts.

An easy solution is to give each presentation a separate time slot. This would not be a feasible solution if the allotment for all talks must be within five hours. If this were the case then the task would be to determine a coloring in which a maximum of five colors will be used. This problem is a difficult one to solve, in fact, we will later see that it is in the important class of $\mathcal{NP}$-complete problems. In general, a minimum number of time slots required to schedule all of the presentations would be the solution that we seek. The minimum number of colors needed to color a graph G is called the *chromatic number* of the graph and is denoted by $\chi(\mathsf{G})$.

For the example scheduling problem, the chromatic number is four and a feasible

coloring for the graph is: $\kappa(T1) = 1$, $\kappa(T2) = 2$, $\kappa(T3) = 3$, $\kappa(T4) = 4$, $\kappa(D1) = 1$, $\kappa(D2) = 3$, $\kappa(H1) = 2$, $\kappa(H2) = 4$.

C. COMBINATORIAL OPTIMIZATION

Combinatorial Optimization (hence referred to as CO) is a problem classification where solving a problem involves searching over many possible solutions to find the "best" solution.

Definition 2.3 An *optimization problem* is a set of *instances* Π where each instance $\mathsf{P} \in \Pi$ is the tuple $(\mathfrak{F}, c)$ where $\mathfrak{F}$ is the set of all solutions for the problem instance P and c is a function mapping defined by $c : \mathfrak{F} \rightarrow \mathbb{R}$. $\mathfrak{F}$ is referred to as the *configuration space.* Elements of $\mathfrak{F}$ are called *configurations.* The function c is called the *objective function.* An optimization problem is a *combinatorial optimization problem* if and only if $\mathfrak{F}$ is countable. ∎

The point $p \in \mathfrak{F}$ is the *solution* to the instance $\mathsf{P} \in \Pi$ if when P is a maximization problem then $\forall x \in \mathfrak{F}$, $c(x) \leq c(p)$ and when P is a minimization problem then $\forall x \in \mathfrak{F}$, $c(x) \geq c(p)$. p is called the *global optimum* and will be referred to as OPT(P).

Inherent in solving any optimization problem are the two procedures for defining $\mathfrak{F}$ and c. The first procedure when given a combinatorial object will determine the *feasibility* of the object (that is, membership in $\mathfrak{F}$). This is referred to as the *feasibility problem* in CO. The second of the procedures when given a feasible configuration will calculate the *cost* for that configuration.

Example 2.5 The Traveling Salesman Problem (TSP) described in the introduction is a CO problem. Formally, the TSP is a graph problem in which the cities are represented by the nodes of the graph and the distances by weights placed on the edges of the graph.

In particular the graph is a complete undirected graph, that is, it is assumed that there will be an undirected path between any pair of cities. Solving this problem requires finding a "circuit" (each city is visited exactly once) on a graph such that the sum of the weights on each edge of the path is minimized. In this case the set $\mathfrak{F}$

is the set of paths traversing every node of the graph and forming a circuit. The cost function maps each path to the sum of the weights on the edges.

Example 2.6 The Max-Flow Problem is used in determining the maximum amount of flow of material through a network. An instance of this might be the movement of manufactured goods through a network of shipment points from the single warehouse to single retail outlet. This problem can also be described as a graph problem in which the nodes are the stop-over shipment points with one of the nodes being the initial shipment point (the warehouse) and one being the final shipment point (the retail outlet). The edges are "one-way" connections between intermediate shipment points that are assigned capacities.

A configuration for the Max-Flow Problem would be an assigned amount of "flow" to each edge that does not exceed the capacity for that edge. A further constraint is imposed that requires the flow into each node be equal to the flow out of that node. The cost function would assign to each configuration the sum of the flow out of the initial node.

Algorithms exist for the solution of the above two problems but the Max-Flow problem is easier to solve than TSP. TSP is classified as a Nondeterministic Polynomial Problem while the Max-Flow is a Polynomial Problem.

D. COMPLEXITY THEORY

The nature of some problems prevents them from being solved easily. The study of this nature involves classification of problems by *complexity*. Roughly speaking, we are interested in the amount of effort involved in solving a particular type of problem. This effort is characterized by the space and time requirements of a given procedure used to solve instances of the problem. The formal study of complexity theory is based on Turing machines. An intuitive approach provides enough understanding to make use of complexity theory in studying efficiency of algorithms. First, we must be clear on what we mean by "algorithm". The accepted definition is that stated by Church's Thesis:

An algorithm is any procedure that can be realized by a Turing machine.

A more intuitive approach is to exhaust all of the implied properties of an algorithm from the above definition [Kn73].

1. An algorithm is a procedure that is finitely specified.
2. An algorithm must stop with an answer after performing a finite number of steps.
3. An algorithm must have each step well-defined and effective. That is, each step should not be ambiguous and could be carried out in a finite amount of time by a human problem solver with paper and pencil.
4. An algorithm has zero or more inputs.
5. An algorithm has one or more outputs.

While computability theory [LP81] is the study of these qualitative properties of algorithms, complexity theory is the study of the quantitative properties of algorithms.

1. Problem Representation In examining the space and time requirements of an algorithm, we realize that these requirements are dependent on the size of the problem instance that is input to the algorithm. For instance, multiplying two numbers takes less time than multiplying 1000 numbers although we perform the same procedure for both instances. Thus, the complexity of the algorithm must be studied in terms of how it is affected by the size of an instance. The formal specification requires one to code (or encode) the problem instance so that it is suitable for input to a Turing machine with alphabet $\{0,1\}$. The coding scheme is fixed for the entire problem domain and is clearly dependent on the details of the Turing machine. Therefore any "reasonable" coding scheme should not affect the measure of complexity. The size of the problem is now described in terms of the coding scheme. Given a problem instance $\mathsf{P} \in \Pi$ and the coding scheme ϵ, the coded problem is given by $\epsilon(\mathsf{P})$. The size of P is the given as $|\epsilon(\mathsf{P})|$, that is, the length of the coded string.

(To simplify later discussions the coding scheme will be implied and the size of the problem will be referred to as n.)

Customarily, complexity measure does not encode the problem instances as 0-1 strings but uses the problem's natural representation. This practice arises because algorithms are measured against algorithms for the same problem. Thus some problem parameter is used as a measure of the size of the problem. For instance, in graph coloring the number of nodes in the graph along with the number of edges in the graph is an indicator of the size of the instance. One must exercise caution in measuring complexity in this manner since subtleties in representation can occur and result in erroneous conclusions.

2. Measuring Complexity In most cases space complexity is not a concern since data can usually be generated when needed and then discarded. When coloring a graph for instance, the number of possibilities is combinatorially large (that is, exponential), but not all of the configurations need to be generated at once. With a simple enumeration scheme, the only configuration that must be kept at any particular point in time is the best current coloring. The main concern is the time required in examining each of these graphs. Since algorithms are machine independent, the time measurements should also reflect this independence of a particular implementation. This is accomplished by giving the measure relative to the number of steps necessary for the algorithm to solve the problem instance of size n. A step is considered to be an addition, subtraction, multiplication, division or comparison. This process yields a function in n and is used in practice to report the complexity of both algorithms and problems.

The use of complexity measures in the study of algorithms provides a means of analyzing an algorithm's performance. The general practice is to consider an instance of size n that would cause the algorithm $\mathcal{A}$ to take the most possible steps in deriving a solution. That is, choose the conditions so that the algorithm would exercise the greatest number of steps in finding a solution. The number of steps derived from this *worst-case* analysis provides a performance guarantee by the algorithm over the

specified problem domain. The function representing this upper-bound for a problem instance of size n, called $f_{\mathcal{A}}(n)$, is then used to rank the algorithm's efficiency for solving the problem at hand.

Definition 2.4 [2] Let $f(n)$ be a function defined by, $f : \mathbb{N} \to \mathbb{N}$. The class of functions $O(f(n))$ is defined as all functions $g : \mathbb{N} \to \mathbb{N}$ for which there exists constants M and c such that for every $n > M$, $g(n) \leq cf(n)$. ∎

A consequence of the above definition provides the following inclusions ($k > 1$):

$$O(1) \subset O(\log_k(n)) \subset O(n) \subset O(n^k) \subset O(k^n) \subset O(n^n)$$

The common classes given above are constant, logarithmic, linear, polynomial, exponential and super-exponential, respectively.

Definition 2.5 An algorithm $\mathcal{A}$ with complexity function $f_{\mathcal{A}}(n)$ is said to have complexity $O(g(n))$ if and only if $f_{\mathcal{A}}(n) \in O(g(n))$. ∎

The notion of efficiency is provided thus:

1. Two algorithms $\mathcal{A}$ and $\mathcal{B}$ are of equivalent efficiency, written $\mathcal{A} \sim \mathcal{B}$, provided $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$.

2. Algorithm $\mathcal{A}$ is more efficient than algorithm $\mathcal{B}$, written $\mathcal{A} \prec \mathcal{B}$, provided $f(n) \in O(g(n))$ and $g(n) \notin O(f(n))$.

A problem's complexity is manifested by the algorithms used to solve instances of that problem. Thus we determine a problem's complexity by exhibiting an algorithm of a given complexity or prove no such algorithm exists. In the former case, if we are able to demonstrate a given algorithm with complexity $O(f(n))$ for the problem then we know that the problem's complexity is at most $O(f(n))$. The later case provides evidence that the problem is of a higher complexity. This is a formidable

[2] This definition of $O(f(n))$ restricts the functions being examined to those that are only interesting in the study of algorithms, hence the slightly altered form.

Table I. Polynomial versus Exponential Complexity

n	$f(n)$ n	n^2	n^3	n^{10}	2^n	10^n
1	10^0	10^0	10^0	10^0	2	10^1
10	10^1	10^2	10^3	10^{10}	10^3	10^{10}
100	10^2	10^4	10^6	10^{20}	10^{30}	10^{100}
1000	10^3	10^6	10^9	10^{30}	10^{301}	10^{1000}

task for many problems. An historic example of this is the Linear Programming problem. In 1947, Dantzig developed the Simplex algorithm [Da63] for solving the Linear Programming problem. Although the number of steps for many practical problems is a polynomial in the size of the problem, there were instances where the algorithm would take an exponential number of steps. Many researchers attempted to find more efficient methods but their failures reinforced the belief that the Linear Programming problem had exponential complexity. This belief prevailed until 1979, the year Khachian introduced the Ellipsoid algorithm [Kh79] for Linear Programming. This algorithm was provably of a polynomial complexity and promised to be one of the greatest achievements in the field.

Definition 2.6 An algorithm $\mathcal{A}$ is said to be of polynomial complexity if and only if $f_{\mathcal{A}}(n) \in O(n^k)$ for some fixed $k \in \mathbb{N}$. An algorithm $\mathcal{A}$ is said to be of exponential complexity if and only if $f_{\mathcal{A}}(n) \in O(k^n)$ for some fixed $k \in \mathbb{Z}^+ - \{1\}$. ∎

Although the definition of algorithm complexity in terms of function classes provides for fine distinctions between algorithms, a coarser distinction appears between the algorithms of polynomial complexity and those that are not. Table I has the number of operations for a problem instance of size n with the given complexity functions. This illustrates the difficulty in solving large problem instances using exponentially complex algorithms. This difference is important in practice where only relatively small problems can be solved using exponential algorithms.

Definition 2.7 A problem P is said to be *tractable* if and only if there exists an algorithm $\mathcal{A}$ for solving P and $\mathcal{A}$ is of polynomial complexity. If no polynomial algorithm exists for P, then P is said to be *intractable.* For intractability, the nonexistence must be provably so. ∎

The dividing line between tractable and intractable problems is one of great concern to the practitioner since intractable problems cannot be solved without difficulty. Thus the demonstration of the Ellipsoid algorithm for the Linear Programming problem was of great importance because it showed that the Linear Programming problem was tractable instead of intractable as once believed. As we will see later, $\mathcal{NP}$-complete problems are a class of problems just beyond the horizon of tractability. Since many practical problems are known to be in this class the separation of tractable and intractable problems becomes a chasm for which no bridge has been found.

E. THEORY OF $\mathcal{NP}$-COMPLETENESS

$\mathcal{NP}$-complete problems are a class of computationally difficult problems in the class of $\mathcal{NP}$ problems. The theory is an attempt to examine their complexity, but has not yet succeeded in this objective. It is not known whether these problems are tractable or intractable but it can be said that if one is tractable then all $\mathcal{NP}$-complete problems are tractable and if one is intractable then all are intractable. Thus, these problems are just outside the class of tractable problems. The study of $\mathcal{NP}$-complete problems is based on decision problems.

Definition 2.8 A *decision problem*, Δ, is the ordered pair (D, S) where D is the set of *instances* and $S \subseteq D$ is the set of *instance-solutions.* Given an instance $d \in D$, we determine if $d \in S$. If $d \in S$ we answer "yes" otherwise we answer "no". ∎

The feasibility problem in optimization is exactly the same as the decision problem: Given domain D and feasibility region $\mathfrak{F}$ and a point $p \in D$, p is feasible if $p \in \mathfrak{F}$ otherwise p is infeasible.

1. $\mathcal{P}$ and $\mathcal{NP}$ problems The two classes of decision problems important to the theory of $\mathcal{NP}$-complete problems are $\mathcal{P}$ and $\mathcal{NP}$.

Definition 2.9 The class of polynomial decision problems $\mathcal{P}$ contain only those decision problems Δ for which there is an algorithm $\mathcal{A}$ to solve Δ with polynomial complexity. ∎

The class of decision problems $\mathcal{NP}$ are described formally as those decision problems that have an algorithm $\mathcal{A}$ that has polynomial complexity when realized as a nondeterministic Turing machine. To solve the problem deterministically requires augmenting the problem instance with a "certificate of verification". This certificate provides apriori knowledge of the solution path used by the nondeterministic Turing machine. So the class of $\mathcal{NP}$ decision problems can be viewed as those decision problems that can "verify" the membership of an instance $d \in \Delta$ in the set of instance-solutions S using a polynomial amount of steps in the size of the problem.

Definition 2.10 The class of nondeterministic polynomial problems $\mathcal{NP}$ contain only those decision problems Δ for which there is an algorithm $\mathcal{A}$ of polynomial complexity that can verify a problem instance given a certificate of verification. ∎

Clearly, the class of $\mathcal{P}$ decision problems is a subset of $\mathcal{NP}$ since the problems in $\mathcal{P}$ can be solved in polynomial time without a certificate of verification. In this case, we would use the polynomial-time algorithm and a blank certificate of verification.

2. $\mathcal{NP}$-complete problems The class of $\mathcal{NP}$-complete problems are decision problems that are the most difficult to solve in $\mathcal{NP}$. All of this is based on what is called *polynomial reducibility.*

Definition 2.11 Let Δ_1 and Δ_2 be decision problems. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be algorithms for solving the decision problems, respectively. Δ_1 is *polynomially reducible* to Δ_2 provided there is a function $f : \Delta_1 \rightarrow \Delta_2$ that satisfies the following conditions:

1. There is an algorithm $\mathcal{A}$ for computing f that has polynomial complexity.

2. For each instance $d \in \Delta_1$, algorithm $\mathcal{A}_1$ answers "yes" for d if and only if $\mathcal{A}_2$ answers "yes" for $f(d)$.

∎

Reducibility between two problems provides a relative measure of difficulty between the problems. Suppose Δ_1 polynomially reduces to Δ_2, then we know that if Δ_2 is tractable, so is Δ_1. The converse of this statement is not necessarily true. Therefore, Δ_1 is at most as difficult to solve as Δ_2.

Definition 2.12 A problem $\Delta \in \mathcal{NP}$ is said to be $\mathcal{NP}$-complete if and only if every decision problem in $\mathcal{NP}$ is polynomially reducible to Δ. ∎

Cook [Co71] proved the existence of the first $\mathcal{NP}$-complete problem, the satisfiability problem (SAT). Since that time many other problems have been shown to be $\mathcal{NP}$-complete [Ka72, GJ79].

3. $\mathcal{NP}$-hard problems The study of $\mathcal{NP}$-complete problems is based solely on decision problems, but most problems that are solved in practice are not decision problems. Karp [Ka72] provides another type of reducibility that relates decision problems with other types of problems, in particular, with optimization problems.

Definition 2.13 Let Δ_1 and Δ_2 be decision problems. Let $\mathcal{A}_1$ be algorithm for solving the decision problem Δ_1. Δ_1 is *one-many polynomially reducible* to Δ_2 provided there is an algorithm $\mathcal{A}_2$ for solving Δ_2 that satisfies the following conditions:

1. Algorithm $\mathcal{A}_2$ uses algorithm $\mathcal{A}_1$ as a "subroutine".

2. Algorithm $\mathcal{A}_2$ has polynomial complexity if each "call" to $\mathcal{A}_1$ is considered as a single step in $\mathcal{A}_2$

∎

Using one-many polynomial reducibility, we are able to show that optimization problems are at least as hard as the associated decision problem. If the associated decision problem is $\mathcal{NP}$-complete then we say that the optimization problem is $\mathcal{NP}$-hard.

Definition 2.14 A problem Π is said to be $\mathcal{NP}$-hard if there is a decision problem Δ that is $\mathcal{NP}$-complete and Δ is one-many polynomially reducible to Π. ∎

To show that the optimization problems are at least as hard we can construct a binary search using the associated decision problem as a subroutine. For example, consider the problem of standard graph coloring. Given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, we wish to know the minimum number of colors needed to color G. The associated decision problem is: "Can G be colored with k or less colors?" Karp showed this decision problem is $\mathcal{NP}$-complete [Ka72]. Now, let `Colorable` be the algorithm for the associated decision problem for graph coloring. The algorithm for solving the graph coloring problem is then given in Figure 5.

```
Let U = sizeof(V)
Let L = 0
Let k = U div 2 + U mod 2
While U > L + 1 Do
    If Colorable(G, k) Then
        Let U = k
    Else
        Let L = k
    End If
    Let k = (U - L) div 2 + (U - L) mod 2 + L
End While
Return k
```

Figure 5. Coloring a standard graph with the decision problem

If we consider the call to `Colorable` as no different from an addition or multiplication then we could show that the above algorithm has complexity $O(\log(n))$ where n is the order of G. Since $\log(n) \in O(n^1)$ then the above algorithm has polynomial complexity and therefore the graph coloring problem is $\mathcal{NP}$-hard.

III. REVIEW OF COMPOSITE GRAPH COLORING

The graph coloring problem serves as a model for many applications such as the time-tabling problem, the job-shop scheduling problem and the register allocation problem in compilers. Graph coloring comes in two forms: standard graph coloring and composite graph coloring.

A. GRAPH COLORING PROBLEMS

An instance of the standard graph coloring problem (referred to as SGCP) is the tuple (G, K) where G is a simple graph and K is a set of coloring functions defined to be

$$\mathsf{K} = \{\kappa | \kappa : \mathsf{V}(\mathsf{G}) \rightarrow \mathbb{Z}^+\}$$

As stated in Section II.B., a coloring of G is a function such that for all $v_i, v_j \in \mathsf{V}(\mathsf{G})$, if $\{v_i, v_j\} \in \mathsf{E}(\mathsf{G})$ then $\kappa(v_i) \neq \kappa(v_j)$. In terms of a CO problem, the set of configurations is the subset of K where each function is bounded above by $|\mathsf{V}(\mathsf{G})|$, that is,

$$\mathfrak{F} = \{\kappa \in \mathsf{K} | \forall v \in \mathsf{V}(\mathsf{G}), \kappa(v) \leq |\mathsf{V}(\mathsf{G})|\}$$

The cost function c is defined to be

$$c(\kappa) = \max_{v \in \mathsf{V}(\mathsf{G})} \{\kappa(v)\}$$

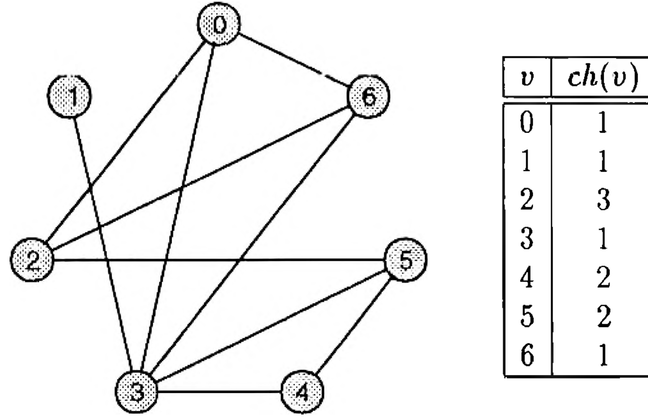
The minimum of $c(\kappa)$ for all $\kappa \in \mathfrak{F}$ is called the chromatic number of the graph G and is usually denoted by

$$\chi(\mathsf{G}) = \min_{\kappa \in \mathfrak{F}} \{c(\kappa)\}$$

Thus the SGCP is a CO problem where the objective is to determine for each instance the chromatic number of the graph.

The composite graph coloring problem (referred to as CGCP) is the graph coloring problem defined over composite graphs. A composite graph differs from the standard graph in that a chromaticity is defined for each vertex in the graph. The chromaticity

is stated as a function ch that maps $\mathsf{V(G)}$ into $\mathbb{Z}^+$, so we say $ch(v)$ is the chromaticity for vertex v. The chromaticity represents the length of an interval of colors that each vertex requires. In scheduling problems, this represents the varying size of the time slots required for the jobs being scheduled. In most cases the time slots must be contiguous and CGCP is a model for problems where this is a requirement. One other requirement that we will make on this function is that there exist at least one pair of vertices v_i and v_j such that $ch(v_i) \neq ch(v_j)$; we will see later the importance of this distinction.



v	$ch(v)$
0	1
1	1
2	3
3	1
4	2
5	2
6	1

Figure 6. Composite Graph G_c

A composite graph is defined as $\mathsf{G}_c = (\mathsf{G}, ch)$ where G is a simple graph and ch is a chromaticity function as described above. A coloring of the composite graph G_c is a function $\kappa : \mathsf{V}(\mathsf{G}_c) \rightarrow \mathbb{Z}^+ \times \mathbb{Z}^+$ such that for each $v \in \mathsf{V}(\mathsf{G}_c)$, if $\kappa(v) = (a, b)$ then $ch(v) = b - a + 1$ and for all $v_i, v_j \in \mathsf{V}(\mathsf{G}_c)$, if $\{v_i, v_j\} \in \mathsf{E}(\mathsf{G}_c)$ and $\kappa(v_i) = (a_i, b_i)$ $\kappa(v_j) = (a_j, b_j)$ then either $a_i > b_j$ or $a_j > b_i$. An instance of the CGCP is the tuple $(\mathsf{G}_c, \mathsf{K})$ where G_c is a composite graph and K is the set of all colorings of G_c.

The CGCP can also be described as a CO problem $(\mathfrak{F}, c)$ where $\mathfrak{F}$ is defined to be the functions of K that are bounded by $\sum_{v \in \mathsf{V}(\mathsf{G}_c)} ch(v)$. That is,

$$\mathfrak{F} = \left\{ \kappa \in \mathsf{K} \mid \max_{v \in \mathsf{V}(\mathsf{G}_c)} \{b | \kappa(v) = (a, b)\} \leq \sum_{v \in \mathsf{V}(\mathsf{G}_c)} ch(v) \right\}$$

The cost function c is defined to be

$$c(\kappa) = \max_{v \in V(G_c)} \{b | \kappa(v) = (a, b)\}$$

As for the SGCP, the chromatic number of G_c is $\chi(G_c) = \min_{\kappa \in \mathfrak{F}} \{c(\kappa)\}$. The objective is, as in the SGCP, to find $\chi(G_c)$.

As was previously mentioned, the chromaticity function for the CGCP had to have at least two vertices with different chromaticities. Clementson and Elphick [CE83] did not make this requirement. Oakes [Oa90] and Roberts [Ro87] stated that under these conditions SGCP⊆CGCP because a chromaticity function can be given for any instance of the SGCP where $ch(v) = 1$ for all v in the standard graph. Thus it is clear that the CGCP is at least as hard as the SGCP. We altered the definition of a composite graph provided by Oakes and Roberts in order to show that the CGCP is no harder to solve than the SGCP, in fact, we will show in Chapter V. that just as SGCP is $\mathcal{NP}$-hard so is CGCP $\mathcal{NP}$-hard.

The rest of this chapter reviews the algorithms used to color composite graphs. Most of these algorithms are derived from algorithms developed for SGCP.

B. EXACT ALGORITHMS

Exact algorithms for CGCP find the chromatic number for the composite graph. The literature contains four exact algorithms. Oakes [Oa90] generalizes the vertex sequential and color sequential coloring algorithms first described by Korman [Ko79] for the SGCP. Oakes also adapted the integer linear program formulation for the SGCP given by Christofides [Ch71]. Roberts [Ro87] developed a mixed-integer linear program for CGCP. We only examine the vertex sequential and color sequential coloring algorithms because of their relevance to the associated heuristic algorithms.

1. Vertex Sequential Given a composite graph G_c, the vertex sequential coloring algorithm comes in two stages. The first stage finds an initial coloring for G_c and the second stage performs exhaustive search by backtracking. The backtracking is done in the most efficient manner by pruning useless or redundant paths in the

search process.

In order to describe the algorithm, we first describe the components that assist in accelerating the search process. The first component is used in both stages and involves ordering the vertices as the graph is colored and recolored. The vertices that constitute a maximal clique are placed first in the sort order so that the search will prune useless paths earlier in the search. This idea is based on the observation that if we color a clique then any other attempt to color the clique will result in either a redundant coloring or a coloring that exceeds the number of colors required for coloring the clique. Unfortunately, the problem of finding a maximal clique is $\mathcal{NP}$-hard [Co71]; therefore a heuristic order is used instead.

A partial coloring of a composite graph G_c is a coloring function f, defined on a subset of $V(G_c)$. We say a graph G_c is partially colored on V' if $\forall v \in V'$, $f(v)$ is defined and $\forall u \in V(G_c) - V'$, $f(u)$ is undefined.

Definition 3.1 If graph G_c is partially colored on V' then the *colored degree* of a vertex $v \in V(G_c) - V'$ is the number of distinct colors used by all vertices in V' adjacent to v, that is,

$$\delta_c(v) = \sum_{i=1}^{k} f(v,j)$$

where k is the number of colors used by the vertices in V' and

$$f(v,j) = \begin{cases} 1 & \text{if } \exists u \in V' \wedge \{v,u\} \in E(G_c) \wedge \kappa(u) = (a,b) \wedge a \leq j \leq b \\ 0 & \text{otherwise} \end{cases}$$

∎

Definition 3.2 If graph G_c is partially colored on V' then the *uncolored degree* of a vertex $v \in V(G_c) - V'$ is the number of vertices in $V(G_c) - V'$ adjacent to v,

$$\delta_u(v) = \sum_{\forall u \in V(G_c) - V'} g(v,u)$$

where

$$g(v,u) = \begin{cases} 1 & \text{if } \exists u \in \mathsf{V}' \wedge \{v,u\} \in \mathsf{E}(\mathsf{G}_c) \\ 0 & \text{otherwise} \end{cases}$$

■

Definition 3.3 If graph G_c is partially colored on V' then the *uncolored adjacent chromatic degree* of a vertex $v \in \mathsf{V}(\mathsf{G}_c) - \mathsf{V}'$ is the sum of the chromaticities of vertices in $\mathsf{V}(\mathsf{G}_c) - \mathsf{V}'$ that are adjacent to v. The uncolored adjacent chromatic degree is given by $\delta_a(v) = \sum_{\forall u \in \mathsf{V}(\mathsf{G}_c) - \mathsf{V}'} [g(v,u)ch(u)]$ ■

The heuristic order suggested by Oakes is given by sorting primarily on colored degree in descending order, i.e., the larger the degree, the higher in the sort order the vertex will be. Second, third and fourth orders are given to be chromaticity, uncolored adjacent chromatic degree and uncolored degree respectively, again with a descending sort.

The first stage begins by assigning the above order to the vertices. A coloring function is then constructed by sequentially assigning to each vertex the set of consecutive colors with the smallest possible starting color. The order of the vertices is kept dynamically: after each assignment the measures for the uncolored vertices are updated and the uncolored vertices are resorted and the "largest" vertex is chosen to be colored next. (The vertices do not actually need to be sorted; a simple linear search of the list to identifiy the largest vertex is all that is required.) Thus, the coloring function κ at vertex v_1, is given the value $\kappa(v_1) = (1, ch(v_1))$. If the graph is partially colored on $\{v_1, v_2, \ldots, v_i\}$ then the value of coloring function for vertex v_{i+1} is $\kappa(v_{i+1}) = (a, a + ch(v_{i+1}) - 1)$ where

$$a = \min\{b | \forall j \leq i+1, \{v_j, v_{i+1}\} \in \mathsf{E}(\mathsf{G}_c) \wedge \kappa(v_j) = (c,d) \Rightarrow c \geq b + ch(v_{i+1}) \vee b > d\}$$

The maximum starting color that needs to be considered for v_{i+1} is $\sum_{j=1}^{i} ch(v_j) + 1$. Table II shows stage 1 for the composite graph in Figure 6. The measures for sorting are the entries in the table in the form: $(\delta_c(v), ch(v), \delta_a(v), \delta_u(v))$.

Table II. Stage 1 of Composite Vertex Sequential Algorithm

iteration	coloring	vertex						
		0	1	2	3	4	5	6
1	$\kappa(2) = (1,3)$	(0,1,5,3)	(0,1,1,1)	(0,3,4,3)	(0,1,7,5)	(0,2,3,2)	(0,2,6,3)	(0,1,5,3)
2	$\kappa(5) = (4,5)$	(3,1,2,2)	(0,1,1,1)	—	(0,1,7,5)	(0,2,3,2)	(3,2,3,2)	(3,1,2,2)
3	$\kappa(0) = (4,4)$	(3,1,2,2)	(0,1,1,1)	—	(2,1,5,4)	(2,2,1,1)	—	(3,1,2,2)
4	$\kappa(6) = (5,5)$	—	(0,1,1,1)	—	(2,1,4,3)	(2,2,1,1)	—	(4,1,1,1)
5	$\kappa(4) = (1,2)$	—	(0,1,1,1)	—	(2,1,3,2)	(2,2,1,1)	—	—
6	$\kappa(3) = (3,3)$	—	(0,1,1,1)	—	(4,1,1,1)	—	—	—
7	$\kappa(1) = (1,1)$	—	(0,1,1,1)	—	—	—	—	—

Stage 2 of the algorithm provides two upper bounds on the chromatic number of the graph with which to prune the search paths. The first bound provides a way of pruning redundant paths. This is based on the following result [Oa90].

Theorem If a composite graph G_c is partially colored on the set of vertices $\{v_1, v_2, \ldots, v_i\}$ then redundant colorings are avoided if v_{i+1} is given the coloring assignment $\kappa(v_{i+1}) = (a, a + ch(v_{i+1}) - 1)$ where

$$a \leq \max\{b | \exists j \leq i, \kappa(v_j) = (c, b)\} + max\{ch(v_j) | 1 \leq j \leq n\}$$

In other words, the colors considered for vertex v_{i+1} need not exceed $k_1 + k_2 + ch(v_{i+1})$ where k_1 is the number of colors used to color the vertices $\{v_1, v_2, \ldots, v_i\}$ and k_2 is the maximum chromaticity for any vertex in the graph.

The second bound provides a way to prune paths for colorings that exceed the current best coloring of the graph established by the search. The upper bound on the optimum number of colors is initially provided in stage 1. Suppose that $K^{(0)}$ is the number of colors used by the initial coloring. In the first iteration of stage 2, if we attempt to recolor vertex v_{i+1} with a color greater than $K^{(0)}$, then we would certainly exceed the upper bound by completing the current coloring. In fact, we need not consider any color greater than or equal to $K^{(0)}$ because we wish only to improve the upper bound, not match it. The above is continued through all iterations. If $K^{(j)}$ is the current upper bound on the chromatic number at the jth iteration then the

starting color assigned to any vertex v at iteration $j+1$ should not exceed $K^{(j)} - ch(v)$.

Therefore, if we have a partial coloring of the graph on $\{v_1, v_2, \ldots, v_i\}$ then the starting color for vertex v_{i+1} need not exceed the minimum of the previous two bounds. That is, if $\kappa(v_{i+1}) = (a, b)$

$$a \leq \min\{k_1 + k_2, K^{(j)} + ch(v_{i+1})\}$$

Now each iteration of stage 2 starts with the graph completely colored. Stage 1 provides this for the first iteration of stage 2. The first step is to identify in the jth iteration the first vertex that is colored with $K^{(j-1)}$. Suppose that vertex v_i was this vertex. Then the vertex v_{i-1} is recolored by assigning a new starting color that is as small as possible but is greater than its current color and less than or equal to $\min\{k_1 + k_2, K^{(j-1)} + ch(v_{i-1})\}$. If successful then the vertices $\{v_i, v_{i+1}, \ldots, v_n\}$ are recolored as was done in stage 1, with dynamic reordering. If v_{i-1} cannot be recolored then backtracking continues to v_{i-2} and recoloring is attempted for this vertex and so on. Stage 2 continues until backtracking returns to vertex v_1, at which point it halts with the chromatic number as the final upper bound $K^{(m)}$ where m is the last iteration.

2. **Color Sequential** The color sequential algorithm assigns colors sequentially to groups of vertices. The algorithm for CGCP is an extension of the color sequential coloring algorithm for SGCP given by Korman [Ko79]. The extension is described by Oakes [Oa90].

In general, the algorithm provides an exhaustive search as does the vertex sequential algorithm but assigns colors to a maximal independent set of vertices beginning with the color 1. Coloring continues until all vertices are colored by selecting another set of independent vertices remaining in the graph. Backtracking will be used to search all possible combinations of the choices of the set of independent vertices.

Let S_1 be a maximal independent set of vertices of the graph G_c. Then all vertices in S_1 are assigned colors 1 to $\texttt{minch}(S_1)$. Where $\texttt{minch}(S_1) = \min\{ch(v) | v \in S_1\}$.

Clearly, some vertices may be partially colored. The next set of independent vertices will be selected such that those not completely colored in the previous iteration will continue to be colored. Thus, we choose a maximal independent set of vertices from the subgraph induced by removing all vertices completely colored. If S_2 is this set then it must satisfy the following condition:

$$\forall v \in S_1, ch(v) > \texttt{minch}(S_1) \Rightarrow v \in S_2$$

Of course, there may be more than one choice of S_2; this is where backtracking is used to test all possible choices. If all vertices have been colored in S_1 then every independent set of the induced subgraph is a candidate. Choosing the set S_3 is similar to the choosing S_2 except that it must contain all partially colored vertices from the previous two iterations. This process continues until all vertices are colored. Once the graph is colored then the algorithm backtracks to the previous iteration where there was a choice of the independent set of vertices. Another selection is made and coloring again proceeds forward until completed. The algorithm halts when we backtrack to S_1 and we have exhausted all possible choices of the maximal independent sets of G_c for starting the coloring process. Table III shows several iterations of the color sequential algorithm for the graph in Figure 6. Note HCA is the highest color assigned at iteration i.

The chromatic number of the graph is maintained during this search and can be used in eliminating some of the search paths. Gillett and Jenness [GJ91] describe a pruning technique which will minimize backtracking. The basic algorithm is augmented with the maximal cliques in the graph. The maximal clique of the graph G_c with the largest chromatic sum determines a lower bound on $\chi(\mathsf{G}_c)$. At each iteration of coloring this information is also kept in order to determine a lower bound on the number of colors required for all uncolored (or partially colored) vertices. At iteration i, given the next assignable color is k then the path is abandoned if $k + B \geq M$ where B is the largest chromatic sum for any maximal clique of the graph of uncolored

Table III. Composite Color Sequential Algorithm

iteration i	independent sets	S_i	minch(S_i)	HCA	uncolored vertices
1	{0,2,4},{0,3},{1,2,6},{3,5,6},{4,5}	{0,2,4}	1	1	{2,4}
2	{1,2,6},{3,5,6},{2,4}	{2,4}	1	2	{2}
3	{1,2,6},{3,5}	{1,2,6}	1	3	{}
4	{3,5}	{3,5}	1	4	{5}
5	{5}	{5}	1	5	{}
6	{0,2,4},{0,3},{1,2,6},{3,5,6},{4,5}	{0,3}	1	1	{}
7	{2,4},{1,2,6},{5,6},{4,5}	{2,4}	2	3	{2}
8	{2},{1,6},{5,6}	{2}	1	4	{}
9	{1,6},{5,6}	{1,6}	1	5	{}
10	{5}	{5}	2	7	{}
11	{2,4},{1,2,6},{5,6},{4,5}	{1,2,6}	1	2	{2}
12	{2,4},{4,5}	{2,4}	2	4	{}
13	{5}	{5}	2	6	{}

vertices and M is the current best coloring.

C. HEURISTIC ALGORITHMS

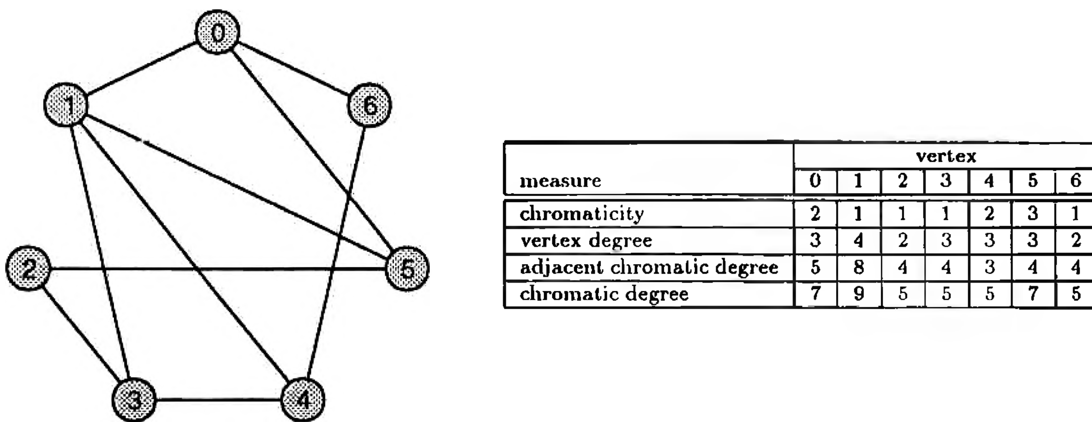
Many heuristics for the SGCP have been developed on the basis of the vertex sequential algorithm. Generalizations of these algorithms have been developed for use with CGCP. These generalizations are primarily due to Clementson and Elphick [CE83], Roberts [Ro87] and Oakes [Oa90]. Roberts also developed color sequential heuristic for CGCP based on a color sequential heuristic for SGCP.

1. Vertex Sequential All of the heuristic algorithms based on the vertex sequential algorithm in the literature use a single pass over the vertices to find a coloring of the graph. The simplest of these algorithms uses a static ordering of the vertices. Given an order of the vertices $v_1, v_2, v_3, \ldots, v_n$ then the algorithm begins by coloring v_1 with colors $(1, ch(v_1))$. The second vertex v_2 is given the colors $(1, ch(v_2))$ if $\{v_1, v_2\} \notin \mathsf{E}(\mathsf{G}_c)$ and the colors $(ch(v_1)+1, ch(v_2)+ch(v_1))$ otherwise. To color vertex v_i, we select the smallest possible starting color a defined by:

$$a = \min\{c | \forall j < i, \kappa(v_j) = (d, e) \wedge \{v_i, v_j\} \in \mathsf{E}(\mathsf{G}_c) \Rightarrow c > e \vee c + ch(v_i) - 1 < d\}$$

This process continues until all vertices are colored. No backtracking occurs in the heuristics.

Two other static measures used are chromatic degree and adjacent chromatic degree. The adjacent chromatic degree is the same measure as the uncolored adjacent chromatic degree for partially colored graphs with the graph completely uncolored. The chromatic degree of a vertex $v \in \mathsf{E}(\mathsf{G}_c)$ is defined to be the sum of the chromaticities of all vertices adjacent to v and $ch(v)$. Figure 7 contains a composite graph and the static measures: chromaticity, degree, adjacent chromatic degree, and chromatic degree which are used in the static-ordering heuristics.



measure	vertex						
	0	1	2	3	4	5	6
chromaticity	2	1	1	1	2	3	1
vertex degree	3	4	2	3	3	3	2
adjacent chromatic degree	5	8	4	4	3	4	4
chromatic degree	7	9	5	5	5	7	5

Figure 7. Composite graph with static vertex measures

Clementson and Elphick gave the first heuristics based on the vertex sequential coloring algorithm. They provided two static ordering rules for sequentially coloring the vertices. The rules are called LF1 and LF2 (standing for Largest-First). The LF1 rule specifies a descending order based on chromaticity of the vertices with secondary order (also descending) based on chromatic degree for the vertices. Rule LF2 uses descending chromatic degree as the primary order and descending chromaticity for the secondary order. Another rule called CLF, which uses a similar static order as the LF1, uses chromaticity as the first sort order followed by the adjacent chromatic degree and a third descending sort based on the degree of the vertex. Roberts also used the Largest-First techniques to develop the LFCD and LFPH ordering rules. The LFCD rule ordered the vertices decreasing using the product of vertex chromaticity and vertex degree. LFPH used a pigeon hole measure $PH(v)$ defined to be the sum

Table IV. Vertex orders for static heuristics

heuristic	order						
	highest			⟶			lowest
LF1	5	0	4	1	2	3	6
LF2	1	5	0	4	2	3	6
CLF	5	0	4	1	3	2	6
CSL	5	0	4	1	3	6	2

of the chromatic degree of v and the product of $ch(v) - 1$ and the degree of v. This measure was devised to determine the difficulty with coloring a specific vertex [Ro87]. Finally, Oakes generalized an ordering rule for SGCP called CSL (Composite Smallest Last) in which the rule orders the vertices so that the *last* vertex colored will be the vertex with the minimum chromaticity. A tie is broken by choosing the vertex to color last with the smallest adjacent chromatic degree, and if a tie remains then the vertex to color last will be the vertex with lowest degree. Table IV shows the vertices sorted by several of the above rules for the graph in Figure 7.

The dynamic reordering algorithms behave as the static ordering algorithms but use a measure that is calculated during the coloring process. The set of uncolored nodes are ordered to determine the best candidate to color next. Three such algorithms have been proposed: two were developed by Roberts and one algorithm was described by Oakes.

The algorithms by Roberts are DYNPH and DYNFPH. These are based on a pigeon-hole measure and a floating point pigeon-hole measure [Ro87]. The algorithm described by Oakes (based on an algorithm for SGCP developed by Brelaz [Br79]) called CDSatur provides some of the best results of both the static and dynamic ordering schemes. The CDSatur ordering rule colors the next vertex by selecting the vertex with maximum chromaticity followed by maximum colored degree, maximum uncolored adjacent chromatic degree and maximum uncolored degree. This order is the reverse of the order defined by the exact vertex sequential algorithm. Table V

Table V. CDSatur heuristic for composite graph coloring

iteration	coloring	vertex						
		0	1	2	3	4	5	6
1	$\kappa(5) = (1,3)$	(2,0,5,3)	(1,0,8,4)	(1,0,4,2)	(1,0,4,3)	(2,0,3,3)	(3,0,4,3)	(1,0,4,2)
2	$\kappa(0) = (4,5)$	(2,3,2,2)	(1,3,5,3)	(1,3,1,1)	(1,0,4,3)	(2,0,3,3)	—	(1,0,4,2)
3	$\kappa(4) = (1,2)$	—	(1,5,3,2)	(1,3,1,1)	(1,0,4,3)	(2,0,3,3)	—	(1,2,2,1)
4	$\kappa(1) = (6,6)$	—	(1,5,1,1)	(1,3,1,1)	(1,2,2,2)	—	—	(1,4,0,0)
5	$\kappa(6) = (3,3)$	—	—	(1,3,1,1)	(1,3,1,1)	—	—	(1,4,0,0)
6	$\kappa(2) = (4,4)$	—	—	(1,3,1,1)	(1,3,1,1)	—	—	—
7	$\kappa(3) = (3,3)$	—	—	—	(1,4,0,0)	—	—	—

demonstrates the CDSatur algorithm for the graph in Figure 7. The table entries are of the form: $(ch(v), \delta_c(v), \delta_a(v), \delta_u(v))$.

The ordering rules can be augmented with color interchange techniques that will improve the coloring given by the above heuristic algorithms. Clementson and Elphick describe the interchange method that shows the best results in practice. Oakes described another interchange method but found that it performed poorly both in speed and improvement of the coloring.

Given a graph G_c with the vertices ordered $v_1, v_2, v_3, \ldots, v_n$, suppose that the heuristic algorithm has colored vertices $v_1, v_2, \ldots, v_{i-1}$. Let M be the number of colors used to color these vertices. Also let the smallest color set assignable to v_i be (a, b). If $b \leq M$ then the coloring is performed and the algorithm proceeds to vertex v_{i+1}. If $b > M$ then a color interchange is attempted to minimize the the additional colors needed. If the interchange fails then v_i is colored with (a, b) and the algorithm continues coloring with v_{i+1}. The interchange method of Clementson and Elphick attempts to identify the set of vertices that can interchange colors with v_i.

The first step in determining the success for an interchange, is to find the possible starting colors for vertex v_i and the vertices that can be used in the color interchange. For a fixed color $k < a$, this is done by identifying a unique vertex $u \in \{v_1, v_2, \ldots, v_{i-1}\}$ adjacent to v_i such that if the coloring of u is (d, e) then $k \leq d \wedge k + ch(v_i) - 1 \geq d$ or $k \leq e \wedge k + ch(v_i) - 1 \geq e$. Thus, we construct the set U_1 of the pairs (u, k) identifying the vertex u as a candidate for interchange

when assigning the starting color k to v_i.

The second step is to ensure that the interchange produces a coloring for vertices $v_1, v_2, \ldots, v_i$ so that we use less than b colors. This is done by constructing the set U_2 of triples (u, k, m) where $(u, k) \in U_1$ and m is the smallest starting color assignable to u if v_i is given the starting color k. All members of U_2 must satisfy the condition $m < b$. The set U_2 identifies all vertices and the color interchange required to reduce the number of colors over the choice of the colors (a, b) for the vertex v_i. The interchange selected is (u, k, m) that minimizes $\max\{k, m\}$ for all elements of U_2.

Clearly, if either U_1 or U_2 are empty then the interchange fails and v_i is assigned the colors (a, b). The algorithms that will be implemented with interchange are CDSatur and CLF and will be referred to as CDSaturI and CLFI respectively.

2. Color Sequential The heuristic algorithms for the color sequential approach are given by Roberts. These algorithms are a generalization of the color sequential algorithm developed by Leighton for SGCP [Le79]. The heuristic processes all vertices that can be colored starting with a color k before any vertex whose starting color is greater than k. The color sequential algorithms are RLF1 and RLFD1.

The basic algorithm for both heuristics begins at each iteration with a starting color k. Three sets are maintained during the iteration: K, I, and U. Set K is initially empty but at the end of the iteration will contain all vertices that have been assigned the color k. I will also be empty at the beginning of the iteration. Finally, set U is the set of all uncolored vertices in the graph at the beginning of the iteration. To begin the iteration, a primary selection rule chooses the best possible candidate u for giving the starting color k from the set U. Vertex u is moved to K and all vertices in U that are not adjacent to u are moved to I. Thus, I is that set of all uncolored vertices that are independent of u and are candidates for a coloring starting with color k. U is the set of uncolored vertices that cannot be given the starting color k because they are adjacent to u. While I is not empty then a secondary selection rule chooses a vertex from I to place in K. If w is the vertex chosen then w is moved to K and all vertices in I adjacent to w are moved to U. If I is empty then the iteration

Table VI. CRLF heuristic for composite graph coloring

iteration	starting color	U	I	K
1	1	{0,1,2,3,4,5,6}	{}	{}
primary selection		{0,1,2}	{3,4,6}	{5}
secondary selection		{0,1,2,3,6}	{}	{4,5}
2	3	{0,1,2,3,6}	{}	{}
primary selection		{0,1,2}	{6}	{3}
secondary selection		{0,1,2}	{}	{3,6}
3	4	{0,1,2}	{}	{}
primary selection		{1}	{2}	{0}
secondary selection		{1}	{}	{0,2}
4	6	{1}	{}	{}
primary selection		{}	{}	{1}
secondary selection		{}	{}	{1}

ends and the next iteration begins with color $k + 1$.

The differences between RLF1 and RLFD1 are the primary and secondary selection rules. For RLF1 the primary selection rule chooses the vertex from U with the maximum chromaticity with ties broken by choosing the vertex with the largest chromatic degree in U. The secondary selection rule for RLF1 uses maximum chromaticity followed by maximum adjacent chromatic degree in I and minimum chromatic degree in U to select the next vertex to color. The RLFD1 algorithm replaces in both of the selection rules the maximum degree of the vertex in place of the maximum chromaticity of the vertex. Oakes suggested an alternate set of rules. The primary selection rule would choose a vertex from U with maximum chromaticity, maximum chromatic degree in U and maximum degree in U (in that order of precedence). The secondary selection rule used the vertex with maximum chromaticity. Ties are broken by selecting the vertex with the maximum adjacent chromatic degree in U, maximum degree in U, minimum chromatic degree in I and minimum degree in I (in that order of precedence). This will be the algorithm that is referred to as CRLF. Table VI illustrates the CRLF algorithm for the graph in Figure 7.

IV. IMPLEMENTATION OF HEURISTIC ALGORITHMS AND RESULTS

This chapter provides empirical results using the best heuristics provided by Oakes [Oa90] and Roberts [Ro87]. The motivation for providing these results is threefold:

1. Introduce an implementation of the interchange method that allows coloring large graphs in a much reduced time frame. This implementation will be used to color graphs of up to 1000 vertices. The results will be compared to that of Oakes and the time reduction analysed.

2. Provide empirical results to support the theoretical findings in Chapter V. by using the probabilistic bounds for the random composite graphs found by Oakes [Oa90].

3. Provide a basis for the study of the Tabu Search method for composite graphs. The results for the best heuristics are compared to the results for Tabu Search in Chapter VII.

A. PROBLEM GENERATION

A set of random composite graphs will be described by the tuple $(\mathtt{ord}, e, \psi, n)$. Where $\mathtt{ord}$ will be the order of the graphs, e is the *edge density*, ψ is the *chromaticity distribution* and n is the cardinality of the set. The edge density is a discrete random variable that determines the probability that an edge is included in the graph. $\text{Prob}\{\{v_i, v_j\} \in \mathsf{E}(\mathsf{G}_c)\} = e$ and $\text{Prob}\{\{v_i, v_j\} \notin \mathsf{E}(\mathsf{G}_c)) = 1 - e$. The chromaticity distribution is a discrete random variable that determines the probabilities for the chromaticities of the vertices. The chromaticity distribution that will be used is a truncated Poisson with parameter $q = 1$. That is, for all $v \in \mathsf{V}(\mathsf{G}_c)$,

$$\text{Prob}\{ch(v) = k\} = \frac{q^k}{(e-1)k!} \quad k = 1, 2, \ldots$$

Table VII. TPOI(1) Probability Distribution

k	Prob$\{p = k\}$
1	0.582
2	0.291
3	0.097
4	0.024
5	0.005
6	0.001
7	0.000

Table VII shows the probabilities used for this study. This distribution will be referred to as TPOI(1).

The test sets used will be that generated by Oakes [Oa90] (see `gengraph.pas` in the Appendix). The sets tested will be of the orders 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000. For each order, two sets will be generated: one with an edge density of 0.2 and another with edge density of 0.5. All sets will use the chromaticity distribution TPOI(1). The number of graphs tested in each set will be 25.

B. IMPLEMENTATION OF THE INTERCHANGE METHOD

The interchange method proves to be beneficial to the vertex sequential algorithms described in the previous chapter. Unfortunately, the implementations provided in the literature do not facilitate the coloring of larger graphs because of the expense of time. Oakes attempted to color graphs of order up to 500 using the interchange method. The CDSatur algorithm with interchange used on the average 12987.45 seconds for graphs of order 500 and edge density 0.5. The reimplementation of the interchange method with the CDSatur algorithm in this research consumed about 422.1 seconds on the same graphs. This time reduction allows the coloring of very large graphs using the interchange technique with any vertex sequential algorithm. Graphs of order 1000 and edge density 0.5 were colored in an average time of 4294.0 seconds.

1. Description of the Implementation Clementson and Elphick [CE83] described the interchange method using four sets. Given a partial coloring of a graph G_c on $V' = \{v_1, v_2, \ldots, v_{i-1}\}$, let v_i be the next vertex to color. Let M be the number of colors used to color the vertices in V' and (a, b) the smallest assignable color set for v_i. The color set P is constructed by finding colors p such that $1 \leq p < a$ and there is a unique vertex $v_j \in V'$ that is adjacent to v_i that uses some or all of the colors in the color set $(p, p + ch(v_i) - 1)$. As before in the interchange described previously, if P is empty then v_i is assigned the color set (a, b) and the algorithm procedes to v_{i+1}. If P is not empty then the set W is constructed which is the vertices $v_j \in V'$ associated with the colors in P. Now a color set Q is constructed that identifies the smallest starting color for each vertex $v_j \in V'$ when vertex v_i is assigned the color set $(p_j, p_j + ch(v_i) - 1)$ (where p_j is the color corresponding to v_j). Lastly, a third color set is produced that identifies all possible interchanges. R is the set of all starting colors assignable to v_i such that the number of colors used to color $\{v_1, v_2, \ldots, v_i\}$ will be reduced if v_i is given a starting color $r_j \in R$ and the interchange candidate $v_j \in V'$ is assigned its smallest possible starting color $q_j \in Q$. That is,

$$R = \{p_j \in P | q_j + ch(v_j) - 1 < b\}$$

Of course, if R is empty then the interchange failed to find any candidate and the color set (a, b) is assigned to v_i and coloring continues. If R is not empty then there is at least one vertex that can be exchanged with v_i. If more than one candidate is available then the vertex chosen for interchange must be the one to satisfy:

$$\min_{v \in V'}\{\max_{j}\{p_j + ch(v_i) - 1, q_j + ch(v_j) - 1\}\}$$

The above interchange can be reduced to calculating two color-indexed arrays by combining sets P and W and sets Q and R.

Chapter III. describes the interchange method based on the reimplementation. The first two sets P and W will be constructed simultaneously by considering for

each color k, where $1 \leq k < a$, the number of vertices adjacent to v_i whose color sets contain the color k. An array of singleton sets A_1 will be constructed by assigning to $A_1[k]$ the empty set if there is not a unique vertex adjacent to v_i whose color set contains k. If the unique vertex is v_j then $A_1[k]$ is assigned the singleton $\{v_j\}$. This array corresponds to U_1 in the description in Chapter III. The second color-indexed array, A_2, will combine the last two color sets by eliminating the need to construct R by excluding all candidates that do not satisfy the conditions for R. Thus the array A_1 is updated as the array A_2 is constructed. Both arrays represent the set U_2 in the description after the construction of A_2. Another way of seeing this is in terms of P, W, Q, and R. A_1 has the following relationship:

$$A_1[p_j] = \begin{cases} \{v_j\} & p_j \in P \text{ and } v_j \in W \\ \emptyset & \text{otherwise} \end{cases}$$

The array A_2 has this relationship:

$$A_2[p_j] = \begin{cases} q_j & q_j \in R \text{ and } p_j \in P \\ \textit{undefined} & \text{otherwise} \end{cases}$$

The actual implementation using the above suggestions for constructing the color-indexed arrays A_1 and A_2 and determining the best candidate is given in Figure 8. Note that the algorithm calculates the arrays A_1, A_2 and the best color for v_i within the same loop.

2. Timings and Analysis All testing for the interchange algorithm was done on a PC computer with a 80486 microprocessor at 33 MHz clock speed. The testing done by Oakes [Oa90] was done on a 80386 microprocessor at 20 MHz clock speed and a math coprocessor. An adjustment factor of 3.3 will be used to adjust for differences in processors and clock speed. (The 80486 is on an average 2-times faster than the 80386 at the same clock speed and the speedup due to the difference in clocks is 33/20 = 1.65 times faster.) Table VIII shows the timings for the implementation of Oakes on graphs of order 50, 100, 200, 300, 400, and 500 with edge density 0.2. The table

```
Let a = StartingColor(v(i))
If a + ch(v(i)) - 1 > M Then
    For k = 1 To a - 1
        A1[k] = {};
    End For
    MinMax = a
    For k = 1 to a - 1
        For j = 1 To i - 1
            If {v(i), v(j)} in E(Gc) AND ((k >= BegColor(v(j)) AND
               k <= EndColor(v(j))) OR (BegColor(v(j)) >= k AND
               BegColor(v(j)) < k+ch(v(i))-1)) Then
                If A1[k] = {} Then
                    A1[k] = {v(j)}
                Else
                    A1[k] = {}
                    Goto ENDSEARCH:
                End If
            End If
        End For
        ENDSEARCH:
        If A1[k] <> {} Then
            BegColor(v(i)) = k
            A2[k] = StartingColor(A1[k])
            If A2[k] + ch(A1[k]) - 1 >= a Then
                A1[k] = {}
            Else If max(A2[k]+ch(A1[k])-1,k+ch(v(i))-1) < MinMax Then
                MinMax = max(A2[k]+ch(A1[k])-1,k+ch(v(i))-1)
                BestColor = k
            End If
        End If
    End For
    a = BestColor
    BegColor(A1[BestColor]) = A2[BestColor]
End If
BegColor(v(i)) = a
```

Figure 8. Reimplementation of the interchange algorithm

Table VIII. Timing results for graphs with $e = 0.2$

graph order	Timings in seconds						Average
	CLF	CLFI	Increase	CDS	CDSI	Increase	
50	0.10	0.60	6.0	0.19	0.77	4.1	5.1
100	0.37	3.22	8.7	0.67	3.91	5.8	7.3
200	1.38	24.30	17.6	2.47	32.41	13.1	15.4
300	2.89	77.17	26.7	5.30	117.77	22.2	24.5
400	4.99	204.54	41.0	9.31	341.51	36.7	38.9
500	7.64	426.67	55.8	14.42	762.45	52.9	54.5
		Average	26.0		Average	22.5	24.2

Table IX. Timing results for graphs with $e = 0.5$

graph order	Timings in seconds						Average
	CLF	CLFI	Increase	CDS	CDSI	Increase	
50	0.19	3.54	18.6	0.30	3.89	13.0	15.8
100	0.70	27.74	39.6	1.05	31.70	30.2	34.9
200	2.75	268.68	97.7	3.82	430.35	112.7	105.2
300	5.87	1106.63	188.5	8.29	1916.04	231.1	209.8
400	10.46	2946.56	281.7	14.86	5688.99	382.8	332.3
500	14.87	6960.58	468.1	21.49	12987.45	604.3	536.2
		Average	182.4		Average	229.0	205.7

also shows the relative time increases associated with the interchange technique. The relative time increase is the time for the algorithm with interchange divided by the time for the algorithm without interchange. Table IX illustrates the same information for the graphs with edge density 0.5. (Note that CDSatur is abbreviated to CDS within the tables.)

Clearly, this information shows the impact of the interchange method and how inefficient the heuristic becomes in terms of time. Also the more dense graphs consume more time in the interchange relative to the heuristic without the interchange method. Tables X and XI shows the actual and adjusted times using the the reimplementation of the interchange for graphs of order up to 500. The adjusted times are the actual

Table X. Actual and adjusted times for the reimplementation ($e = 0.2$)

graph order	Actual Time (secs)				Adjusted Time (secs)			
	CLF	CLFI	CDS	CDSI	CLF	CLFI	CDS	CDSI
50	0.01	0.03	0.15	0.18	0.03	0.10	0.50	0.59
100	0.05	0.19	1.09	1.34	0.17	0.63	3.60	4.42
200	0.22	1.43	8.24	10.11	0.73	4.72	27.19	33.36
300	0.57	8.87	27.49	33.96	1.88	29.27	90.72	112.07
400	1.19	11.15	65.59	82.83	3.93	36.80	216.45	273.34
500	2.03	22.76	128.50	163.50	6.70	75.11	424.05	539.55

Table XI. Actual and adjusted times for the reimplementation ($e = 0.5$)

graph order	Actual Time (secs)				Adjusted Time (secs)			
	CLF	CLFI	CDS	CDSI	CLF	CLFI	CDS	CDSI
50	0.02	0.09	0.30	0.38	0.07	0.30	1.00	1.25
100	0.07	0.57	2.24	2.98	0.23	1.88	7.39	9.83
200	0.38	4.37	17.61	24.12	1.25	14.42	58.13	79.60
300	1.04	14.73	59.95	83.39	3.43	48.61	197.84	275.19
400	2.18	35.29	144.30	208.00	7.19	116.46	476.19	686.40
500	4.09	71.55	283.90	422.10	13.50	236.12	936.87	1392.93

times multiplied by the adjustment factor of 3.3.

The times associated with CDSatur without interchange seem strange since the times are greater than that of Oakes but the core algorithm was implemented differently than in Oakes — the differences will be explained later in this chapter along with suggestions on how the differences can be eliminated.

Since the interchange is so important to the vertex sequential algorithms, the speedups make the interchange more appealing for use. The time required to color graphs with $e = 0.5$ is 10% of the time found by Oakes. For example, in Table IX the entry for CDSaturI with 500 vertices has a time of 12987.45 seconds while the adjusted time for CDSaturI with 500 vertices in Table XI is 1392.93 seconds. This results in an 89% reduction in runtime. Further, the average increase in time consumption

Table XII. Interchange for graphs $(1000, *, \text{TPOI}(1), 25)$

edge	Time (secs)			Time (secs)		
density	CLF	CLFI	increase	CDSatur	CDSaturI	increase
0.2	11.36	152.30	13.4	827.67	1121.00	1.4
0.5	32.67	696.70	21.3	2351.41	4294.00	1.8

for all graphs with $e = 0.2$ is 4.7, down from 24.2, and when $e = 0.5$ the average increase is found to be 6.7, down from 205.7. As a result much larger graphs can be colored. Table XII shows the time statistics for graphs of order 1000. From this table, we observe that the increase in run time for the CDSatur algorithm was less than two times. As we will see later, the interchange method does not improve the colorings on the less dense graphs by much in spite of the extra effort. In any case, the reimplementation of the interchange method may prove useful in the practice of coloring composite graphs.

C. EXPERIMENTATION AND RESULTS

This section provides further testing and analysis over that provided by Roberts [Ro87] and Oakes [Oa90]. The interchange method is used in conjunction with both the CLF and CDSatur algorithms on graphs larger than 500 and some interesting observations are made. We also establish the empirical evidence for the theorems developed in the next chapter.

1. Related Studies Roberts implemented 12 heuristics and Oakes implemented 10 heuristics for composite graph coloring. Of those studies only a few of the algorithms stand out as superior. Roberts study used LF1, LF2, LFPH, LFCD, DYNPH, DYNFPH, LF1I, LF2I, LFPHI, LFCDI, RLF1, and RLFD1. Roberts test data consisted of 25 instances of random composite graphs with 100 vertices and edge densities of 0.10, 0.15, 0.20, 0.30, 0.40, and 0.50. In addition, graphs with orders 200, 300, 400, and 500 were tested with edge densities 0.10, 0.15, and 0.20. The chromaticities were generated using five different distributions, including the TPOI(1)

Table XIII. Coloring results from Roberts for graphs $(*, 0.2, \text{TPOI}(1), 25)$

graph order	average colors used				
	LF1I	LFPHI	LFCDI	RLF1	RLFD1
100	15.7	15.5	15.7	15.9	16.0
200	23.6	23.8	23.9	23.4	23.6
300	31.8	31.6	31.7	30.7	30.7
400	39.4	39.2	39.2	37.8	37.8
500	46.2	46.2	46.3	44.3	44.4

distribution suggested by Clementson and Elphick. Oakes study used CLF, CSL, CDSatur, CRLF, CLFI, CSLI, CDSaturI, and three vertex sequential algorithms with a different interchange technique: CLFI2, CSLI2, and CDSaturI2. Sets of 25 instances were constructed of orders 50, 100, 200, 300, 400, and 500 with edge densities 0.2 and 0.5. All algorithms were tested on these sets. Additionally, the CRLF, CDSatur, CLF, and CSL were tested on orders from 500 to 1000 (by 50) for edge densities of 0.2 and 0.5. Only TPOI(1) was used to generate the chromaticities for these graphs.

Roberts concluded that five of the twelve algorithms consistently provided a smaller average number of colors than all of the others. These are: LF1I, LFPHI, LFCDI, RLF1, and RLFD1. Table XIII shows the results reported by Roberts for the average number of colors used by graphs of order 100 to 500 and edge density 0.2. Table XIV provides the number of "wins" for each of the algorithms on the same set of graphs. An algorithm produces a win by coloring an instance of a random composite graph with at least as few colors as all of the other algorithms.

Oakes decided to implement only one representative from the vertex sequential with interchange and the color sequential algorithms. Thus, LF1I, LFPHI, and LFCDI would be represented with CLFI while RLF1 and RLFD1 would be represented with CRLF. In addition, Oakes also implemented two vertex sequential with and without two interchange techniques: CSL and CDSatur. Table XV shows the results for coloring graphs with edge density 0.2 while Table XVI shows the same results for graphs with edge density 0.5. Figures 9 and 10 show the number of wins for the

Table XIV. Number of Wins reported by Roberts $(*, 0.2, \text{TPOI}(1), 25)$

graph order	number of wins				
	LF1I	LFPHI	LFCDI	RLF1	RLFD1
100	12	16	13	9	8
200	12	7	9	16	15
300	3	4	4	20	20
400	1	2	2	19	21
500	0	0	0	20	19

algorithms for the color sequential heuristic and the vertex sequential heuristics with interchange. From this information Oakes concluded the following:

1. The interchange method of Clementson and Elphick (called I1) worked equally well with the CLF and CDSatur algorithms for small graphs (order 50), producing better results than the other heuristics.

2. CDSaturI1 produced the best results for graphs of order 100.

3. CRLF and CDSaturI1 dominated the other heuristics for graphs of order above 100.

4. The other tested interchange method (called I2) performed poorly when compared to the method of Clementson and Elphick.

5. The time increased significantly for the heuristics when augmented with interchange.

6. CDSatur is more competitive with CRLF when time is a consideration although the average number of colors required is more for larger graphs.

An observation made by both Oakes and Roberts was the apparent dominance of the vertex sequential algorithms for graphs of small order and the color sequential algorithms' dominance for graphs of large order (see Figure 9 and Table XIV). This

Table XV. Coloring results from Oakes for graphs (*, 0.2, TPOI(1), 25)

graph order	average colors used									
	CLF	CSL	CDS	CRLF	CLFI1	CSLI1	CDSI1	CLFI2	CSLI2	CDSI2
50	11.2	11.8	10.8	10.7	10.4	10.8	10.4	10.9	11.4	10.6
100	17.3	18.2	16.1	15.8	15.8	16.2	15.2	16.4	17.5	15.9
200	26.9	28.2	25.2	24.9	25.0	25.6	24.4	26.0	27.4	25.0
300	34.3	35.2	31.3	31.0	31.9	32.6	30.5	33.0	33.8	31.3
400	41.5	43.0	38.5	37.7	39.6	40.2	38.1	40.1	41.4	38.6
500	49.2	50.0	45.1	44.1	46.0	47.0	44.4	47.0	48.2	45.3

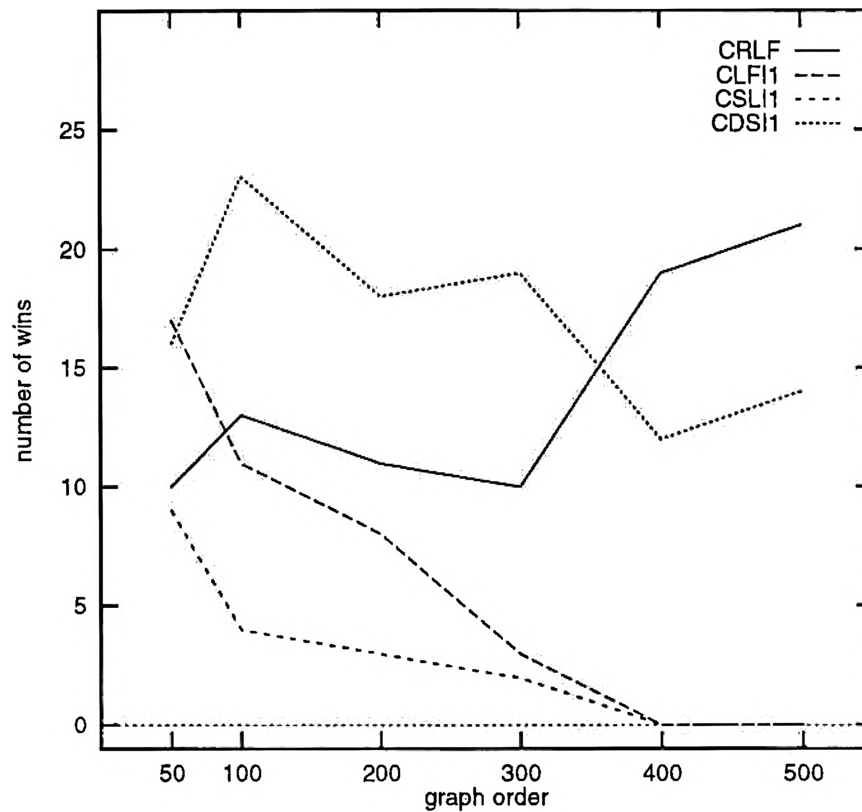


Figure 9. Number of wins from Oakes for graphs (*, 0.2, TPOI(1), 25)

Table XVI. Coloring results from Oakes for graphs $(*, 0.5, \mathrm{TPOI}(1), 25)$

graph	average colors used									
order	CLF	CSL	CDS	CRLF	CLFI1	CSLI1	CDSI1	CLFI2	CSLI2	CDSI2
50	20.4	20.8	19.3	19.4	18.7	19.1	18.4	19.4	20.0	18.9
100	33.3	33.9	31.1	31.5	30.8	30.8	30.2	31.9	32.5	30.8
200	56.7	57.0	53.0	53.0	52.6	53.7	51.8	54.1	54.6	52.0
300	74.6	75.6	70.0	69.5	70.5	70.7	68.4	70.9	72.4	69.1
400	93.8	94.9	88.5	86.9	87.9	89.3	86.8	89.4	90.4	88.0
500	110.7	112.4	105.4	103.2	104.4	105.0	103.0	106.2	107.4	103.8

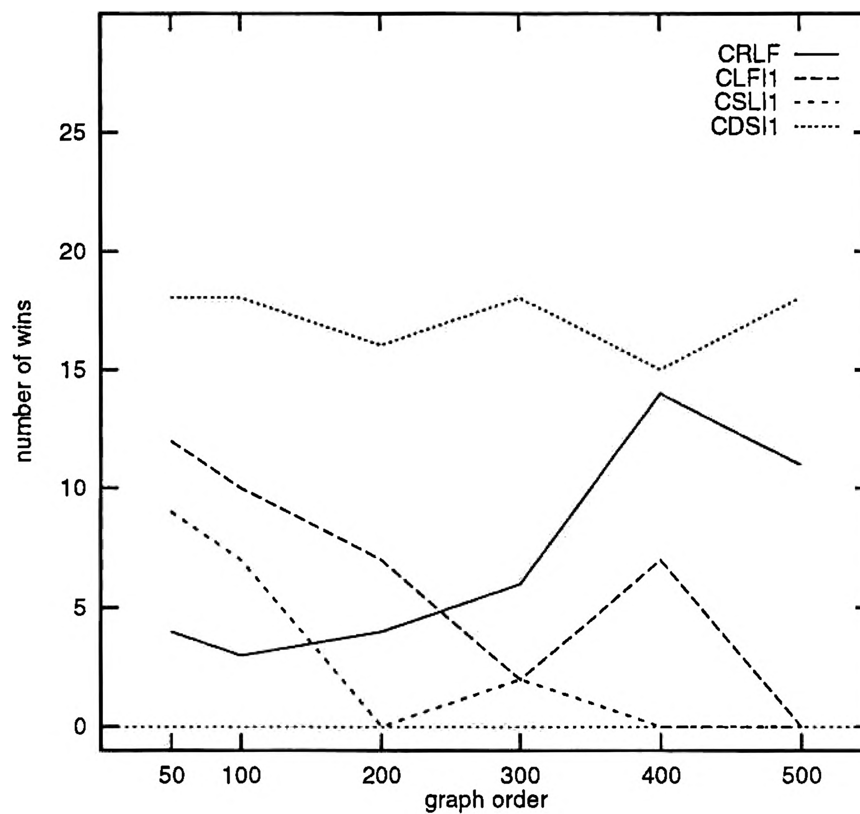


Figure 10. Number of wins from Oakes for graphs $(*, 0.5, \mathrm{TPOI}(1), 25)$

situation is not as clear for graphs of density 0.5 as is depicted in Figure 10. The CDSaturI1 algorithm dominates the other algorithms for all orders.

2. Testing and Analysis The two previous studies provided insight into the use of heuristics but fail in coloring graphs up to 1000 vertices for the choice heuristics (Oakes used the interchange methods only up to graphs of order 500). The time barrier removed, this study will use the interchange technique to analyse graphs of orders 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 with edge densities of 0.2 and 0.5 and chromaticities generated using TPOI(1) random variables. As mentioned, these data sets are those used by Oakes so that direct comparisons can be made.

Because of the observations made previously we justified a reduction in the algorithms analyzed. This study will consider only the CLF, CLFI, CDSatur, CDSaturI, and CRLF. These choices were made because only the representatives from the statically ordered vertex sequential, dynamically ordered vertex sequential, and color sequential algorithms that provided the best performance should be considered. The I2 interchange method will not be considered since it did not compete well with the I1 interchange method. Table XVII provides the average number of colors examined for graphs with an edge density of 0.2. Because the same data sets as that of Oakes were used, the observed averages for the graphs with order 50 to 500 are the same. Table XVIII is for the graphs with edge density of 0.5. Figures 12 and 14 provides the same information graphically. From this information we can see that CLF does not compete with the other heuristics but the interchange method with CLF produces good results for small graphs—actually competing with CDSaturI. The clear winners in the heuristics are CDSaturI and CRLF. Figure 11 shows more clearly the relationship between CDSaturI and CRLF for graphs with edge density of 0.2 and Figure 13 for graphs with edge density of 0.5.

The following were the observations made from this study:

1. The CLF heuristic produces colorings that on the average require a larger number of colors than do the other heuristics. On the other hand, the CLF heuristic

Table XVII. Coloring results for graphs $(*, 0.2, \mathrm{TPOI}(1), 25)$

graph order	average colors used				
	CLF	CLFI	CDS	CDSI	CRLF
50	11.2	10.4	10.8	10.4	10.7
100	17.3	15.8	16.1	15.2	15.8
200	26.9	25.0	25.2	24.4	24.9
300	34.3	31.9	31.3	30.5	31.0
400	41.5	39.6	38.5	38.1	37.7
500	49.2	46.0	45.1	44.4	44.1
600	56.0	52.5	51.3	51.6	50.7
700	62.5	59.2	58.3	57.6	56.7
800	70.4	66.2	65.4	64.5	63.7
900	76.3	71.8	70.3	70.2	68.8
1000	82.5	77.8	76.2	76.3	74.5

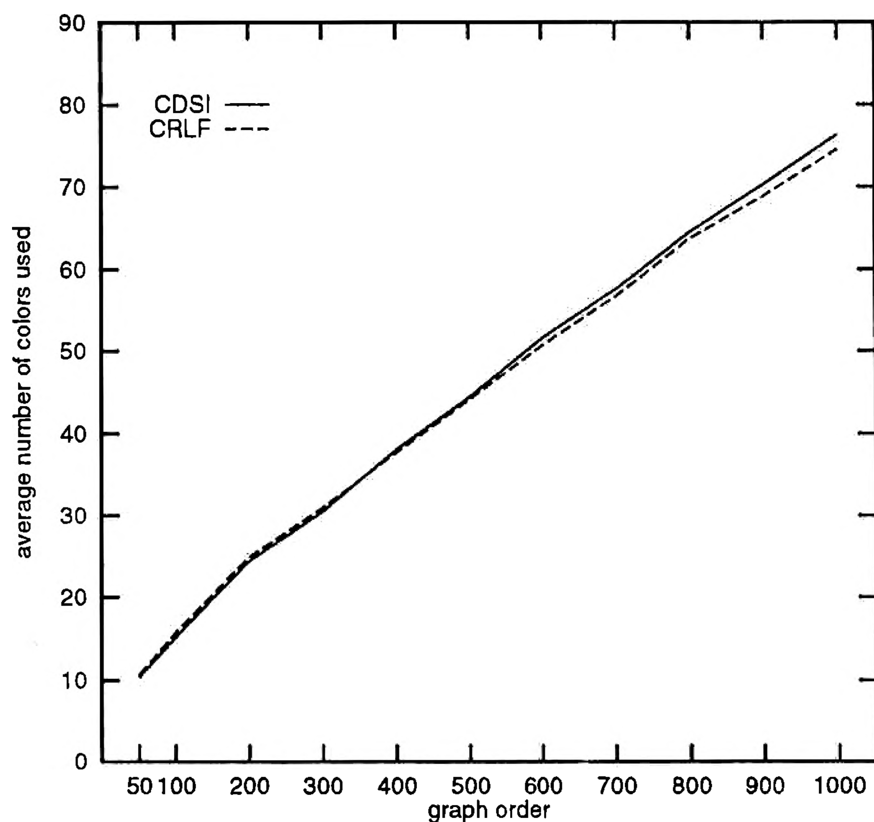


Figure 11. Coloring results for CDSaturI and CRLF $(*, 0.2, \mathrm{TPOI}(1), 25)$

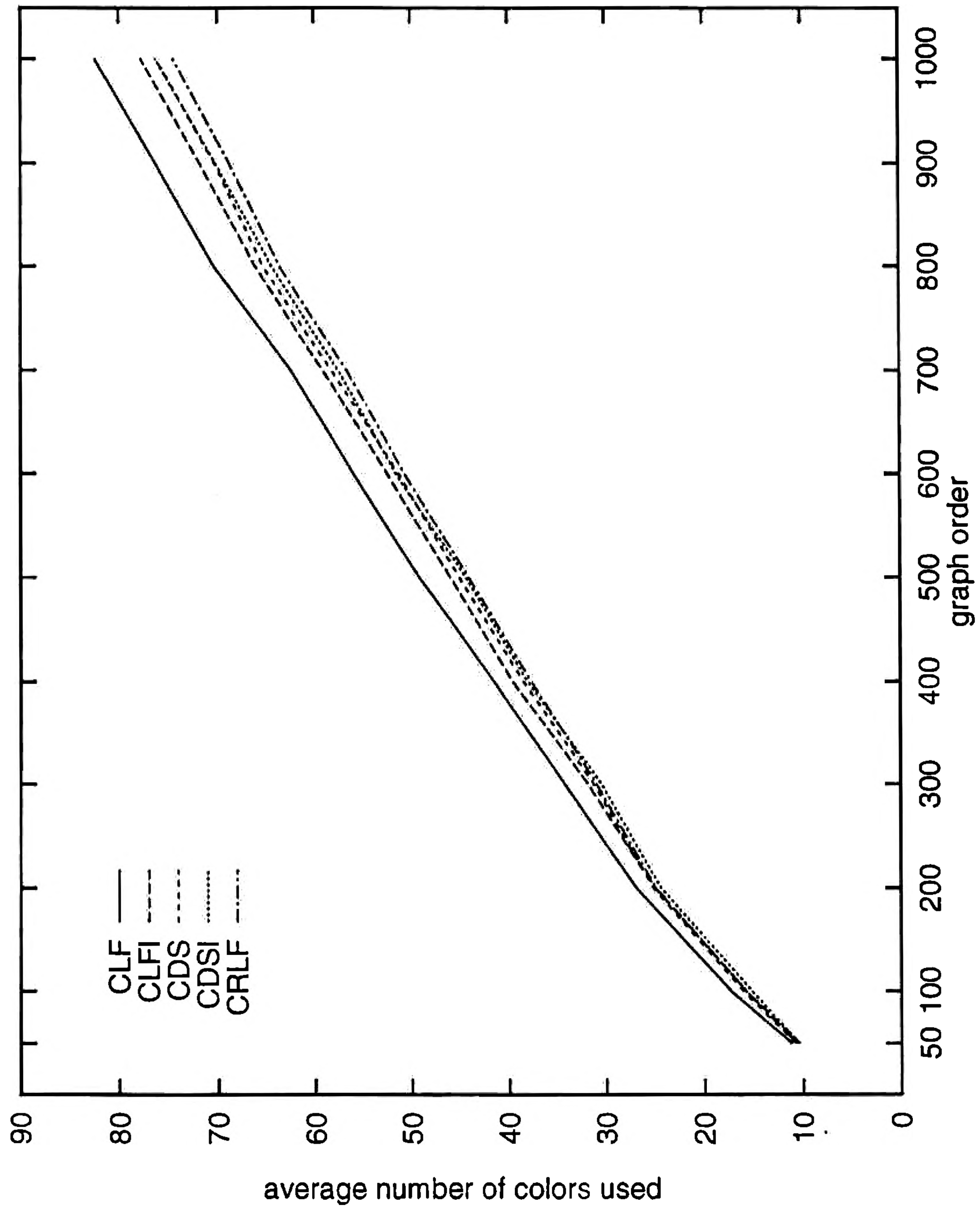


Figure 12. Coloring results for graphs $(*, 0.2, \mathrm{TPOI}(1), 25)$

Table XVIII. Coloring results for graphs $(*, 0.5, \text{TPOI}(1), 25)$

graph order	average colors used				
	CLF	CLFI	CDS	CDSI	CRLF
50	20.4	18.7	19.3	18.4	19.4
100	33.3	30.8	31.1	30.2	31.5
200	56.7	52.6	53.0	51.8	53.0
300	74.6	70.5	70.0	68.4	68.5
400	93.8	87.9	88.5	86.8	86.9
500	110.7	104.4	105.4	103.0	103.2
600	128.8	122.4	121.4	121.2	120.0
700	145.7	139.1	138.4	137.1	136.2
800	163.9	155.9	156.5	154.4	152.6
900	178.2	171.4	171.0	170.3	167.1
1000	195.7	187.0	186.2	184.9	182.5

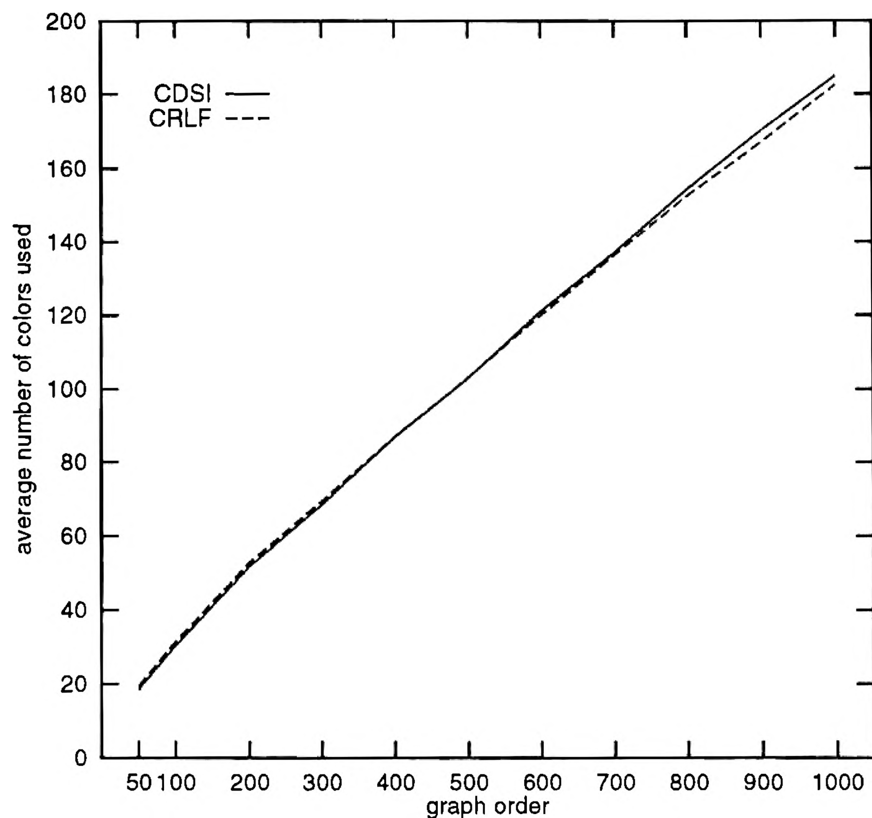


Figure 13. Coloring results for CDSaturI and CRLF $(*, 0.5, \text{TPOI}(1), 25)$

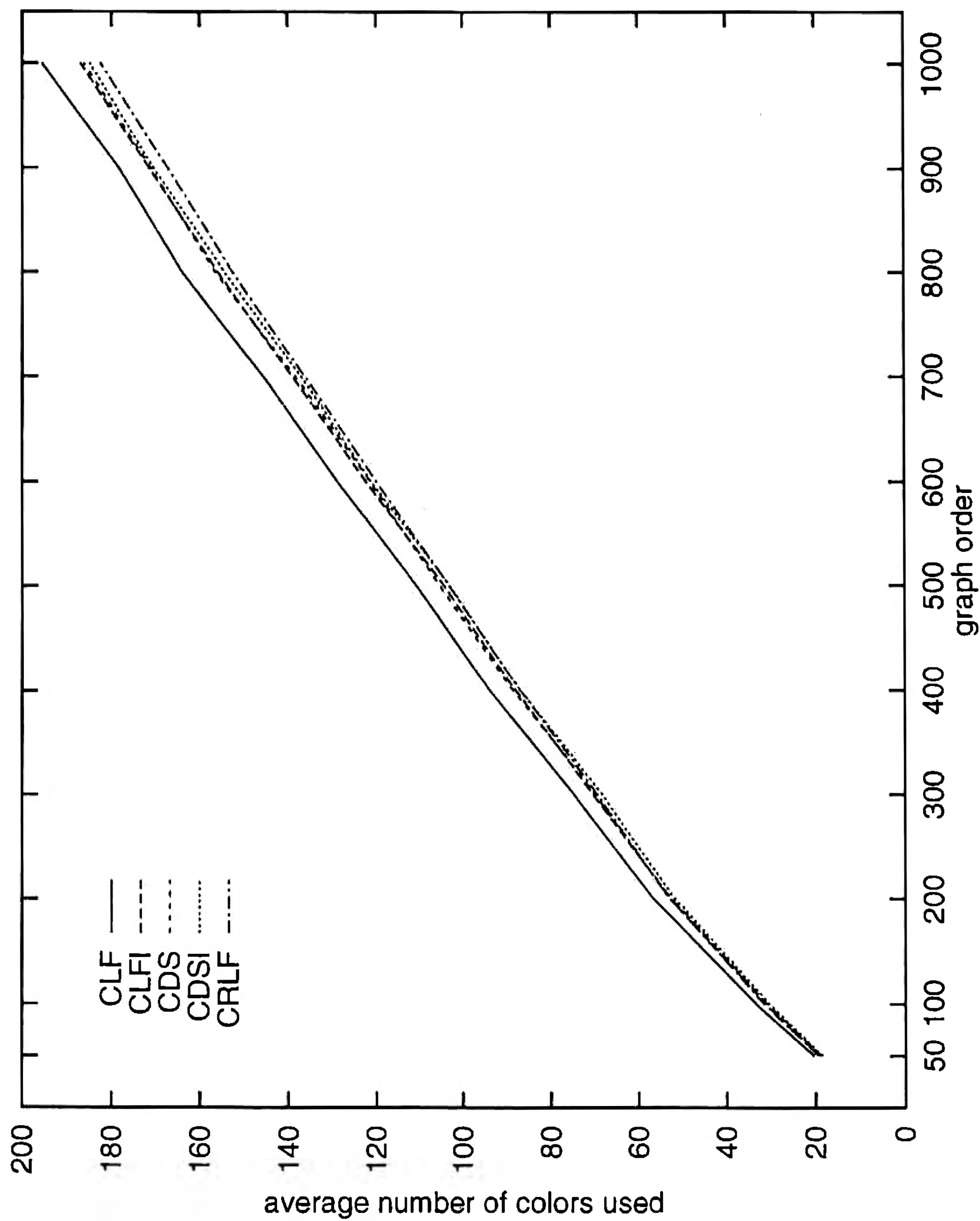


Figure 14. Coloring results for graphs $(*, 0.5, \mathrm{TPOI}(1), 25)$

was by far the fastest algorithm. CLF requires on the average 32.67 seconds for graphs with 1000 vertices and edge density of 0.5. The closest algorithm to that was CLFI which required 696.70 seconds, a factor of 21.3 (see Table XII).

2. The CDSaturI heuristic was superior to the other heuristics for "small" graphs. Oakes was able to exhibit this for graphs with edge densities of 0.2 and 0.5. The actual size of the graph is dependent on the density of the graph being colored. For graphs with $e = 0.2$, the order of the graph being colored should be less than 350 (see Figure 11). From Oakes study this could not be determined for graphs with $e = 0.5$ because the CDSaturI performed best for all orders. Figure 13 shows that the order of the graph should be less than 500.

3. The interchange method applied to CDSatur heuristic had mixed results for graphs over 500 with edge density 0.2. The reimplementation of the interchange consumed more time but did not perform up to expectations obtained on graphs with 0.5 edge density. Figure 15 shows the number of wins for graphs with 0.2 edge density as the order increases. This would indicate that CDSatur would be provide similar results to the CDSaturI while being more efficient.

4. The CRLF heuristic was superior to the other heuristics for "large" graphs. Again the size of the graph is dependent on the edge density. This role reversal of vertex sequential and color sequential algorithms on performance was a phenomenon that was observed by Oakes and Roberts for graphs with density 0.2. Clearly this is also the case for graphs of density 0.5 but the point of crossover was higher for higher order graphs. Figures 16 and 17 illustrate the wins obtained by each of the heuristics for all of the orders tested.

5. With the reimplementation of the interchange method of Clementson and Elphick, the times for CRLF and CDSaturI were comparable for large graphs. The average times for the CRLF algorithm for graphs with 0.2 and 0.5 edge densities were 800.7 and 1049.0 seconds, respectively. The CDSaturI algorithm

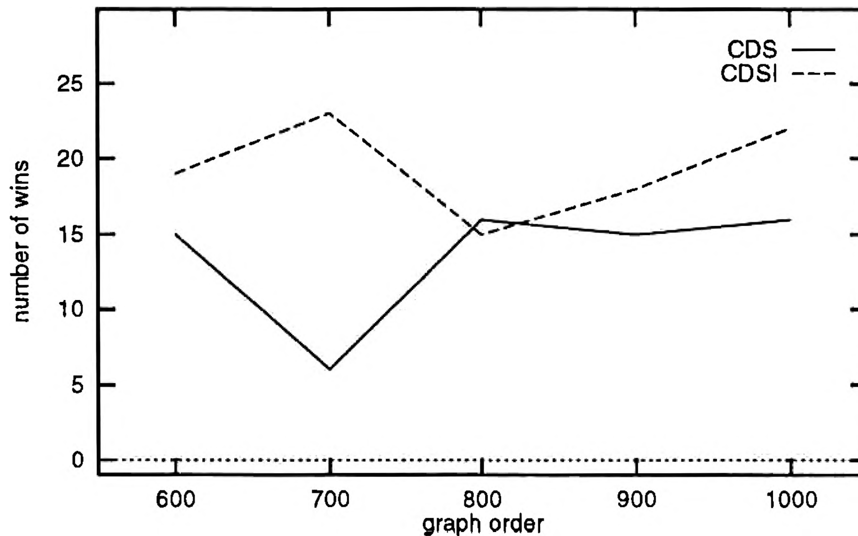


Figure 15. Comparing wins for CDS and CDSI heuristics $(*, 0.2, \mathrm{TPOI}(1), 25)$

for the same set of graphs were 1121.0 and 4294.0 seconds, respectively. This only reflects a 140% increase when $e = 0.2$ and a 409% increase when $e = 0.5$.

Before ending this section some more analysis is in order. The purpose of this analysis is to support the theoretical finding of the next chapter. As we will see, CGCP is $\mathcal{NP}$-hard and that approximation is as difficult as finding exact solutions to the problem if we expect guaranteed upper bounds on the approximations. This analysis is based on probabilistic bounds for graphs with chromaticities generated with TPOI(1) random variables. Oakes [Oa90] found upper and lower probabilistic bounds for these graphs and suggested that these bounds could be used as "confidence intervals" for use in the study of heuristics. The lower bound, midpoint of the interval, and the upper bounds for the graphs in this study are found in Tables XIX and XX. These bounds will be used as a means of measuring the error in approximating the chromatic number for the graph using the best heuristic estimate for the algorithms in this study.

The error estimates for this study were calculated as follows. The first error

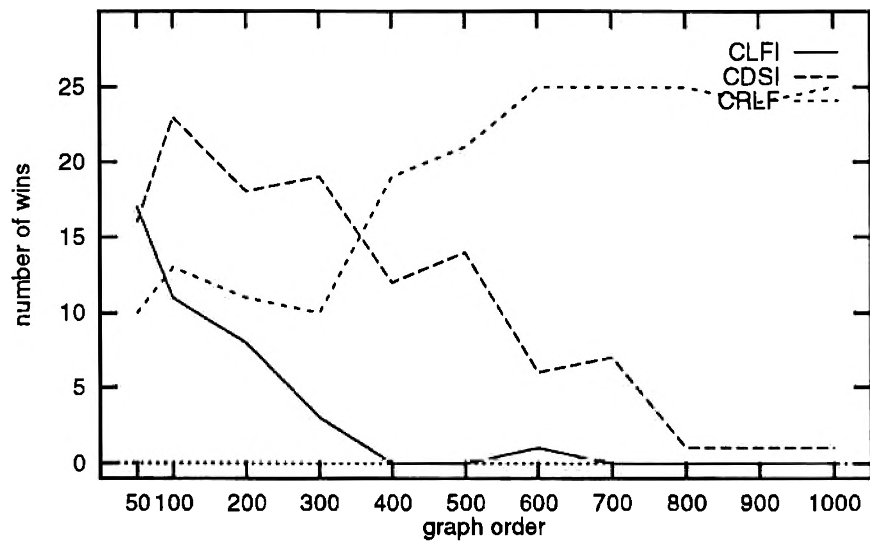


Figure 16. Number of wins for heuristics $(*, 0.2, \mathrm{TPOI}(1), 25)$

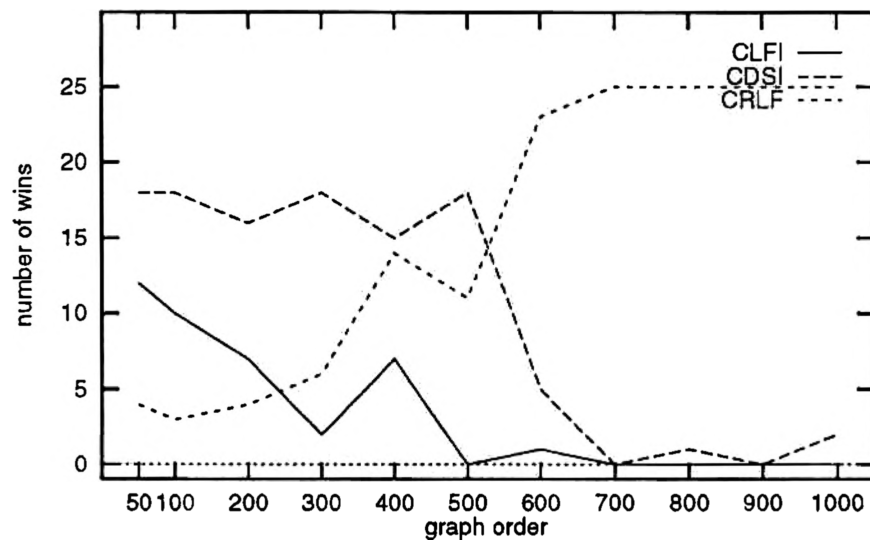


Figure 17. Number of wins for heuristics $(*, 0.5, \mathrm{TPOI}(1), 25)$

Table XIX. Probabilistic bounds for graphs with $e = 0.2$

graph order	lower bound	mid point	upper bound
50	7.0	8.8	10.5
100	10.0	12.5	15.0
200	17.0	19.7	22.4
300	22.0	25.6	29.1
400	28.0	31.6	35.2
500	32.0	36.5	41.0
600	37.0	41.8	46.5
700	42.0	47.0	52.0
800	46.0	51.7	57.3
900	47.0	54.7	62.3
1000	50.0	58.7	67.4

Table XX. Probabilistic bounds for graphs with $e = 0.5$

graph order	lower bound	mid point	upper bound
50	14.0	16.3	18.6
100	22.0	25.7	29.3
200	38.0	42.8	47.5
300	51.0	57.6	64.1
400	64.0	71.8	79.5
500	76.0	85.2	94.3
600	88.0	98.2	108.4
700	101.0	111.9	122.7
800	112.0	124.1	136.1
900	122.0	135.6	149.2
1000	133.0	147.7	162.3

estimate was absolute error:

$$\varepsilon = \chi^* - \frac{UB + LB}{2}$$

where χ^* is the estimate provided by the heuristic for the chromatic number of the graph, UB and LB are the upper and lower bounds, respectively. The average absolute error estimate for each set of graphs is given in Table XXI for graphs with $e = 0.2$ and Table XXII for $e = 0.5$. Figures 18 and 19 correspond to the smallest average absolute error estimate for each set of graphs colored over all heuristics.

A second error estimate, the relative deviation from the upper bound is given by:

$$\delta = \frac{\chi^* - UB}{UB}$$

where χ^* is the estimate for the chromatic number of the graph and UB is the upper bound of the confidence interval. Tables XXIII and XXIV provide the average values for the relative deviation from the midpoint for graphs with edge density 0.2 and 0.5. Figures 20 and 21 illustrate the smallest average deviation for each set of graphs colored over all heuristics.

These findings support the theory of the next chapter. One theorem states that there exists an ϵ-absolute approximation algorithm for CGCP only if CGCP$\in \mathcal{P}$. This theorem states that there is no fixed ϵ so that the approximation provided by the heuristic is within ϵ of the chromatic number of the graph. Clearly, the trend in both of the Figures 18 and 19 seems to indicate this. A stronger result in the next chapter states that there exists an ϵ-relative approximation algorithm for CGCP with $\epsilon < 1$ only if CGCP$\in \mathcal{P}$. Another way of stating this is that no heuristic can guarantee that the estimate is within $2\chi(G_c)$ for all instances of CGCP. There will be at least one graph that will have an estimate larger than $2\chi(G_c)$. Although this is clearly not the case for some subsets of CGCP, the set of all complete graphs for example, it is not as clear what the implications are for TPOI(1)-type graphs. In general, we would not expect this average to be much less than 2 since there are

Table XXI. Absolute error estimate in heuristics (*.0.2, TPOI(1), 25)

graph order	average absolute error				
	CLF	CLFI	CDS	CDSI	CRLF
50	2.5	1.7	2.1	1.7	2.0
100	4.8	3.3	3.6	2.7	3.3
200	7.2	5.3	5.5	4.7	5.2
300	8.7	6.3	5.7	5.0	5.5
400	9.9	8.0	6.9	6.5	6.1
500	12.7	9.5	8.6	7.9	7.6
600	14.2	10.8	9.5	9.9	8.9
700	15.5	12.2	11.3	10.6	9.7
800	18.8	14.6	13.8	12.9	12.1
900	21.6	17.2	15.6	15.6	14.1
1000	23.8	19.1	17.5	17.6	15.8

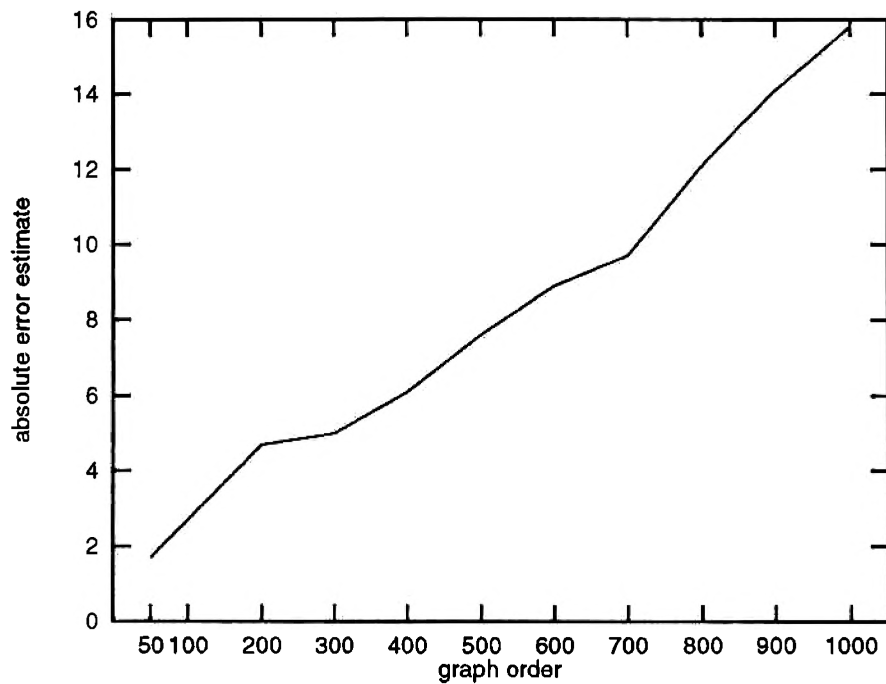


Figure 18. Absolute error estimate in heuristics (*, 0.2, TPOI(1), 25)

Table XXII. Absolute error estimate in heuristics (*, 0.5, TPOI(1), 25)

graph order	average absolute error				
	CLF	CLFI	CDS	CDSI	CRLF
50	4.1	2.4	3.0	2.1	3.1
100	7.6	5.2	5.5	4.6	5.9
200	13.9	9.9	10.3	9.1	10.3
300	17.2	13.0	12.5	10.9	11.9
400	22.1	16.2	16.8	15.1	15.2
500	25.5	19.3	20.2	17.8	18.0
600	30.6	24.2	23.2	23.0	21.8
700	33.9	27.3	26.6	25.3	24.4
800	39.9	31.9	32.5	30.4	28.6
900	42.6	35.8	35.4	34.7	31.5
1000	48.0	39.4	38.5	37.3	34.8

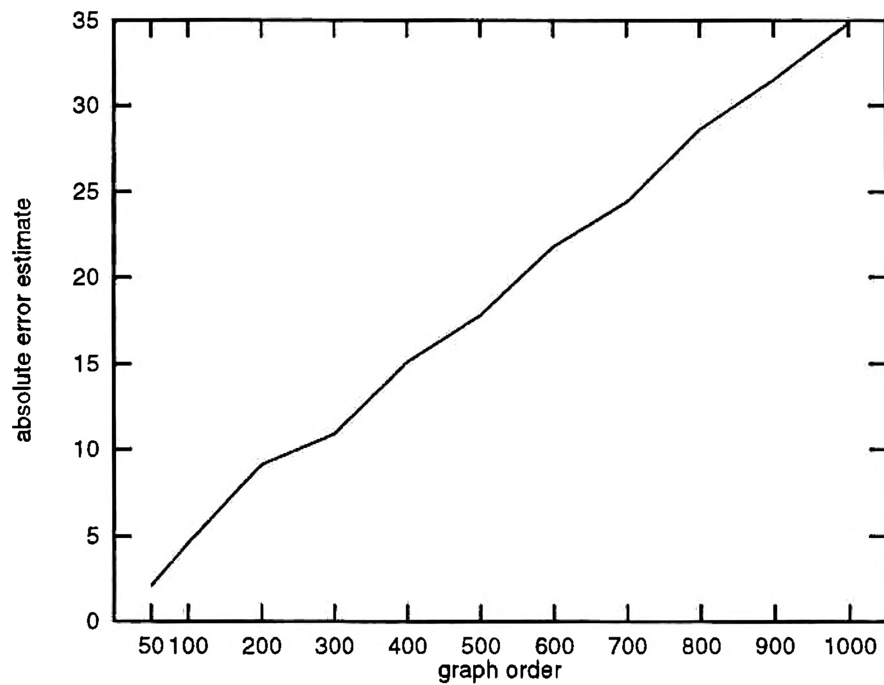


Figure 19. Absolute error estimate in heuristics (*, 0.5, TPOI(1), 25)

Table XXIII. Relative deviation from the upper bound (*.0.2, TPOI(1), 25)

graph order	average relative deviation				
	CLF	CLFI	CDS	CDSI	CRLF
50	0.07	-0.01	0.03	0.00	0.02
100	0.15	0.05	0.07	0.02	0.05
200	0.20	0.11	0.13	0.09	0.11
300	0.18	0.10	0.08	0.05	0.07
400	0.18	0.13	0.09	0.08	0.07
500	0.20	0.12	0.10	0.08	0.08
600	0.20	0.13	0.10	0.11	0.09
700	0.20	0.14	0.12	0.11	0.09
800	0.23	0.16	0.14	0.13	0.11
900	0.22	0.15	0.13	0.13	0.10
1000	0.22	0.15	0.13	0.13	0.11

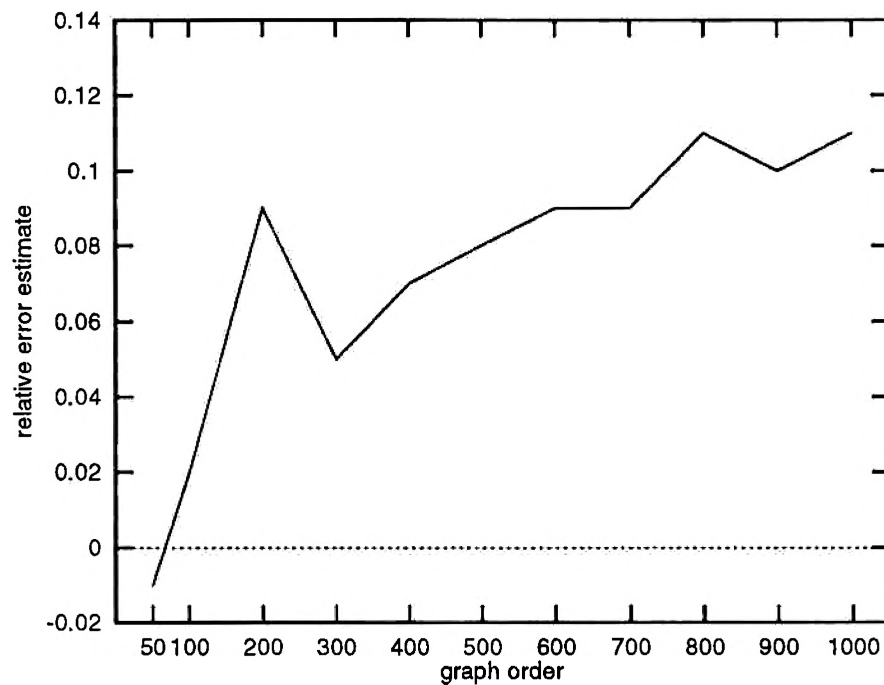


Figure 20. Relative deviation from the upper bound (*, 0.2, TPOI(1), 25)

Table XXIV. Relative deviation from the upper bound $(*, 0.5, \text{TPOI}(1), 25)$

graph order	average relative deviation				
	CLF	CLFI	CDS	CDSI	CRLF
50	0.10	0.00	0.04	-0.01	0.04
100	0.30	0.05	0.06	0.03	0.08
200	0.33	0.11	0.12	0.09	0.12
300	0.30	0.10	0.09	0.07	0.08
400	0.31	0.11	0.11	0.09	0.09
500	0.30	0.11	0.12	0.09	0.09
600	0.31	0.13	0.12	0.12	0.11
700	0.30	0.13	0.13	0.12	0.11
800	0.32	0.15	0.15	0.13	0.12
900	0.31	0.15	0.15	0.14	0.12
1000	0.33	0.15	0.15	0.14	0.12

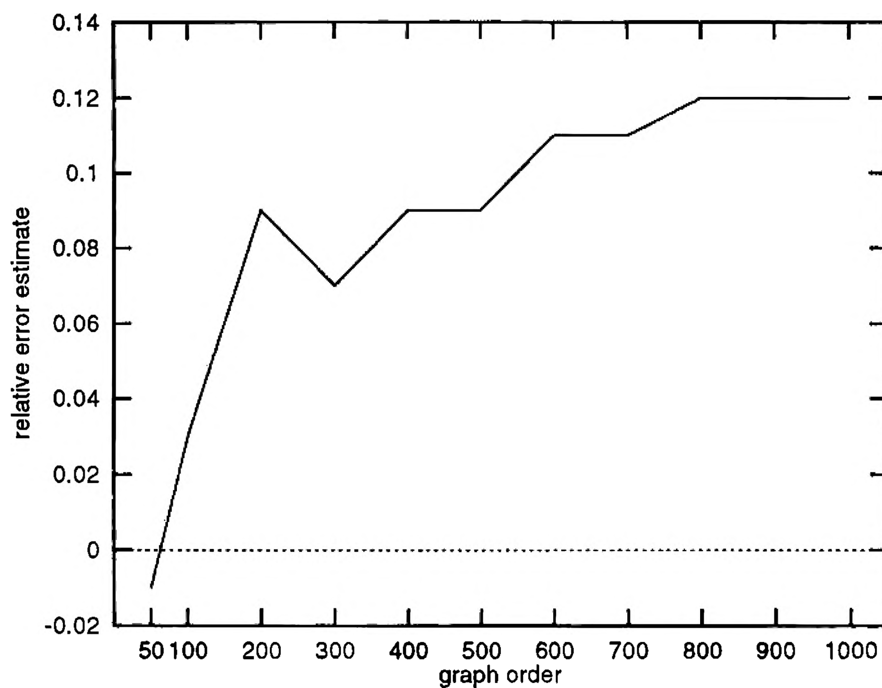


Figure 21. Relative deviation from the upper bound $(*, 0.5, \text{TPOI}(1), 25)$

many graphs of one order that would be easier to color. Figures 20 and 21 are an attempt to measure this error in the heuristics.

D. IMPLEMENTATION SPECIFICS

Oakes implementation of CDSatur performed with times that were much shorter than the implementation presented here. This discrepancy can be explained in the difference in data structures used to represent the graph. Oakes used an adjacency list as the underlying representation while the implementations in this study used the adjacency matrix as the data structure. Both have different run-time characteristics depending on the operations that the algorithms perform. The basic function of these heuristics is determining adjacency of vertices since this will in fact be used in choosing the colors. Given two vertices, determining adjacency with an adjacency matrix requires $O(1)$ operations but the use of the adjacency list will require scanning the adjacency list of one of the vertices in question. This on the average will require $O(en)$ operations, where e is the edge density and n is the order. But the real difference in run-time behavior occurs in the coloring function in the vertex sequential algorithms. This function, given a vertex, will attempt to find the smallest feasible color set and assign this set to the vertex. In the process the function will need to know all vertices adjacent to the vertex. With an adjacency list, the adjacent vertices are already known and the coloring algorithm will on the average use $O(en)$ operations when accessing this list. On the other hand, the use of the adjacency matrix will require scanning the row of the vertex for which adjacency is needed; this will require $O(n)$ operations. If both data structures were implemented then the times for both the CDSatur and CDSaturI would see a decrease in run-time. The storage requirement for each graph would be no more than double the requirement for any single data structure. Where time is the main consideration then both of the data structures should be used to represent the abstract graph. With these recommendations, the expected time requirement for coloring vertices should be on the average $100e\%$ that of the time for the algorithms presented here (e is the edge density of the graph).

V. THEORETICAL RESULTS

This chapter explores theoretical results for CGCP based on SGCP. There are results to show that the decision problem for CGCP is $\mathcal{NP}$-complete and that CGCP is $\mathcal{NP}$-hard. Further results are developed to show the complexity of approximating solutions to CGCP. All results are based on the definition in Section III.A.

A. THE DECISION PROBLEM IS $\mathcal{NP}$

The associated decision problem for the SCGP is known to be $\mathcal{NP}$-complete [Ka72]. In Section II.E.3. we illustrated that SGCP was $\mathcal{NP}$-hard based on this result. The CGCP will also be shown to be $\mathcal{NP}$-hard. We begin by showing that the decision problem is in $\mathcal{NP}$.

Theorem 5.1 The decision problem associated with the Composite Graph Coloring problem is a member of the $\mathcal{NP}$ set of problems.

Proof In order to show that the decision problem for CGCP is in $\mathcal{NP}$, we must first describe the certificate of verification. With most CO problems, the associated decision problem can be described as the problem of feasibility. For any instance $(\mathsf{G}_c, \mathsf{K})$ of the CGCP, a feasible solution is any one of the coloring functions from K. Thus given a coloring, constructing an algorithm of polynomial complexity that would verify the coloring would be sufficient to show that the associated decision problem is in $\mathcal{NP}$. The algorithm is shown in Figure 22. This algorithm can be shown to have complexity $O(|\mathsf{V}(\mathsf{G}_c)|^2)$. It should be noted that the binary representation of each instance of CGCP is also on the order of $|\mathsf{V}(\mathsf{G}_c)|^2$. ∎

A `C` language version of this algorithm (function `VerifyColors` in file **`graph.c`**) is in the Appendix.

B. COMPOSITE GRAPH COLORING IS $\mathcal{NP}$-HARD

Since we have shown that the decision problem of CGCP is in $\mathcal{NP}$, we proceed to show that it is also $\mathcal{NP}$-complete. To do so, we will show that each instance of the

```
Let Valid_Coloring = TRUE
For i = 1 To sizeof(V(G))
    For j = i+1 To sizeof(V(G))
        If {v(i), v(j)} in E(G) Then
            If BegColor(v(i) >= BegColor(v(j)) AND
               BegColor(v(i)) <= EndColor(v(j)) Then
                Let Valid_Coloring = FALSE
            End If
            If BegColor(v(j) >= BegColor(v(i)) AND
               BegColor(v(j)) <= EndColor(v(i)) Then
                Let Valid_Coloring = FALSE
            End If
        End If
    End For
End For
Return Valid_Coloring
```

Figure 22. Algorithm for validating a coloring

SGCP is polynomially reducible to an instance of the CGCP.

Theorem 5.2 The decision problem associated with the Composite Graph Coloring problem is a member of the $\mathcal{NP}$-complete set of problems.

The function used in the reduction is defined for each instance of the SGCP (G, K) by constructing and instance of the CGCP $(\mathsf{G}_c, \mathsf{K}')$. G_c is defined as the tuple (G', ch) where G' is a simple graph with vertex set $\mathsf{V}(\mathsf{G}) \cup \{v'\}$ and edge set $\mathsf{E}(\mathsf{G}) \cup \{\{v, v'\} | v \in \mathsf{V}(\mathsf{G})\}$. The vertex v' is different than any vertex in $\mathsf{V}(\mathsf{G})$. The chromaticity function ch is defined as:

$$ch(v) = \begin{cases} 1 & \text{if } v \in \mathsf{V}(\mathsf{G}) \\ 2 & \text{if } v = v' \end{cases}$$

Figure 23 depicts this for an instance of the SGCP.

The graph G_c satifies the requirements of Section III.A. since the vertex v' has a chromaticity of 2.

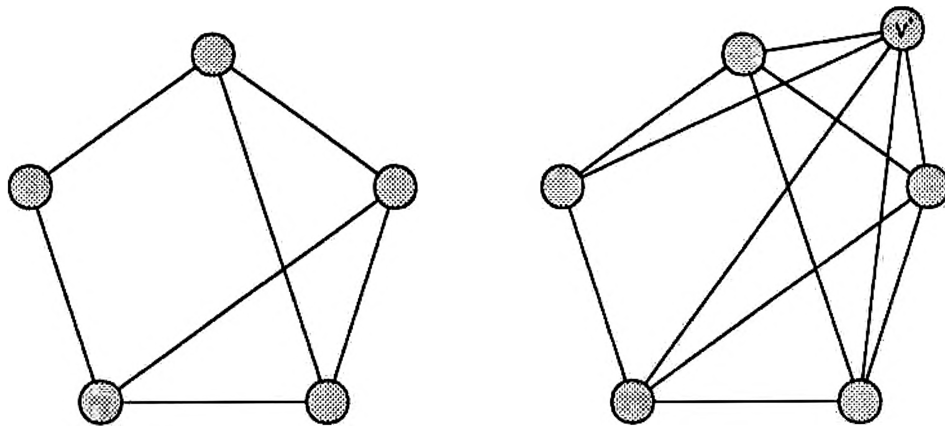


Figure 23. Composite Graph constructed using a Standard Graph

Proof To show that the decision problem for the SGCP is polynomially reducible to decision problems of the CGCP, we must show that the above function, f, meets the two conditions stated in Definition 2.11.

The first condition requires that the function be computed in polynomial time, that is, given an instance of the SGCP (G, K), the transformation is of the complexity $O(p(|\mathsf{E}(\mathsf{G})| + |\mathsf{V}(\mathsf{G})|))$ where $p(n)$ is some polynomial. This is clearly true since the number of edges added in the transformation is $|\mathsf{V}(\mathsf{G})|$ and the single vertex v' is added to $\mathsf{V}(\mathsf{G})$ to construct the graph G_c.

Before satisfying the second condition we must first determine the decision problems involved. The associated decision problem for the CGCP is "Is $\chi(\mathsf{G}_c) \leq k$?" Since each vertex in $\mathsf{V}(\mathsf{G})$ is connected to v' then a coloring of G_c must color v' different than any vertex in $\mathsf{V}(\mathsf{G})$; therefore given a coloring for G_c, the colors used by the subgraph induced by $\mathsf{V}(\mathsf{G})$ will require exactly two less colors. This translates to the corresponding decision problem for the SGCP: "Is $\chi(\mathsf{G}) \leq k - 2$?"

Now assume we are given an instance of the SGCP such that $\chi(\mathsf{G}) \leq k-2$ for some $k \geq 3$. That is, an algorithm for solving the decision problem would answer "yes". Consider the problem $(\mathsf{G}_c, \mathsf{K}')$ constructed by using f on this instance. Then any algorithm for this instance of the CGCP must color the subgraph induced by $\mathsf{V}(\mathsf{G})$ with $k - 2$ or less colors and vertex v' will require 2 more colors. Thus the coloring

of G_c would require k or less colors and the corresponding decision problem would answer "yes". Clearly, this same argument applies if the algorithm for the SGCP decision problem answers "no" since $\chi(G) > k - 2$ would mean that $\chi(G_c) > k$. Therefore, the decision problem for the CGCP is $\mathcal{NP}$-complete. ∎

The CGCP can now be shown to be $\mathcal{NP}$-hard from the fact that the associated decision problem is $\mathcal{NP}$-complete. As before we construct a binary search using the associated decision problem as a subroutine. Let G_c be a composite graph for an instance of CGCP. The algorithm that would determine $\chi(G_c)$ is similar to that used to show SGCP is $\mathcal{NP}$-hard and is shown in Figure 24.

```
Let U = sizeof(V(Gc))
Let L = 0
Let k = U div 2 + U mod 2
While U > L + 1 Do
    If CompositeColorable(Gc, k) Then
        Let U = k
    Else
        Let L = k
    End If
    Let k = (U - L) div 2 + (U - L) mod 2 + L
End While
Return k
```

Figure 24. Coloring a composite graph using the decision problem

Again we consider the call to `CompositeColorable` as no different from an addition or multiplication and the above algorithm has complexity $O(\log_2(n))$ where n is the order of G_c. Clearly, the above algorithm has polynomial complexity.

C. STRONGLY $\mathcal{NP}$-COMPLETE RESULTS

Further results can be established showing the difficulty of the decision problem associated with the CGCP. These results are based on the theory of *strongly $\mathcal{NP}$-complete problems*.

Definition 5.1 A problem Π is a *number problem* if and only if there exists no polynomial, $p(n)$ such that for every instance $\mathsf{P} \in \Pi$, MAX(P)$\leq p(|\mathsf{P}|)$, where MAX(P) represents the largest magnitude of any integer appearing in the instance P. ∎

For the PARTITION problem [GJ79, pp. 94-95], whose associated decision problem is $\mathcal{NP}$-complete, an upper bound is placed on the magnitude of the integers in the problem. This restriction allows the construction of an algorithm that is of polynomial complexity in terms of the size of the instance. An algorithm produced in this way for solving an $\mathcal{NP}$-complete problem is said to be *pseudo-polynomial.* Thus given an instance P of the problem Π and a polynomial p such that MAX(P) $\leq p(|\mathsf{P}|)$, then the pseudo-polynomial algorithm for Π will have complexity $O(p(|\mathsf{P}|))$ for the instance P.

Discovering a number problem involves constructing functions for the size of a problem instance SIZE and the magnitude of the largest integer MAX. Some latitude in the construction is available since any polynomially related functions will work as would any "reasonable" representation. This will allow some conveniences when choosing SIZE and MAX functions.

Definition 5.2 A pair of functions (SIZE, MAX) are *polynomially related* to the pair of functions (SIZE′, MAX′) if and only if there exists a polynomial of one variable p and a polynomial of two variables q for all instances $\mathsf{P} \in \Pi$ such that SIZE(P) $\leq$ p(SIZE′(P)) and SIZE′(P) $\leq p$(SIZE(P)) and MAX(P) $\leq q$(SIZE′(P), MAX′(P)) and MAX′(P) $\leq q$(SIZE(P), MAX(P)). ∎

SGCP is an example of an $\mathcal{NP}$-complete problem that is not a number problem. An encoding for an instance of SGCP would include an integer for the number of vertices and an adjacency matrix. The size of an instance of SGCP with this encoding would be $\log_2(n) + n^2$ where n is the number of vertices in the graph. The largest integer in the encoding is n. Thus if $\mathsf{P} \in$ SGCP then SIZE(P) $= \log_2(n) + n^2$ and MAX(P) $= n$. Clearly, MAX(P) $\leq$ SIZE(P) for all $n \in \mathbb{N}$ so that the choice of the polynomial $p(n) = n$ would provide the evidence that SGCP is not a number problem.

On the other hand, CGCP is a number problem. A similar encoding will be used for any instance of CGCP using an adjacency matrix, in addition the chromaticity function will be represented by an array. Therefore, given any instance of $\mathsf{P} \in$ CGCP, SIZE(P) $= \log_2(n) + n^2 + n\log_2(M)$, where n is the number of vertices and M is the maximum chromaticity. The function for determining the maximum integer for the problem instance is MAX(P) $= \max\{n, M\}$. For any fixed n, SIZE(P) is on the order of $O(\log_2(M))$ and MAX(P) is on the order of $O(M)$ and no polynomial exists that will satisfy the inequality $M \leq p(\log_2(M))$. To show this, without loss of generality assume that there exists some $k \in \mathbb{Z}^+$ such that $M \leq (\log_2(M))^k$. Since $\log_2(n) > 0$ and increasing for all $n > 1$, $\log_2(M) \leq k\log_2(\log_2(M))$. Thus,

$$\frac{\log_2(M)}{\log_2(\log_2(M))} \leq k$$

If we consider the limit of the above indeterminate form by applying L'Hospital's rule then

$$\lim_{M\to\infty} \frac{\log_2(M)}{\log_2(\log_2(M))} = \infty$$

This contradicts the fact that a finite k exists that satisfies the inequality.

Definition 5.3 A decision problem Π is *strongly* $\mathcal{NP}$-complete if and only if there exists some polynomial $p(n)$ for which the set of instances Π_p defined to be those instances $\mathsf{P} \in \Pi$ such that MAX(P)$\leq p(|\mathsf{P}|)$ is $\mathcal{NP}$-complete. ∎

Therefore, no pseudo-polynomial algorithm can exist for a number problem that is strongly $\mathcal{NP}$-complete. If an $\mathcal{NP}$-complete problem is not a number problem then the integers used in the encoding of the instances are either fixed for all instances or are directly related to describing the size of the structures involved in the encoding. Thus, just as the example for SGCP, the integers involved are bounded above by a polynomial in terms of the size of the problem. Consequently, any $\mathcal{NP}$-complete problem that is not a number problem is also strongly $\mathcal{NP}$-complete and no pseudo-polynomial algorithm can exist for these problems.

Theorem 5.3 The decision problem associated with the Composite Graph Coloring problem is strongly $\mathcal{NP}$-complete.

Proof We show this by choosing $p(n) = 2$. Now consider the set of instances of CGCP for which $\mathsf{MAX(P)} \leq 2$. This set, CGCP_2, has instances where the graphs satisfy the condition: $\forall v \in \mathsf{V}(\mathsf{G}_c), ch(v) \leq 2$. Consider I, an instance of CGCP_2 such that G_c is the graph in instance I. Let G_c satisfy the following:

1. Given $|\mathsf{V}(\mathsf{G}_c)| = n$ then the set of vertices $\{v \in \mathsf{V}(\mathsf{G}_c) | ch(v) = 1\}$ is of size $n-1$. That is, there is only one vertex with chromaticity 2.

2. If v' is the vertex of chromaticity 2, then for each $v \in \mathsf{V}(\mathsf{G}_c)$, $\{v, v'\} \in \mathsf{E}(\mathsf{G}_c)$

The above construction is the same construction used in the previous section to show that the decision problem associated with CGCP is $\mathcal{NP}$-complete. Clearly, the decision problems associated with instances in CGCP_2 is in $\mathcal{NP}$, because $\mathrm{CGCP}_2 \subset \mathrm{CGCP}$. Since each instance of SGCP can be polynomially reduced to an instance of CGCP_2 then the decision problem associated with CGCP_2 must be $\mathcal{NP}$-complete. Therefore the decision problem of CGCP is strongly $\mathcal{NP}$-complete. ∎

The above shows that the chromaticities of CGCP do not add to the difficulty of the problem. This is clearly true if you consider how most approaches only use the first and last colors assigned to vertices during the coloring process to determine conflicts between adjacent nodes. Further with this result we know that no pseudo-polynomial algorithm exists for the associated decision problem of CGCP.

D. COMPLEXITY OF APPROXIMATION

Because of the difficulty of finding the optimal solutions to $\mathcal{NP}$-hard problems many algorithms compromise quality for efficiency. Approximation algorithms replace the exact versions in practice to provide near optimal solutions using much less resources. The term "near" requires some degree of qualification and indeed can often be measured theoretically. Some of the measures are described in [GJ79, PS82, SW88].

Given an instance P of a CO problem Π with cost function c and the global optimum OPT(P), suppose that an approximation algorithm, $\mathcal{A}_{\text{sub}}$, halts with a feasible solution p. Then the quality of the approximation algorithm can be measured in the following ways:

Definition 5.4 $\mathcal{A}_{\text{sub}}$ is said to be an *ϵ-absolute approximation algorithm* if and only if $\mathcal{A}_{\text{sub}} \in \mathcal{P}$ and there exists $\epsilon > 0$ such that each instance P of the problem Π satisfies the condition $|c(p) - c(\text{OPT}(\text{P}))| \leq \epsilon$. ∎

Definition 5.5 $\mathcal{A}_{\text{sub}}$ is said to be an *ϵ-relative approximation algorithm* if and only if $\mathcal{A}_{\text{sub}} \in \mathcal{P}$ and there exists $\epsilon > 0$ and $M \in \mathbb{Z}^+$ such that for each instance P of the problem Π satisfies the following condition:

$$\text{OPT}(\text{P}) \geq M \Longrightarrow \frac{|c(p) - c(\text{OPT}(\text{P}))|}{c(\text{OPT}(\text{P}))} \leq \epsilon$$

∎

Clearly, the most desirable quality is ϵ-absolute approximation algorithms. As problem size increases, the relative error in the approximation goes to 0. The ϵ-relative approximation algorithm provides a maximum relative deviation of the error for large problems. Garey and Johnson [GJ76] have shown for the SGCP that no ϵ-relative approximation algorithm can exist for $\epsilon < \frac{1}{3}$ unless $\mathcal{P} = \mathcal{NP}$[3]. That is, unless an exact algorithm of polynomial complexity exists to solve the SGCP no approximation algorithm exists with the above guarantee. Thus approximation of the optimum solution for SGCP is just as difficult as finding the optimum. We show similar results for CGCP.

Theorem 5.4 CGCP $\in \mathcal{P}$ if and only if there exists an ϵ-absolute approximation algorithm for CGCP.

Proof First, assume that the CGCP can be solved by an algorithm $\mathcal{A}$ of polynomial

[3]Garey and Johnson [GJ79, p. 128] used a different but equivalent definition for ϵ in which the value was $\frac{4}{3}$ in the original proof.

complexity, then for any choice of $\epsilon > 0$, $\mathcal{A}$ satisfies the conditions for an ϵ-absolute approximation algorithm. Therefore, if CGCP can be solved by an algorithm of polynomial complexity then there exists an ϵ-absolute approximation algorithm for CGCP.

Now assume that there exists an ϵ-absolute approximation algorithm $\mathcal{A}_{\text{sub}}$ for some $\epsilon > 0$ for the CGCP. Let G_c be the graph of an instance of the CGCP. Construct G_c^ϵ as the graph with $\lceil\epsilon\rceil + 1$ isomorphic copies of G_c such that G_c^ϵ is the join of the copies. That is,

$$G_c^\epsilon = \underbrace{G_c \oplus G_c \oplus \cdots \oplus G_c}_{\lceil\epsilon\rceil + 1 \text{ copies}}$$

Figure 25 is an example for $\epsilon = 1.7$.

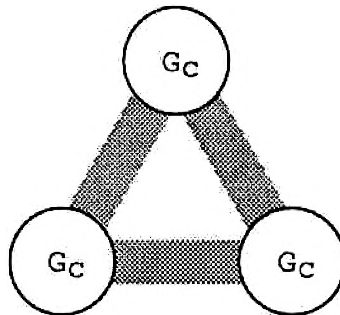


Figure 25. Composite Graph constructed using 3 isomorphic copies of G_c

Clearly, from the above construction then OPT(G_c^ϵ)= $(\lceil\epsilon\rceil + 1)$OPT($G_c$). Now let k be the number of colors used by the feasible solution found by algorithm $\mathcal{A}_{\text{sub}}$ on graph G_c^ϵ. Then

$$(\lceil\epsilon\rceil + 1)\text{OPT}(G_c) \leq k$$

Because $\mathcal{A}_{\text{sub}}$ is a ϵ-absolute approximation algorithm then

$$k \leq (\lceil\epsilon\rceil + 1)\text{OPT}(G_c) + \epsilon$$

Since $\epsilon \leq \lceil\epsilon\rceil$,

$$k \leq (\lceil\epsilon\rceil + 1)\text{OPT}(\text{G}_c) + \lceil\epsilon\rceil$$

Therefore,

$$(\lceil\epsilon\rceil + 1)\text{OPT}(\text{G}_c) \leq k \leq (\lceil\epsilon\rceil + 1)\text{OPT}(\text{G}_c) + \lceil\epsilon\rceil$$

Dividing by $(\lceil\epsilon\rceil + 1)$ yields,

$$\text{OPT}(\text{G}_c) \leq \frac{k}{(\lceil\epsilon\rceil + 1)} \leq \text{OPT}(\text{G}_c) + \frac{\lceil\epsilon\rceil}{(\lceil\epsilon\rceil + 1)}$$

We also know that $\frac{\lceil\epsilon\rceil}{(\lceil\epsilon\rceil+1)} < 1$, so,

$$\text{OPT}(\text{G}_c) \leq \frac{k}{(\lceil\epsilon\rceil + 1)} < \text{OPT}(\text{G}_c) + 1$$

Finally,

$$\text{OPT}(\text{G}_c) = \left\lfloor \frac{k}{(\lceil\epsilon\rceil + 1)} \right\rfloor$$

From the above results we construct an algorithm to solve CGCP with polynomial complexity. Let `ConstructGraphE` be a procedure that will construct the graph G_c^ϵ from the graph G_c. Let `ApproximateChi` represent a function call to the algorithm $\mathcal{A}_{\text{sub}}$ which returns the number of colors used to color the instance of CGCP. `Gc` is the graph of the instance that we are coloring and `e` represents ϵ. The algorithm is given in Figure 26.

```
Ge = GraphCopy(Gc)
For I = 1 to Ceil(e) Do
    Ge = GraphJoin(Ge, Gc)
End For
Let k = ApproximateChi(Ge)
Let chi = Floor(k/(Ceil(e)+1))
Return chi
```

Figure 26. Polynomial algorithm for coloring composite graphs

One can easily show that `GraphJoin` has polynomial complexity $O(|\text{V}(\text{G}_c)|^2 +$

$|E(G_c)|$) which is clearly polynomial in the size of G_c. Therefore, the above algorithm solves any instance of CGCP in a polynomial number of steps. Thus we have shown that ϵ-absolute approximation for CGCP is no easier than finding the optimum. ∎

A much more powerful result is an extension of the results of Garey and Johnson [GJ79].

Theorem 5.5 CGCP $\in \mathcal{P}$ if and only if there exists an ϵ-relative approximation algorithm for CGCP with $\epsilon < 1$.

Proof Assuming that the CGCP can be solved by an algorithm, $\mathcal{A}$, of polynomial complexity then choose any $\epsilon > 0$ and $\mathcal{A}$ satisfies the requirements to be an ϵ-relative approximation algorithm. Thus if CGCP $\in \mathcal{P}$ then there is an ϵ-relative approximation algorithm for CGCP with $\epsilon < 1$.

Now assume there is for CGCP an ϵ-relative approximation algorithm $\mathcal{A}_{\text{sub}}$ with some fixed $\epsilon < 1$ and $M \in \mathbb{Z}^+$. Let G_c be the graph of an instance of CGCP such that $\text{OPT}(G_c) \geq M$ and let k be the number of colors found for the feasible solution returned by $\mathcal{A}_{\text{sub}}$. Then from the definition,

$$k \leq 2\text{OPT}(G_c)$$

Consider the graph of an instance in CGCP of the form $G'_c = (G \cup (\{v'\}, \{\}), ch)$ where G is any standard graph and v' is a vertex not in G. The chromaticity function ch is defined to be 1 for all vertices in $V(G)$ and 2 for vertex v'. Note that the vertex v' is disconnected because the edge set in the union of the graphs is empty. This is used to satisfy the requirement that the graph constructed be a composite graph and we can now use $\mathcal{A}_{\text{sub}}$ to approximate the number of colors for a standard graph.

Once again let k be the number of colors required by the feasible solution returned by $\mathcal{A}_{\text{sub}}$ when given graph G'_c. Then clearly if i is the number of colors used by the subgraph G, $i \leq k$. Also, $\text{OPT}(G) = \text{OPT}(G'_c)$ because v' is not connected to any

vertex in G. Therefore we have,

$$i \leq 2\text{OPT}(G)$$

or

$$\frac{|i - \text{OPT}(G)|}{\text{OPT}(G)} \leq 1$$

From the above results we construct an ϵ-relative approximation algorithm for SGCP. Given a standard graph G, represented as `G` in Figure 27, we use the algorithm $\mathcal{A}_{\text{sub}}$ as the function call `ApproximateChi` which returns the number of colors used.

```
Gc = GraphUnion(G, ({v'},{}))
Let k = ApproximateChi(Gc)
c = BegColor(v')
DeleteVertex(Gc, v')
For i = c  To c + ch(v') - 1 Do
    j = 1
    found = FALSE
    While j <= GraphOrder(Gc) AND NOT found Do
        If i >= BegColor(v(i)) AND i <= EndColor(v(i)) Then
            found = TRUE
        End If
        j = j + 1
    End While
    If NOT found Then
        k = k - 1
    End If
End For
Return k
```

Figure 27. ϵ-relative algorithm for SGCP

The algorithm uses only a polynomial number of steps in the size of the standard graph G. Thus given a standard graph G such that $\text{OPT}(G) \geq M$, then the above algorithm will guarantee that a polynomial number of steps are performed in the size of G.

Garey and Johnson [GJ79, p. 144] have shown the following:

Theorem If there exists an ϵ-relative approximation algorithm for SGCP with $\epsilon < 1$ then $\mathcal{P} = \mathcal{NP}$.

Therefore SGCP $\in \mathcal{P}$, and from previous results in this chapter we know that CGCP is also a member of $\mathcal{P}$. Thus if there is an ϵ-relative algorithm for CGCP with $\epsilon < 1$ for a fixed $M \in \mathbb{Z}^+$ then CGCP $\in \mathcal{P}$. ∎

Using the above technique we can show the following generalization:

Theorem 5.6 There is an ϵ-relative algorithm for SGCP with $\epsilon < N$ for $N \in \mathbb{Z}^+$ if there is an ϵ-relative algorithm for CGCP with $\epsilon < N$.

Proof Assume that for some $N \in \mathbb{Z}^+$, there exists an ϵ-relative algorithm $\mathcal{A}_{\text{sub}}$ for CGCP with $\epsilon < N$ and $M \in \mathbb{Z}^+$. Again we choose an instance with graph G_c such that $\text{OPT}(\mathsf{G}_c) \geq M$. Let k be the number of colors found for the feasible solution returned by $\mathcal{A}_{\text{sub}}$. In this case,

$$k \leq (N+1)\text{OPT}(\mathsf{G}_c)$$

As before, consider instance in CGCP of the form $\mathsf{G}'_c = (\mathsf{G} \cup (\{v'\}, \{\}), ch)$ where G is any standard graph and v' is a vertex not in G with the chromaticity function ch defined to be 1 for all vertices in $V(\mathsf{G})$ and 2 for vertex v'.

If k is the number of colors required by the feasible solution returned by the algorithm when given graph G'_c, then the number of colors used by the subgraph G must be less than k and $\text{OPT}(\mathsf{G}) = \text{OPT}(\mathsf{G}'_c)$ vertex in G. In a similar way we can conclude that, $i \leq (N+1)\text{OPT}(\mathsf{G})$ or

$$\frac{|i - \text{OPT}(\mathsf{G})|}{\text{OPT}(\mathsf{G})} \leq N$$

Clearly, if $\text{OPT}(\mathsf{G}) \geq M$ then $\text{OPT}(\mathsf{G}_c) \geq M$. Using the algorithm in Figure 27, with input of the standard graph `G` with $\text{OPT}(\texttt{G}) \geq M$ and the algorithm $\mathcal{A}_{\text{sub}}$ as the function call `ApproximateChi`, we construct an ϵ-relative algorithm with $\epsilon < N$ and $M \in \mathbb{Z}^+$ for SGCP. ∎

The importance of the above theorem rests in the fact that any results shown concerning the bounds on ϵ for ϵ-relative algorithms for SGCP are now immediately applicable to CGCP. This is the contrapositive of the above statement: If there does not exist an ϵ-relative algorithm for SGCP with $\epsilon < N$ then there does not exist an ϵ-relative algorithm for CGCP with $\epsilon < N$. Thus, the bounds on ϵ for CGCP are directly tied to results for SGCP. The inverse statement has yet to be proven.

Unfortunately, the results of this chapter indicate that the heuristic algorithms of the previous chapter cannot provide guaranteed performance over all instances of CGCP. Further, there is little hope of finding approximation algorithms of polynomial complexity that prove to be "good" over all instances of CGCP.

VI. TABU SEARCH

This chapter introduces the Tabu Search technique used in Combinatorial Optimization described by Glover [Gl89a, Gl90]. This search technique attempts to compete with Simulated Annealing [LA88, JA89, JA90] and Genetic Algorithms [Go89], also used in CO. All of these search techniques attempt to overcome the deficiencies of local search in order to find the global optimum. But whereas both Simulated Annealing and Genetic Algorithms use methods for constructing moves that are not within the current neighborhood in the search, Tabu Search moves from neighborhood to neighborhood by using a "steepest descent—mildest ascent" strategy. Several Tabu Search algorithms are described in this chapter for CGCP and used to color random composite graphs.

The following introductory remarks assume that the problem to which local search and Tabu Search are being applied is a minimization problem.

A. COMPONENTS OF LOCAL SEARCH

Tabu Search is closely related to local search in that the move from configuration to configuration in the search space is based on a "neighborhood" search. A neighborhood is defined by a move function.

Definition 6.1 A *move* is a function m defined for a specified domain $\mathfrak{F}_m$ to the configuration space $\mathfrak{F}$. Thus given a point in the configuration space x, $m(x)$ may or may not be defined. Typically, the move function is defined if $m(x)$ is feasible and undefined when infeasible. ∎

A set of moves, $\mathfrak{M}$, will by defined for the CO problem in order to facilitate the search of the configuration space. The moves define the neighborhoods used in the search.

Definition 6.2 A *neighborhood* is the function $N : \mathfrak{F} \rightarrow \mathfrak{P}(\mathfrak{F})$ where $\mathfrak{F}$ is the

configuration space. For each configuration, $x \in \mathfrak{F}$, N is defined as follows:

$$N(x) = \{y \in \mathfrak{F} | \exists m \in \mathfrak{M}, y = m(x)\}$$

We say y is a *neighbor* of x if $y \in N(x)$. ∎

Before local search can begin, a starting configuration must be chosen as the current configuration to initialize the search. Once this is done, then the neighborhood is constructed and the neighbors are evaluated with respect to the cost function. In minimization problems, any neighbor that has a smaller cost than the current configuration is made the current configuration and another iteration is done by again constructing and evaluating the neighborhood of the current configuration. If there is no such neighbor to the current configuration then the search halts with the current configuration as the solution to the problem instance. Figure 28 shows the local search technique. Of course, local search halts with the global optimum if the cost function is unimodal over the configuration space. Otherwise, local search only finds the optimum over the final neighborhood, called a *local optimum*. In general, cost functions are not unimodal and this presents a problem to optimizing many practical problems.

The performance of the algorithm can also be affected by the interaction of the starting configuration and the neighborhoods defined for the search. A poor starting condition usually leads to a poor solution when using local search. One simple suggestion made to overcome some of the drawbacks to local search is to use a *Multistart* local search. Multistart local search initializes several local searches on the problem by randomly choosing different starting configurations for each search. Even though this technique has potential, all of the more recent search techniques provide more powerful techniques in overcoming these drawbacks.

```
Let CurrentCfg = StartCfg(F)
Let MinFound = False
While NOT MinFound Do
    EVALUATE:
    MinFound = True
    For EACH Neighbor IN Neighborhood(CurrentCfg)
        If Cost(Neighbor) < Cost(CurrentCfg) Then
            MinFound = False
            CurrentCfg = Neighbor
            Goto EVALUATE:
        End If
    End For
End While
Return CurrentCfg
```

Figure 28. Local search algorithm

B. COMPONENTS OF TABU SEARCH

Tabu Search was first introduced by Glover and has since been applied to many different problems: scheduling [DT92, MR92], traveling salesman [Fi90, MG89], vehicle routing [Os92, ST92], quadratic assignment [Ta91], graph coloring [HW87, JG91], maximum clique [GS92], independent sets [Fr90], multiconstraint 0-1 knapsack problem [DV92], and neural networks [WH89].

1. Simple Tabu Search As with local search, Tabu Search attempts to find a localized "best" configuration at each iteration. The unique feature of Tabu Search is the tabu list. This is a list of moves that are considered "taboo" or restricted from use. Also, Tabu Search evaluates the entire neighborhood before choosing the best neighbor configuration. The best configuration in a minimization problem is the configuration that has the minimum cost over the neighborhood. This configuration then becomes the current configuration regardless of the fact that it might have a cost higher than the current configuration. Thus, a "mild" ascent out of a local optimum is possible, making the move the least damaging to the current optimum. Figure 29 shows the Simple Tabu Search algorithm for CO.

By definition, the tabu list T is a subset of $\mathfrak{M}$, kept at each iteration in order

```
Let CurrentCfg = StartCfg(F)
Let BestCfg = CurrentCfg
Let TabuList = {}
Let count = 0
While count < iterations Do
    Let count = count + 1
    Let BestMove = FirstNonTabuMove
    For EACH Move IN MoveSet(CurrentCfg)
        If NOT (Move IN TabuList) Then
            If Cost(Move(CurrentCfg))
                    < Cost(BestMove(CurrentCfg)) Then
                Let BestMove = Move
            End If
        End If
    End For
    Let CurrentCfg = BestMove(CurrentCfg)
    If Cost(CurrentCfg) < Cost(BestCfg) Then
        Let BestCfg = CurrentCfg
    End If
    Let TabuList = TabuList + {BestMove}
End While
Return BestCfg
```

Figure 29. Simple Tabu Search algorithm

to restrict the search from visiting a previously visited configuration. This aids in diversifying the search as well as eliminating cycling during the search. A tabu list can also be defined as solution specific moves. A solution specific move is the tuple (x, m), where $x \in \mathfrak{F}$ and $m \in \mathfrak{M}$. This would allow the selection of a move if it has not been applied to the configuration under consideration in some past iteration.

Definition 6.3 A move $m \in \mathfrak{M}$ is said to be *tabu* if $m \in T$ and m is said to be *admissible* if $m \in \mathfrak{M} - T$. ∎

In practice, the tabu list can be kept in several forms depending on the details that are needed to provide efficient search. For most applications, the tabu list reflects only a portion of the configuration that has changed. This is done as a means of efficiency in time and space. Thus the tabu list restricts previous moves but also excludes admissible moves because of this incomplete information. This is not much of a problem since the tabu list is also finite and tabu information eventually cycles through the list releasing moves from the tabu status.

As with local search a starting configuration must be chosen but unlike local search there is no specified stopping condition for Tabu Search. For most applications the stopping condition is an absolute number of iterations of the algorithm. Hertz and de Werra [HW87] apply Tabu Search to SGCP by partitioning the vertex set and assigning to each partition a color. This partition does not guarantee a feasible solution to the problem instance, thus the unique stopping condition: end the search as soon as you find a feasible solution. Another type of stopping condition is to allow the best configuration to remain stable over a specified number of iterations.

2. Aspiration Levels The idea of admissibility can be extended to make the Tabu Search more robust. As before we will admit moves that are non-tabu, but situations arise where some tabu move would meet some of our aspirations for improvement. Thus, we would allow the tabu restriction to be relaxed and admit the move. This relaxation is accomplished by defining an *aspiration level* for each solution specific move (x, m) such that if the cost of the solution is less than the aspiration level $A(x, m)$ then the move is admissible. Therefore, we say that a move $m \in \mathfrak{M}$ is

admissible if $A(x,m) < c(x)$ or $(x,m) \notin T$.

One of the difficulties involved in allowing the tabu status to be overridden is the problem of cycling. A cycle is a sequence of iterations of length $k > 1$ such that if i is the first iteration then the configurations $x_i = x_{i+k}$. Although impractical, cycling can be prevented if a complete history of the search were kept and denying any move to a configuration x if x has been visited previously. Two weaker forms for preventing a cycle are suggested by Glover [Gl89a, p. 193]:

1. The move m has been applied to x before.

2. The move m^{-1} has been applied to $m(x)$ before.

The first strategy prevents cycling if the move from x to $m(x)$ has already been made. Thus we may visit a previously visited solution by means of a different path. The second strategy will allow the last move to be immediately reversed, but prevents it from happening a second time. Both of these strategies provide a more efficient method for cycle inhibition when compared to the former method.

3. Implementing Tabu Lists and Aspiration Levels Implementations of the tabu list involve keeping a list of *steps* of the form $(x, m(x))$ where $m(x)$ is the configuration obtained by applying move m to configuration x. Thus, (x,m) is tabu if and only if $(x, m(x)) \in T$. The list is of a finite length t and is usually a circular list where only the t most recent moves are considered inadmissible. When the next iteration takes the step $(y, m(y))$, then the list is updated by deleting the oldest remaining step and inserting the step $(y, m(y))$.

The aspiration level can be implemented as $A : C \rightarrow C$ where C is the range of the objective function $c(x)$. Initially for all i, $A(i)$ is set to $\max_{x \in \mathfrak{F}}\{c(x)\}$, then at each iteration the function is updated with the assignment:

$$A(i) = \min\{A(i), c(m(x))\}$$

So a step $(x, m(x))$ would be admissible if $(x, m(x)) \notin T$ or $c(m(x)) < A(c(x))$.

Another way of implementing the aspiration level is by using a global aspiration level for the search. That is, the aspiration level A for any iteration is $A = c(\mathbf{best})$ where **best** is the configuration that has the smallest cost for any configuration found so far in the search. Initially, $A = c(\mathbf{start})$ where **start** is the starting configuration. At each iteration, the aspiration level is updated by the assignment:

$$A = \min\{A, c(m(x))\}$$

This is the best way to implement the aspiration level for objective functions with a range over $\mathbb{R}$.

C. TABU SEARCH ALGORITHMS FOR CGCP

This section presents the model for CGCP used in this study. Several neighborhoods are developed for use within this model as well as the starting configurations.

1. Related Research Graph coloring models exist for both the standard and composite graphs. Hertz and de Werra [HW87] constructed a Tabu Search algorithm for SGCP. Jenness and Gillett [JG91] constructed a parallel version of the Tabu Search algorithm for CGCP.

Given a standard graph G, Hertz and de Werra used the notion of set partition to color the graph. Finding the coloring of a graph is the same as finding the partition of the set V(G) so that the sets of the partition form independent sets, that is, if π is a partition of V(G) such that $\pi = \{V_1, V_2, \ldots, V_p\}$, for each edge $\{v, u\} \in E(G)$ if $v \in V_i$ and $u \in V_j$ then $i \neq j$. Once a partition is established then the coloring is found by assigning a color to each $V_i \in \pi$; this color is inherited by each vertex in the set. The partition π is a *minimal coloring* if π satisfies the independence condition and if π' is any other partition satisfying the independence condition then $|\pi| < |\pi'|$. The chromatic number of the graph is $\chi(G) = |\pi|$.

The Tabu Search implemented by Hertz and de Werra searched the infeasible partitions of the vertex set for a fixed partition size p until a coloring was found or a specified number of iterations had expired. The objective function is defined for

the partition $\pi = \{V_1, V_2, \ldots, V_p\}$ by first defining the sets $R(V_i)$ as $R(V_i) = \{e \in E(G) | e = \{v, u\} \wedge v, u \in V_i\}$. $R(V_i)$ is the set containing all edges in E(G) that have both vertices in set V_i. The objective function is:

$$c(x) = \sum_{i=1}^{p} |R(V_i)|$$

A random partition is generated for the starting configuration so that $c(x) > 0$. This also provides the stopping condition for the search: stop if $c(x) = 0$.

A move within the space consists of choosing at random a vertex v associated with some edge in $\cup_{i=1}^{p} R(V_i)$. Suppose that $v \in V_i$, then another point is generated by providing a new color j for v such that $i \neq j$. The neighboring partition by recoloring v is given by:

$$(V_1, V_2, \ldots, V_i - \{v\}, \ldots, V_j \cup \{v\}, \ldots, V_p)$$

The moves are generated by sampling the neighborhood of the current solution.

A tabu list has a fixed size and is updated at each iteration by adding to the list (v, i) so that v will not be returned to V_i (as long as it remains on the tabu list). The aspiration level is implemented by initially assigning $A(i) = i - 1$ for all i. If a move involves stepping from partition x to partition y such that $c(y) < A(c(x))$ then $A(c(x))$ is updated with the assignment: $A(c(x)) = c(y) - 1$. The algorithm used by Hertz and de Werra is as follows:

1. Input graph G, partition size p, tabu list size t, neighborhood sample size s, and maximum number of iterations n.

2. Generate a random partition $x = (V_1, V_2, \ldots, V_p)$. Choose an arbitrary tabu list T of size t. Let iterations = 0.

3. *Repeat until* $c(x) = 0 \vee$ iterations $\geq n$

   Generate s neighbors y such that $(x, y) \notin T \vee c(y) \leq A(c(x))$ (generation halts if a neighbor y is found such that $c(y) < c(x)$). Let x^* be the best neighbor

generated. Update the tabu list T by inserting the move (x, x^*) and removing the oldest move. Let $x = x^*$ and increment iterations.

4. If $c(x) = 0$ then $x = (V_1, V_2, \ldots, V_p)$ is a p-coloring of G, otherwise, the search failed to find a p-coloring of G.

One drawback to the above method is the potential to fail. This failure does not produce a coloring for the graph. The algorithm could be made into a procedure call to a binary search that would strategically set upper and lower bounds for the interval of search.

Jenness and Gillett implemented a similar algorithm for CGCP in which success was guaranteed. It is based on the probabilistic lower bounds given by Oakes [Oa90]. Given a composite graph G_c, the algorithm would begin with random partition π_1 of $V(G_c)$ whose size is given as the probabilistic lower bound of the chromatic number for the graph. The algorithm attempts to find a color using a partition of this size over a specified number of iterations; if this partition size fails then the partition size is increased by 1 and the search continues. Whenever the cost $c(x) = 0$ the algorithm halts.

The cost function was implemented to reflect the chromaticities associated with the vertices. The function P was defined for each vertex in a partition $(V_1, V_2, \ldots, V_p)$ as:

$$P(v) = \sum_{\{v,u\} \in E(G_c) \wedge v,u \in V_i} |ch(v) \cap ch(u)|$$

In other words, given a vertex v, $P(v)$ sums the number of color conflicts for all adjacent vertices u in the same partition as v, by finding the intersection of their color sets. The total cost of a configuration is the sum of all vertex costs, $c(x) = \sum_{i=1}^{|V(G_c)|} P(v_i)$.

Moves are chosen by randomly selecting a vertex favoring vertices for which $P(v) > 0$. Once a vertex is chosen then a new color is given so that the vertex is in a different partition. The tabu list records the moves as the tuple (v, i), where v is the vertex being recolored and i is the old color. The algorithm was implemented on an Intel

Table XXV. Results from Jenness and Gillett for CGCP using Tabu Search

(*, 0.5, TPOI(1), 25)

graph order	number of processors	number of iterations	colors used	time (secs)
50	16	300	18.8	38.0
	8	100	20.6	10.1
100	16	100	34.3	44.3
	16	300	30.9	67.9
	8	300	31.4	46.1
500	16	100	126.0	307.0
	16	300	116.5	748.9

IPSC/2 hypercube with 16 processors. Table XXV provides the results of that study. Observations show that the number of processors used and the increase in the number of iterations per trial partition lower the average number of colors used and increase the run-time associated with the search. Even though the results improved the average number of colors used by the heuristics on small graphs, the improvements did not meet the upper probabilistic bounds on any of the graph orders tested. Considering these results, the implementation was only moderately successful.

2. Configuration Space This section describes the configuration space used by the Tabu Search algorithms that we have constructed. This configuration space is based on the vertex sequential algorithms described in Chapter III.

The vertex sequential algorithms construct a permutation of the vertices and apply a minimum color to each vertex in sequence to obtain a coloring of the graph. Recall that the colors are assigned to vertices $v_1, v_2, \ldots, v_{i-1}$ before vertex v_i. Figure 30 is the algorithm for assigning the minimum color to vertex v_i in the sequence. Elmer [El93, p. 27] has shown that there is some permutation of the vertices so that when applying the algorithm in Figure 30 the optimal solution is obtained.

Theorem The optimal solution to a composite graph is obtainable by coloring the graph using the basic vertex sequential algorithm with some

ordering of the vertices.

Therefore, the configurations will be the set of all permutations of the vertex set $V(G_c)$. If the order of the graph is n then the size of the configuration space is $n!$. This is much too large to exhaust even for a graph of order 25. The Tabu Search will establish a neighborhood for each configuration and sample the configuration space by moving from neighborhood to neighborhood.

One of the advantages of the above configuration space is the fact that any moves we establish will produce another permutation of the vertex set. This permutation is always guaranteed to be a feasible coloring once we apply the algorithm in Figure 30. This eliminates some of the search time required in making a move since we eliminate the test for an infeasible move.

```
Let color = 1
LOOP:
For j = 1 to i - 1
    If {v(i), v(j)} in E(Gc) AND
        color <= EndColor(v(j)) AND
        BegColor(v(j)) <= color + ch(v(i)) Then
        Let color = EndColor(v(j)) + 1
        Goto LOOP:
    End If
End For
Return color
```

Figure 30. Function for finding the minimum color for vertex v_i

3. Implementation of Components Any implementation of Tabu Search must also describe the starting configuration, the condition to end the search and the management of details of the tabu list for the configuration space.

The Tabu Search algorithms described here will allow for both a random starting point and a heuristically ordered starting point. The random starting point will be some random permutation of the vertex set for the graph. The heuristically ordered starting point will use one of two of the static orders described as vertex sequential heuristic algorithms. The two orders used for these points will be CLF and LF2.

In order to determine the end of the search, a specified number of iterations must be performed after a change is found. Once a change is found then the count starts over. If the number of iterations expires before a change is found then the search halts. This allows a rough form of stabilization of the solution before the search ends.

The information kept in the tabu list is dependent on the type of neighborhood that is established in the search. Two types of moves are used to construct the neighborhoods: pairwise swapping and vertex insertion. Given a permutation, pairwise swapping will select two positions in the permutation and swap the vertices in those positions. Thus if $v_1 v_2 \ldots v_i \ldots v_j \ldots v_n$ is a permutation then the move (i, j) will result in the permutation $v_1 v_2 \ldots v_j \ldots v_i \ldots v_n$ The tabu list will keep the information (i, j) and (v_j, v_i) in order to disallow the reversal of the currently accepted move. Vertex insertion moves a single vertex to a new position, so that if (i, j) is a move with vertex insertion then the permutation $v_1 v_2 \ldots v_i \ldots v_j \ldots v_n$ becomes the permutation $v_1 v_2 \ldots v_{i-1} v_{i+1} \ldots v_{j-1} v_i v_j \ldots v_n$. The tabu list will store (j, i) and v_i.

4. Neighborhoods The neighborhood is one of the most important aspects of Tabu Search. The choice of a poor neighborhood can lead to long run-times and an unsatisfactory solution. One of the focuses of these algorithms is to test several neighborhoods to determine the behavior of the Tabu Search when modeling CGCP using vertex permutations. We developed 8 different neighborhoods to test the model.

The first neighborhood is constructed using moves PI which use permutations over an interval. Since the entire configuration space is permutations over the entire vertex set we restrict the permutation over a "small" interval of positions in the configuration. For instance, if we have a graph G_c of order 5 then a configuration might be given by a permutation of the vertex labels separated by commas: 3,2,1,4,5. The positions are labeled from left to right starting with 1. If we consider permutations over an interval of 3 then we obtain the following neighbors:

3,1,2,4,5	2,3,1,4,5	2,1,3,4,5	1,3,2,4,5	1,2,3,4,5
3,2,4,1,5	3,1,2,4,5	3,1,4,2,5	3,4,2,1,5	3,4,1,2,5
3,2,1,5,4	3,2,4,1,5	3,2,4,5,1	3,2,5,1,4	3,2,5,4,1

Each row above corresponds to a position for the interval. Row one corresponds to the interval starting at position 1 and ending at position 3. Row two corresponds to the interval starting at position 2 and ending at position 4 and finally row three corresponds to the interval starting at position 3 and ending at position 5. Notice that the interval "slides" the length of the configuration to allow movement of each vertex. The problem with this specification is that the interval size must be "small" because a permutation over an interval of length i will produce $i! - 1$ neighbors for each position.

The neighborhood using PWS moves is based on pairwise swaps over an interval of the positions in the permutation. Over any interval, all pairwise swaps that are possible are made. Thus, given the graph above with the permutation 3,2,1,4,5 then all of the neighbors using an interval of size 3 are:

2,3,1,4,5   1,2,3,4,5   3,1,2,4,5

3,1,2,4,5   3,4,1,2,5   3,2,4,1,5

3,2,4,1,5   3,2,5,4,1   3,2,1,5,4

As before, each row corresponds to a position of the interval. When compared to neighborhood using PI moves, using only PWS moves dramatically reduces the size of the neighborhoods as the size of the interval grows.

The next neighborhood uses SH moves is based on the movement of a single vertex by vertex insertion. SH is similar to PWS but instead of making swaps, it will attempt to "shuffle" a single vertex to a new position. If a vertex is moved from position i to j then $j - i$ vertices are shifted along with the movement of the vertex at position i to position j. As before, with permutation 3,2,1,4,5 and an interval size of 5, the neighbors are:

2,3,1,4,5   2,1,3,4,5

3,1,2,4,5   3,1,4,2,5

1,3,2,4,5   3,2,4,1,5   3,2,4,5,1

3,4,2,1,5   3,2,1,5,4

3,2,5,1,4

The interval used above is centered around the vertex being shuffled. Thus each row above corresponds to each vertex being centered on an interval. Note that all redundancies have been removed.

The next two neighborhoods are based on PWS and SH moves. A neighborhood constructed using VC moves cycles vertices from the first and last position to any other position by swapping. Neighborhood using VI moves follows the same pattern but uses insertion instead of swapping when moving the vertices. Thus if 3,2,1,4,5 is the permutation then the neighborhood using VC is:

2,3,1,4,5 1,2,3,4,5 4,2,1,3,5 5,2,1,4,3
3,5,1,4,2 3,2,5,4,1 3,2,1,5,4

The neighborhood of the same permutation using VI moves is:

2,3,1,4,5 2,1,3,4,5 2,1,4,3,5 2,1,4,5,3
5,3,2,1,4 3,5,2,1,4 3,2,5,1,4 3,2,1,5,4

The set of moves CR attempts to use the current coloring in order to construct the neighborhood. If the highest color assigned in the current coloring is c then all vertices v_i with c in their color set are moved to a new position in an attempt to reduce the number of colors used by the graph. The movement of the vertices is by vertex insertion instead of swapping. The neighborhood of a permutation is then the set of all permutations obtained by inserting some vertex v_i that uses the color c in a new position. The size of the neighborhood depends on the number of vertices that are colored with color c.

The moves FH use a subset of the PWS moves. The neighborhood constructed using FH moves is at least four times smaller than the neighborhood using PWS moves. A neighborhood is constructed by first establishing a "fence" for a permutation. A fence is a fixed position over which vertices are swapped. Fences are established in several ways.

1. Fixed position (FP): the fence position is fixed at a specified position for all iterations in the search.

2. Random position (RP): the fence position is randomly selected at each iteration.

3. Forward cycling (FC): the fence position is cycled from the first position to the last position over all iterations.

4. Backward cycling (BC): the fence position is cycled from the last position to the first position.

5. Permuted cycling (PC): the fence position is cycled using a permutation of the positions. The permutation is randomly selected at the end of each cycle.

As an example, consider the permutation 3,2,1,4,5 with a fence at position 2 then the neighbors for this permutation using FH moves is:

1,2,3,4,5   4,2,1,3,5   5,2,1,4,3
3,1,2,4,5   3,4,1,2,5   3,5,1,4,2

The fence position 2 falls between the vertices 2 and 1 above. In general, a fence position of p will divide the permutation at positions p and $p+1$.

The last neighborhood that we construct is based on a slightly different implementation of Tabu Search. The moves for constructing the neighborhoods are a subset of PWS moves. When the Tabu Search starts, a subset of positions are randomly selected and assigned tabu status. These positions are "frozen" for a number of iterations equal to the length of the tabu list. Every position that is assigned tabu status cannot be either the source or target of a move. This form of establishing tabu status requires no overhead to search the tabu list as did the previous implementations. This is done by using two lists, a non-tabu list and a tabu list. The tabu list and non-tabu list are both nonempty and form a partition over the position indices of the configuration. As mentioned previously, a random proper subset of the positions are used to start the tabu list. Once this is done, then a random position in the non-tabu list is selected and the vertex at this position in the configuration is swapped with any vertex whose position is also in the non-tabu list. Merely by selecting from the non-tabu list of positions, we determine all non-tabu moves. As an example, let us

once again consider our permutation 3,2,1,4,5 and also suppose that the non-tabu list was $\{2,4,5\}$ and the position selected from this list at random was 4. Then the neighbors produced are:

3,4,1,2,5   3,2,1,5,4

We call this type of move: vertex relocation (VR). This provides a much more efficient implementation of Tabu Search than the previous implementations.

One final note about neighborhoods should be mentioned. Even though all of the above moves generate neighborhoods with a size that is polynomially related to the size of the problem (PI is not polynomially related to interval size), for "large" graphs the size of the neighborhoods can require large amounts of computer time to evaluate. Therefore, a simple sampling strategy was devised in order to sample the neighborhoods. The set of all moves is generated for the specified configuration (recall that this can be done since all moves produce a feasible configuration). A permuted indices is then generated from 1 to the size of the neighborhood so that each neighbor has an equal chance of being selected in the sampling. If the number of samples requested is s then the first s indices are selected as the sample of the neighborhood.

D. ANALYSIS OF ALGORITHMS

This section examines the factors contributing to the complexity of the above algorithms. The complexity is analyzed for each algorithm and compared.

The Tabu Search algorithms are all based on the algorithm in Figure 29. Clearly, one of the major factors of this algorithm is the number of iterations that the search executes. Within each iteration, the algorithm can be divided into 3 steps. The first step is to determine the first non-tabu move. In the worst case analysis, the tabu list is exhausted before a non-tabu move is found. Thus if t is the length of the tabu list then all t tabu moves must be examined. Comparing moves is almost constant for any choice of the above algorithms, so that the complexity of determining the first non-tabu move is $O(t)$. The next step is examining each neighbor of the current

configuration. This includes examining the cost of determining the coloring of each neighbor in order to find the best neighbor. Given a vertex, determining a color requires on the order of $O(|V(G_c)|^2)$ steps. Thus coloring the graph G_c requires on the order of $O(|V(G_c)|^3)$ steps. Since each neighbor is colored, then the examination of the entire neighborhood will require on the order of $O(n|V(G_c)|^3)$ steps where n is the size of the neighborhood. The last step in the iteration requires updating the tabu list. Our implementation uses an array to represent the tabu list so that the steps involved are constant regardless of the order of the graph or the size of the tabu list. In the final analysis, the number of steps using the Tabu Search with our implementation is on the order of $O(i(t + n|V(G_c)|^3))$ where i is the number of iterations, t is the size of the tabu list, and n is the size of the neighborhood.

Further analysis of the neighborhood size n will provide additional details of the factors involved in the algorithms. The size of the neighborhoods is directly related to the order of the graph. The neighborhood generated using moves PI will be of size $(|V(G_c)| - j + 1)(j! - 1)$ where j is the interval size chosen. Because the size of the interval causes the size of the neighborhood to grow exponentially, it is impractical to allow $j \geq 5$. The moves PWS will produce a neighborhood of the size $(j - 1)(|V(G_c)| - j + 1) + (j - 1)(j - 2)/2$ where j is again the size of the interval. Note that if the interval were allowed to be as large as the order of the graph then the neighborhood size would be on the order of $O(|V(G_c)|^2)$. Moves SH for our implementation will generate $2(m|V(G_c)| - j(j + 1)/2) - |V(G_c)| + 1$ where $m = (j - 1)/2$ (j is the length of the interval). The order for PWS is the same for SH as the interval approaches the order of the graph $O(|V(G_c)|^2)$. Both of the move sets VC and VI are of order $O(|V(G_c)|)$. The number of neighbors created by moves VI is $2(|V(G_c)| - 1)$ while moves VC the number of neighbors created is $2(|V(G_c)| - 1) - 1$. The neighborhood for CR moves is somewhat harder to analyze since the number of vertices whose color set contains the highest color assigned depends on the graph and coloring. But an upper bound can be obtained by assuming the worst case, that is, all of the vertices' color sets contain the highest color assigned. Of course, this is

Table XXVI. Complexity of the Tabu Search algorithms for CGCP

$S = |V(G_c)|$, t = tabu list size,
i = number of iterations, j = interval size

PI	PWS	SH	VC
$O(i(t+j!S^4))$	$O(i(t+S^5))$	$O(i(t+S^5))$	$O(i(t+S^4))$

VI	CR	FH	VR
$O(i(t+S^4))$	$O(i(t+S^5))$	$O(i(t+S^5))$	$O(iS^4)$

not possible but we do have an upper bound for the size of the neighborhood. This assumption reduces the neighborhood to the same neighborhood produced by SH moves with an interval size of $|V(G_c)|$. Therefore, CR is on the order $O(|V(G_c)|^2)$. As mentioned previously, FH moves produces a neighborhood approximately one-fourth the size of the neighborhood produced by PWS. The size is at most $|V(G_c)|^2/4$ depending on the position of the fence.

The last neighborhood requires separate analysis since the Tabu Search is implemented somewhat differently. In particular, the search for the first non-tabu move is in constant time because of the use of a non-tabu list. The neighborhood generated by VR moves is $|V(G_c)| - t - 1$ in size where t is the size of the tabu list. The complexity of the Tabu Search algorithm using VR moves is on the order of $O(i|V(G_c)|^4)$.

Table XXVI summarize the complexity for each of these algorithms. These can be somewhat simplified if we make three assumptions:

1. The tabu list size t will always be "small" when compared to $|V(G_c)|$.

2. The number of iterations is on the order of $O(|V(G_c)|)$.

3. The size of the interval j for the CO moves will never be larger than 4.

This would allow us to categorize the Tabu Search algorithms with respect to complexity. The algorithms based on the moves PWS, SH, CR, and FH are on the order of $O(|V(G_c)|^6)$ while the algorithms based on the moves CO, VC, VI, and VR are on

the order of $O(|V(G_c)|^5)$. All algorithms are of polynomial complexity with respect to the size of the graph but the order is too high and sampling techniques will need to be incorporated so that larger graphs can be processed.

E. EXPERIMENTATION

This section will examine the results obtained by coloring random composite graphs with the algorithms described in the previous sections. The experimentation will be done in the following steps:

1. Preliminary results will be established on graphs of order 50 for edge densities 0.2 and 0.5. Each algorithm will be run on each data set and the algorithms that prove to run more efficiently and on the average use fewer colors will be used for further experimentation.

2. The Tabu Search algorithm using FH moves will be tested to establish differences in the moving of the fence.

3. The better algorithms will be tested with the sampling rates of 10%, 25%, 50% and 100% to determine the difference in performance. Also the effectiveness of the tabu list is examined.

4. The algorithms and sampling rate that provides the overall best performance will be run on 100 vertex graphs with edge densities 0.2 and 0.5 in order to further determine the run-time behavior and performance.

5. The algorithm established as being the best overall performer will be tested on 200 and 300 vertex graphs with edge densities 0.2 and 0.5.

All data sets used will be those generated by Oakes [Oa90]. This will allow comparisons to be made to the heuristic algorithms in Chapter IV.

1. Preliminary Testing The data sets for this testing will be the graph of order 50 with edge densities 0.2 and 0.5 as generated by Oakes. Each of the algorithms for Tabu Search will be tested using 100 iterations and a tabu list size of 25. In

addition, each neighborhood will be fully sampled for each run. The Tabu Search algorithms tested will be those with neighborhoods obtained by using PI, PWS, SH, VC, VI, CR, and VR moves. The FH moves will be tested separately for different methods for establishing the fence. In all of the tables the algorithms will be identified by the type of moves that the algorithm uses, that is, the Tabu Search algorithm using the PWS moves will be identified in the table as PWS, and so on. Table XXVII shows the performance of these algorithms on graphs with 50 vertices and an edge density of 0.2. This table provides the average, maximum, and minimum for the number of iterations, the number of colors used, the time in seconds, and the number of moves rejected because of tabu status. The last is a measure of the effectiveness of the tabu list structure during search. Table XXVIII provides the same results for graphs of order 50 with an edge density of 0.5. Note that tabu status is not tested using VR moves because the algorithm used a non-tabu list in choosing moves.

The results of the Tabu Search algorithms that use the FH moves are in Table XXIX and Table XXX. Each of the methods of establishing the fence are tested. The tests were each run with 100 iterations and a tabu list size of 25. The neighborhoods were sampled at 100%.

The number of wins for all 12 algorithms is given in Table XXXI for both densities. From the previous statistics on performance, the algorithms that will be further analyzed are the Tabu Search algorithms that use vertex relocation (VR) and fence moving based on random selection (RP). The VR moves provides next to the lowest average number of colors used for graphs of density 0.2 and the lowest for graphs of density 0.5 as well as the fastest run-times of any algorithm tested. This will allow a large number of iterations for the final testing. The algorithm based on RP was selected as the next overall best because of the low average number of colors used and a smaller variance than that of the algorithm based on PC. Even though algorithm CR performed the best on graphs with density 0.2 it did not perform well on graphs with density 0.5. PWS and SH both had the highest run-times and would produce unreasonable run-times for a higher number of iterations.

Table XXVII. Results for Tabu Algorithms on graphs (50, 0.2, TPOI(1), 25)

	PI	PWS	SH	VC	VI	CR	VR
iterations							
average	101.6	101.5	112.7	101.2	111.8	116.1	108.4
minimum	100.0	100.0	100.0	100.0	100.0	100.0	100.0
maximum	124.0	108.0	198.0	104.0	200.0	183.0	149.0
colors used							
average	10.40	10.04	10.04	10.20	10.36	9.92	9.96
minimum	9.00	8.00	9.00	8.00	9.00	8.00	8.00
maximum	12.00	12.00	12.00	12.00	12.00	12.00	12.00
time (secs)							
average	568.1	803.2	1152.2	65.4	78.3	73.9	23.9
minimum	509.4	730.7	927.1	61.4	62.7	55.3	20.2
maximum	672.6	854.9	2164.2	70.9	145.7	127.3	32.5
rejections							
average	890.3	56.6	273.6	51.3	196.8	4.3	—
minimum	321.0	49.0	195.0	49.0	160.0	0.0	—
maximum	1206.0	76.0	483.0	58.0	382.0	66.0	—

Table XXVIII. Results for Tabu Algorithms on graphs (50, 0.5, TPOI(1), 25)

	PI	PWS	SH	VC	VI	CR	VR
iterations							
average	104.9	102.6	116.0	102.7	113.5	127.4	121.1
minimum	100.0	101.0	100.0	100.0	100.0	101.0	101.0
maximum	129.0	114.0	200.0	114.0	151.0	171.0	184.0
colors used							
average	18.20	17.96	17.96	18.16	18.00	18.20	17.76
minimum	16.00	15.00	15.00	15.00	15.00	15.00	15.00
maximum	22.00	20.00	22.00	21.00	20.00	21.00	20.00
time (secs)							
average	988.5	1463.0	2053.0	114.5	134.6	109.9	42.2
minimum	837.1	1323.1	1448.7	105.8	105.7	76.3	34.6
maximum	1311.0	1721.3	3746.8	126.8	189.9	161.1	67.4
rejections							
average	730.3	57.4	267.0	55.8	166.2	0.2	—
minimum	362.0	49.0	98.0	49.0	131.0	0.0	—
maximum	1214.0	83.0	496.0	97.0	220.0	3.0	—

Table XXIX. Results for FH moves on graphs $(50, 0.2, \mathrm{TPOI}(1), 25)$

	FP	RP	FC	BC	PC
iterations					
average	100.5	119.7	115.8	127.3	112.6
minimum	100.0	100.0	100.0	100.0	100.0
maximum	103.0	193.0	194.0	242.0	166.0
colors used					
average	10.60	10.08	10.12	10.08	10.08
minimum	8.00	8.00	8.00	9.00	8.00
maximum	12.00	12.00	12.00	12.00	12.00
time (secs)					
average	460.5	365.2	359.4	386.0	294.4
minimum	427.2	273.0	300.3	279.4	221.0
maximum	497.4	594.7	629.6	731.8	449.8
rejections					
average	57.4	5.0	1.0	0.7	4.4
minimum	50.0	0.0	0.0	0.0	0.0
maximum	87.0	37.0	50.0	36.0	27.0

Table XXX. Results for FH moves on graphs $(50, 0.5, \mathrm{TPOI}(1), 25)$

	FP	RP	FC	BC	PC
iterations					
average	105.0	125.9	115.9	127.4	119.0
minimum	100.0	101.0	100.0	101.0	101.0
maximum	178.0	193.0	189.0	199.0	199.0
colors used					
average	19.00	17.88	17.96	17.92	17.88
minimum	16.00	15.00	15.00	15.00	15.00
maximum	22.00	20.00	20.00	21.00	22.00
time (secs)					
average	875.3	732.8	650.9	718.0	588.2
minimum	679.2	522.6	510.0	537.6	417.8
maximum	1450.0	1172.0	1046.0	1073.0	1328.0
rejections					
average	63.5	5.9	2.0	1.9	6.1
minimum	49.0	1.0	0.0	0.0	1.0
maximum	150.0	32.0	51.0	47.0	37.0

Table XXXI. Number of wins for Tabu Search algorithms

density	PI	PWS	SH	VC	VI	CR	VR	FP	RP	FC	BC	PC
0.2	5	13	14	10	6	16	15	3	15	13	13	12
0.5	9	9	15	7	8	9	15	0	12	11	12	15

2. Testing Parameters The Tabu Search algorithms allow the use of different starting configurations as well as sampling rates for the neighborhoods. In this section we explore the affect of varying these parameters of the search. The algorithms tested were vertex relocation (VR) and random fence movement (RP).

Testing the variations of the starting configuration was performed as before with 100 iterations and a tabu list size of 25. The sampling rate of the neighborhood was 100%. The starting configurations were generated as a random permutation, CLF vertex order, and LF2 vertex order. Table XXXII shows the results of these two algorithms for each of the starting configuration on graphs of order 50 and edge density 0.2. Table XXXIII is for graphs of order 50 and edge density 0.5.

The sampling of neighborhoods was done using rates of 10%, 25%, 50%, and 100%. The other parameters of the problem were 100 iterations with a tabu list of size 25. The results of these tests are in Tables XXXIV and XXXV for graphs of order 50.

The best starting configuration for these tests proved to be CLF vertex order. The sampling rates show that the tabu list is of little use since the search will have a low probability of attempting to reverse the previous move. Thus, the Tabu Search with sampling the neighborhoods reduces to a steepest-descent/mildest-ascent search. In fact, any randomness can be seen to make the tabu list structure in these algorithms to be less effective. This is view is supported in Tables XXVII, XXVIII, XXIX, and XXX. The more deterministic the sampling of the neighborhood of the configuration then the tabu list becomes an effective device for limiting search in already explored regions in the configuration space. In addition, the sampling deteriorates the average number of colors used. Therefore the algorithms would ideally be run sampling 100%

of the neighborhoods. Unfortunately, the time increase limits the size of the graph which can be colored in a reasonable time.

Finally, an extended run using 1000 iterations was used to color graphs of order 50 of both edge densities. The algorithm RP was used to test the convergence for a large number of iterations. The testing sampled 100% of the neighborhood and used a tabu list of length 50. The tests averaged 9.52 colors for graphs of density 0.2 and 16.68 colors used for graphs of density 0.5. Unfortunately, the amount of time on the average for each graph of density 0.5 was approximately an hour (3191.33 seconds). Compared to 42 seconds for the VR algorithm to obtain 17.76, this was an increase in run-time of 76 times (a multiple larger than the order of the graph).

3. Testing Larger Graphs From the previous results, the algorithms should (when possible) use 100% sampling of the neighborhoods. Part of the goal of using these algorithms is to find a reasonable approach to improving the coloring of the composite graphs. Thus, the approach of this section will use sampling techniques for both the VR and RP algorithms of the previous section. Although this will likely cause the approximations to be of poorer quality, the times will be considerably smaller.

The first test involves graphs of order 100 and 200 with edge densities 0.2 and 0.5. The algorithms used will be VR and RP. For the graphs of order 100, the VR algorithm will use 400 iterations and 100% sampling of the neighborhood and algorithm RP will use 200 iterations for 0.2 edge density and 100 iterations for 0.5 edge density and both with a 25% sampling rate. The tabu list will be of length 50 with the starting configuration using CLF vertex order. The graphs of order 200 will use 500 iterations for the VR algorithm and a sampling rate of 50% while the RP algorithm will use 200 iterations and a sampling rate of 10%. The results of this test are given in Table XXXVI and XXXVII. Even though sampling demonstrated deterioration in previous tests, the RP algorithm performed better than the VR algorithm in terms of the number of colors used for graphs with edge density 0.5. The time used by RP was on the order of 5 times that of VR but the improvement

Table XXXII. Results from variations of starting configurations for $e = 0.2$

	VR			RP		
	Random	LF2	CLF	Random	LF2	CLF
iterations						
average	130.5	113.8	108.4	117.2	116.2	119.7
minimum	102.0	100.0	100.0	100.0	100.0	100.0
maximum	185.0	169.0	149.0	174.0	221.0	193.0
colors used						
average	10.40	10.28	9.96	10.48	10.24	10.08
minimum	9.00	9.00	8.00	9.00	8.00	8.00
maximum	12.00	12.00	12.00	13.00	13.00	12.00
time (secs)						
average	29.2	24.9	23.9	385.7	371.2	365.2
minimum	22.6	21.7	20.2	306.5	304.4	273.0
maximum	41.5	38.4	32.5	585.8	661.4	594.7
rejections						
average	—	—	—	4.1	4.2	5.0
minimum	—	—	—	0.0	0.0	0.0
maximum	—	—	—	19.0	27.0	37.0

Table XXXIII. Results from variations of starting configurations for $e = 0.5$

	VR			RP		
	Random	LF2	CLF	Random	LF2	CLF
iterations						
average	156.1	132.4	121.1	166.8	130.8	125.9
minimum	103.0	101.0	101.0	104.0	101.0	101.0
maximum	226.0	204.0	184.0	282.0	197.0	193.0
colors used						
average	18.40	18.12	17.76	19.04	18.28	17.88
minimum	15.00	15.00	15.00	16.00	15.00	15.00
maximum	21.00	21.00	20.00	25.00	23.00	20.00
time (secs)						
average	57.5	48.3	42.2	1049.0	757.1	732.8
minimum	36.7	34.4	34.6	578.7	553.6	522.6
maximum	80.3	78.0	67.4	2551.0	1181.0	1172.0
rejections						
average	—	—	—	9.8	5.6	5.9
minimum	—	—	—	0.0	2.0	1.0
maximum	—	—	—	95.0	26.0	32.0

Table XXXIV. Results from neighborhood sampling for $e = 0.2$

	VR				RP			
	10%	25%	50%	100%	10%	25%	50%	100%
iterations								
average	103.6	113.9	110.1	108.4	118.6	118.6	110.8	119.7
minimum	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
maximum	138.0	172.0	166.0	149.0	179.0	200.0	152.0	193.0
colors used								
average	10.60	10.24	10.12	9.96	10.40	10.16	10.24	10.08
minimum	9.00	8.00	9.00	8.00	8.00	8.00	8.00	8.00
maximum	12.00	12.00	12.00	12.00	14.00	14.00	13.00	12.00
time (secs)								
average	4.7	9.0	14.0	23.9	41.8	95.5	177.5	365.2
minimum	4.1	7.0	12.3	20.2	34.1	78.9	152.1	273.0
maximum	6.1	15.0	21.7	32.5	65.6	159.0	244.3	594.7
rejections								
average	—	—	—	—	0.4	0.6	0.6	5.0
minimum	—	—	—	—	0.0	0.0	0.0	0.0
maximum	—	—	—	—	6.0	10.0	8.0	37.0

Table XXXV. Results from neighborhood sampling for $e = 0.5$

	VR				RP			
	10%	25%	50%	100%	10%	25%	50%	100%
iterations								
average	104.3	121.4	119.9	121.1	120.9	121.0	134.9	125.9
minimum	100.0	100.0	100.0	101.0	100.0	101.0	102.0	101.0
maximum	128.0	193.0	181.0	184.0	185.0	196.0	262.0	193.0
colors used								
average	19.24	18.28	18.16	17.76	19.56	18.56	18.08	17.88
minimum	16.00	15.00	15.00	15.00	16.00	16.00	16.00	15.00
maximum	22.00	21.00	20.00	20.00	25.00	22.00	24.00	20.00
time (secs)								
average	7.1	14.0	23.7	42.2	72.9	179.6	383.6	732.8
minimum	6.1	10.9	16.4	34.6	53.6	133.5	269.2	522.6
maximum	8.3	23.4	36.2	67.4	112.1	305.8	763.3	1172.0
rejections								
average	—	—	—	—	0.5	0.5	1.1	5.9
minimum	—	—	—	—	0.0	0.0	0.0	1.0
maximum	—	—	—	—	7.0	6.0	12.0	32.0

in the average number of colors used made this increase reasonable.

The second tests were run on graphs of order 300 with both edge densities. The algorithm used was VR with the number of iteration two times the order of the graph. The neighborhood was sampled at 20% and has a tabu list of length 150. The results of these tests are in Table XXXVIII. The number of graphs colored in this test was reduced to 2.

F. CONCLUSIONS

The following summarizes the observations made in this chapter.

1. The Tabu Search algorithm incorporating CR moves was the best on graphs of order 50 with edge density 0.2, producing a 9.92 average for colors used. The algorithm did not perform as well on graphs with edge density 0.5.

2. The VR algorithm's run-times were the smallest of all algorithms tested. Also, for small graphs the VR algorithm's performance was the best overall.

3. The starting configuration with a CLF vertex order consistently provided the best approximation to the solution. The number of iterations indicate that on the average the local optimum to CLF was the solution found, but the maximum number of iterations indicates that some graph found a better solution after moving away from the CLF order. For instance, in Table XXXIII the maximum number of iterations used by some instance with edge density 0.5 was 193 when CLF order was used. Thus, 93 steps were taken before the best solution was found.

4. Sampling has adverse effects on the quality of the solution and the usefulness of the tabu list structure. The number of move rejections due to tabu status is effectively 0 for all sampling rates of 50% or less that were tested. It was also observed that for two algorithms and two different densities that the average number of colors, maximum number of colors, and minimum number of colors assigned decreases with the increase in sampling rate.

Table XXXVI. Results of Tabu Search for graphs $(100, *, TPOI(1), 25)$

	$e = 0.2$		$e = 0.5$	
	VR	RP	VR	RP
iterations				
average	420.6	247.0	464.7	240.9
minimum	400.0	200.0	402.0	201.0
maximum	726.0	471.0	752.0	454.0
colors used				
average	15.48	15.60	29.68	29.20
minimum	14.00	14.00	27.00	27.00
maximum	17.00	20.00	32.00	32.00
time (secs)				
average	830.1	4739.0	1638.0	7282.0
minimum	701.9	3533.0	1332.0	5342.2
maximum	1444.0	8987.0	2626.0	11037.4

Table XXXVII. Results of Tabu Search for graphs $(200, *, TPOI(1), 25)$

	$e = 0.2$		$e = 0.5$	
	VR	RP	VR	RP
iterations				
average	519.2	253.1	464.7	267.4
minimum	501.0	200.0	402.0	201.0
maximum	615.0	380.0	752.0	404.0
colors used				
average	25.18	24.24	53.16	51.16
minimum	23.00	22.00	48.00	48.00
maximum	27.00	26.00	57.00	55.00
time (secs)				
average	7344.1	3319.6	13946.8	6090.0
minimum	6819.3	2672.1	14909.4	4598.0
maximum	8968.2	4681.0	20539.9	10336.6

Table XXXVIII. Results of Tabu Search for graphs (300, *, TPOI(1), 2)

	$e = 0.2$	$e = 0.5$
iterations		
average	607.5	640.0
minimum	605.0	619.0
maximum	610.0	661.0
colors used		
average	32.00	69.50
minimum	34.00	66.00
maximum	30.00	73.00
time (secs)		
average	9982.5	24951.0
minimum	9879.3	24082.0
maximum	10085.8	25819.9

5. In spite of the increase in iterations and the use of the entire neighborhood, the VR algorithm did not perform as well as the RP algorithm on graphs of order 100 and 200. This is most likely due to the fact that VR fixed one vertex that occurs in the swap while RP does not. Thus at times when VR makes a poor choice for the fixed vertex to move it must make a move. RP has (in a sense) two degrees of freedom with which it can choose a move. This added flexibility makes RP more robust than VR.

6. Although the analysis of the complexity of the algorithms assumed that the number of iterations was linear in terms of the order of the graph, the algorithms did not perform well on the larger graphs when the number of iterations was intentionally left "small" in order to reduce the run-time. This supports the theorems of Chapter V. The theorems on approximation imply that the number of iterations will be on the order of $O(2^{|V(G_c)|})$ for the composite graph G_c. Thus, in order to get within the probabilistic upper bounds the number of iterations must approach this exponential complexity in terms of the order of the graph.

VII. TABU SEARCH VERSUS HEURISTICS

In this chapter we compare the results of the heuristics of Chapter IV. with the previous chapter and summarize the findings. The theory of Chapter V. is shown to have impact on the behavior of Tabu Search.

Table XXXIX shows the best heuristic approximation, the best approximation obtained by Tabu Search and the probabilistic lower bound for graphs of order 50, 100, 200, and 300 with edge density 0.2. Table XL shows the same results for graphs with edge density 0.5. The table shows the heuristic approximation in the column marked with h and the Tabu Search approximation in the column marked ts. The column marked with a u contains the probabilistic upper bound found by Oakes [Oa90]. Figures 31 and 32 show these results graphically.

The following conclusions can be made from this comparison of the Tabu Search algorithms and the heuristics for CGCP.

1. The Tabu Search algorithms provide a better average for the number of colors used for graphs where the number of iterations was sufficient. The larger graphs (200 and above) where the run-times became unreasonable the Tabu Search algorithms did not perform as well as the CRLF algorithm. Given enough iterations, the author conjectures that the Tabu Search would provide better averages than even the CRLF algorithm.

2. Clearly, the heuristics have the advantage when time is a consideration since the graphs of order 50 can each be colored in less than 0.5 seconds while the tabu search required an average of 3191.33 seconds for each graph. For large graphs, the times prohibit using the algorithms for practical graph coloring. The VR algorithm can be used for graphs smaller than 100 to obtain both reasonable run-times and good results.

3. The Tabu Search can provide colorings that fall within the upper probabilistic bounds for these problems. Consistently achieving that goal will cost in increasing the run-times to the point where the algorithm becomes exponential

Table XXXIX. Best approximations and probabilistic upper bounds for $\chi(G_c)$

$e = 0.2$

order	u	h	ts
50	10.50	10.40	9.52
100	15.00	15.24	15.48
200	22.40	24.44	24.24
300	29.10	30.52	32.00

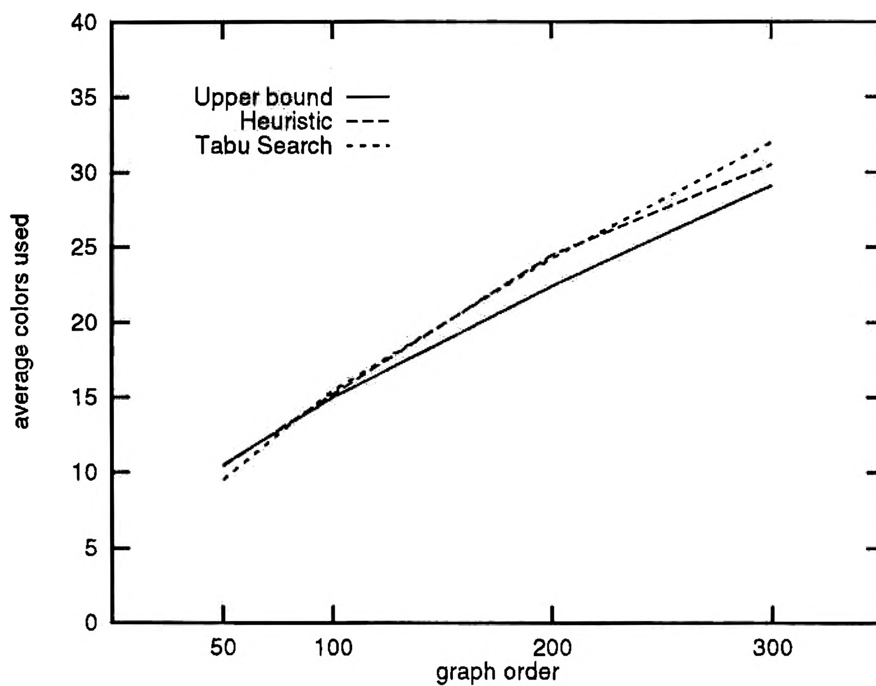


Figure 31. Approximations and upper bounds for graphs with $e = 0.2$

Table XL. Best approximations and probabilistic upper bounds for $\chi(G_c)$

$e = 0.5$

order	u	h	ts
50	18.60	18.40	16.68
100	29.30	30.20	29.20
200	47.50	51.80	51.16
300	64.10	68.40	69.50

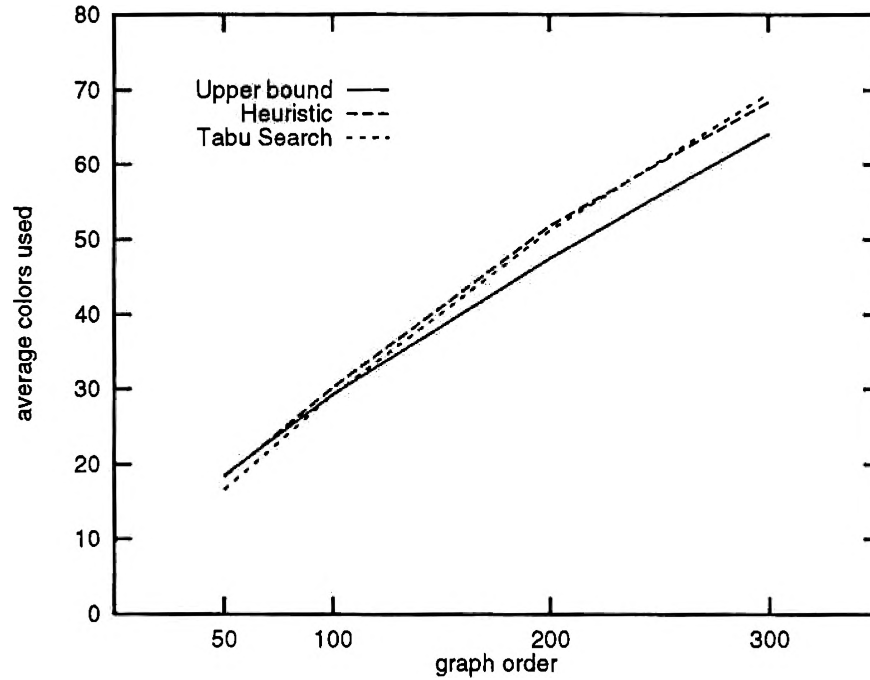


Figure 32. Approximations and upper bounds for graphs with $e = 0.5$

in nature (in terms of the order of the graph). This is highly impractical but the Tabu Search algorithms provide the option of balancing the quality of the solution with the cost (in time) of the solution.

The practical significance of these results is that the Tabu Search algorithms fill the gap between the heuristic coloring algorithms and the exact coloring algorithms.

VIII. SUMMARY AND FURTHER RESEARCH

Our goals were stated as follows:

1. Study the current heuristic algorithms for approximating the chromatic number for composite graphs and analyze the error involved in the approximation using probabilistic upper bounds.

2. Establish $\mathcal{NP}$-completeness results for the Composite Graph Coloring Problem. Show that approximation of the chromatic number with guaranteed upper bounds is as difficult as finding exact solutions.

3. Design, implement, and analyze Tabu Search algorithms for the Composite Graph Coloring Problem. Show that the Tabu Search can be used to improve the approximation of the chromatic number given by the heuristics.

Chapter IV. reimplemented the interchange method of Clementson and Elphick [CE83] for vertex sequential coloring algorithms. This reimplementation was used to color graphs of order up to 1000 using as a basis for the algorithms CLF and CDSatur. Error was analyzed with respect to probabilistic upper bounds and it was demonstrated that the error increased with the increase in the order of the graph. This provided empirical evidence that these heuristic algorithms were not able to guarantee an upper bound (neither absolute nor relative) for a random set of composite graphs.

The $\mathcal{NP}$-completeness results confirmed the outcomes of the error analysis in Chapter V. In this chapter, we proved that the Composite Graph Coloring Problem was $\mathcal{NP}$-hard. This was based on the proof that the associated decision problem for CGCP was $\mathcal{NP}$-complete. We further showed that CGCP was a number problem and also strongly $\mathcal{NP}$-complete. Finally, the problem of approximating the chromatic number with guarantees is as hard as the $\mathcal{NP}$-complete problem of exact coloring, that is, if we expect the chromatic number to be bounded above by an absolute or relative bound then the approximation algorithm will require exponential time in terms of the size of the graph.

The chapter on Tabu Search, Chapter VI., explored the possibility of modelling CGCP as a combinatorial optimization problem and searching the configuration space with Tabu Search. Tabu Search was shown to improve the heuristic approximations in Chapter VII. It was also noted that the times associated with Tabu Search should be expected since the configuration space is an instance of CGCP and the theory suggests that the times will increase.

Several options for further research are open that can be seen as extentions of this current work. The first involves the converse of Theorem 5.7.

Theorem If there is an ϵ-relative algorithm for CGCP with $\epsilon < N$ for $N \in \mathbb{Z}^+$ if ther is an ϵ-relative algorithm for SGCP with $\epsilon < N$.

Although it is not clear that the above is true, the proof or disproof of the above would allow the categorization of the SGCP and CGCP in terms of difficulty of approximation.

Tabu Search algorithms could be constructed using the color sequential algorithms as a basis for the search. The configuration space would consist of a partition of the vertex set. The partition would not follow that of Jenness and Gillett [JG91] but would be a partition that guaranteed that each set in the partition formed an independent set of vertices for the graph. Determining the cost involves ordering the partitions in some order and coloring the graph using the algorithm in Figure 30. The order of the vertices within the partition will not affect the coloring since they form an independent set. This follows (in form) the CRLF algorithm which consistently out performed the vertex sequential algorithms for graphs of larger orders.

Other work that can be explored is the comparison of Tabu Search algorithms with both Simulated Annealing and Genetic algorithms. Recently, Elmer [El93] implemented parallel versions of Simulated Annealing and Genetic algorithms for CGCP. The average colors for the algorithms implemented by Elmer were better than the results in this study but the times were also much larger than the times required using Tabu Search. Unfortunately, Elmer's study is not directly comparable to our results because the implementations used in Intel IPSC/860 with 16 processors. Comparison

of the convergence rates of Tabu Search, Simulated Annealing and Genetic algorithms when applied to CGCP would provide further information on the behavior of these techniques.

BIBLIOGRAPHY

[Be85] C. Berge, *Graphs* (Second revised edition). Elsevier Science Publishing Company, 1985.

[BM76] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications.* Elsevier Science Publishing Company, 1976.

[Br79] D. Brelaz, "New Methods to Color the Vertices of a Graph". *Communications of the ACM* 22 (1979), 251-256.

[CE83] A. T. Clementson and C. H. Elphick, "Approximate Colouring Algorithms for Composite Graphs". *Journal of the Operational Research Society* 34:6 (1983), 503-509.

[Ch71] N. Christofides, "An Algorithm for the Chromatic Number of a Graph". *The Computer Journal* 14:1 (1971), 38-39.

[Co71] S. A. Cook, "The complexity of theorem-proving procedures". *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 1971.

[DV92] F. Dammeyer and S. Voß, "Dynamic Tabu List Management Using the Reverse Elimination Method". *Annals of Operations Research* (1992).

[Da63] G. B. Dantzig, *Linear Programming and Extensions.* Princeton University Press, 1963.

[DT92] M. Dell'Amico and M. Trubian, "Applying Tabu Search to the Job-Shop Scheduling Problem". *Annals of Operations Research* (1992).

[El93] B. S. Elmer, *The Design, Analysis, and Implementation of Parallel Simulated Annealing and Parallel Genetic Algorithms for the Composite Graph Coloring Problem.* Ph.D. Dissertation, University of Missouri–Rolla, 1993.

[Fi90] C-N. Fiechter, "A Parallel Tabu Search Algorithm for Large Traveling Salesman Problems". Ecole Polytechnique Fédérale de Lausanne, ORWP 90/1.

[Fr90] C. Friden, A. Hertz, and D. de Werra, "TABARIS: An Exact Algorithm Based on Tabu Search For Finding A Maximum Independent Set In A Graph". *Computers Opns. Res.* 17:5 (1990), 437-445.

[GJ76] M. R. Garey and D. S. Johnson, "The Complexity of Near-Optimal Graph Coloring". *Journal of the ACM* 23:1 (1976), 43-49.

[GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GS92] M. Gendreau, A. Hertz and G. Laporte, "A Tabu Search Heuristic for the Vehicle Routing Problem". *Management Science*, (1991).

[GJ91] B. Gillett and J. Jenness, "A Parallel Color Sequential Algorithm for CGCP". *Eleventh European Congress on Operational Research* (April 1991).

[Gl89a] F. Glover, "Tabu Search—Part I". *ORSA Journal on Computing* 1:3 (1989), 190-206.

[Gl89b] F. Glover, "Tabu Search: A Tutorial". Center for Applied Artificial Intelligence, University of Colorado.

[Gl90] F. Glover, "Tabu Search—Part II". *ORSA Journal on Computing* 2:1 (1990), 4-32.

[Go89] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[HW87] A. Hertz and D. de Werra, "Using Tabu Search Techniques for Graph Coloring". *Computing* 39 (1987), 345-351.

[JG91] J. Jenness and B. Gillett, "A Parallel Tabu Search Strategy for Coloring Composite Graphs". *ORSA/TIMS Joint National Meeting* (May 1991)

[Jo74] D. S. Johnson, "Worst Case Behavior of Graph Coloring Algorithms,", F. Hoffman, R. A. Kingsley, R. B. Levow, R. C. Mullin and R. S. Thomas (eds.), *Proceedings of the Fifth Southeast Conference on Combinatorics, Graph Theory and Computing.* Utilitas Mathematica Publishing, 1974, 513-527.

[JA89] D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon, "Optimization By Simulated Annealing: An Experimental Evaluation, Part II (Graph Coloring and Number Partitioning)". *Operations Research* (1989).

[JA90] D. S. Johnson, C. R. Aragon, L. A. McGeoch and C. Schevon, "Optimization By Simulated Annealing: An Experimental Evaluation, Part I (Graph Partitioning)". *Operations Research* (1990).

[Ka72] R. M. Karp, "Reducibility among combinatorial problems," in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations.* Plenum Press, 1972, 85-103.

[Kh79] L. G. Khachian, "A Polynomial Algorithm in Linear Programming". *Soviet Mathematics Doklady* 20 (1979), 191-194.

[Kn73] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second edition. Addison-Wesley, 1973.

[Ko79] S. M. Korman, "The Graph Coloring Problem". N. Christofides, A. Mingozzi, P. Toth, and C. Sandi (eds.), *Combinatorial Optimization.* Wiley, 1979.

[LA88] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications.* D. Reidel Publishing, 1988.

[Le79] F. T. Leighton, "A Graph Coloring Algorithm for Large Scheduling Problems". *Journal of Research of the National Bureau of Standards*, 84:6 (1979), 489-506.

[LP81] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation.* Prentice Hall, 1981.

[MG89] M. Malek, M. Guruswamy, M. Pandya and H. Owens, "Serial and Parallel Simulated Annealing and Tabu Search Algorithms for the Traveling Salesman Problem". *Annals of Operations Research*, 21 (1989), 59-84.

[MR92] E. L. Mooney and R. L. Rardin, "Tabu Search for a Class of Scheduling Problems". *Annals of Operations Research*, (1992).

[Oa90] J. L. Oakes, *Algorithms and Probablistic Bounds for the Chromatic Number of Random Composite Graphs.* Ph.D. Dissertation, University of Missouri–Rolla, 1990.

[Os92] I. H. Osman, "Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem". *Annals of Operations Research*, (1992).

[PS82] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, 1982.

[Ro87] J. Roberts, *Heuristic Coloring Algorithms For the Composite Graph Coloring Problem.* Ph.D. Dissertation, University of Missouri–Rolla, 1987.

[ST92] F. Semet and E. Taillard, "Solving Real-Life Vehicle Routing Problems Efficiently Using Taboo Search". *Annals of Operations Research*, (1992).

[SW88] R. Sommehalder and S.C. van Westrhenen, *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility.* Addison-Wesley International Computer Science Series, 1988.

[Ta91] E. Taillard, "Robust Taboo Search for the Quadratic Assignment Problem". *Parallel Computing*, 17 (1991), 443-455.

[WH89] D. de Werra and A. Hertz, "Tabu Search Techniques: A Tutorial and an Application to Neural Networks". *OR Spektrum* 11 (1989), 131-141.

APPENDIX

SOURCE CODE LISTINGS

The source listings are provided in alphabetic order within this Appendix. A brief description is also provided for each file.

crlf.c Implements the CRLF algorithm for composite graphs.

```
/* CRLF.C
 *      Description: Composite Color Sequential Searching Shell:
 *                   This is used in conjunction with the heuristic methods.
 *                   This is to be called by using the SHELL.C main program.
 */

#include <time.h>
#include "misc.h"
#include "groupo.h"
#include "graph.h"

/* This shell is designed for the execution of the CRLF composite graph */
/* coloring heuristic.                                                  */

typedef struct
{
  int vertex; /* vertex number in adjacency matrix          */
  bool uncolored; /* false if already colored, else true        */
  int lowerb; /* current lower bound for first color        */
  int chroma; /* vertex chromaticity                        */
  int u1deg; /* U1 set adjacent chromatic degree           */
  int u2deg; /* U2 set adjacent chromatic degree           */
} sorttype;

/* NOTE: U1 is defined to be the set of uncolored nodes with a minimum  */
/* lower bound on the beginning color -- this set is always defined to  */
/* the set at the beginning of the sorted list of nodes in the index    */
/* which all have the same lower bound -- the one to color is always    */
/* first.                                                               */

sorttype *index;

local proc CreateIndex ARGS ((graph g, sorttype ** index));
int firstrule ARGS ((const void *a, const void *b));
int nextrule ARGS ((const void *a, const void *b));

#ifdef __STDC__
proc
crlf (GROUP * gp, graph * g, colors * k, word * chi, float *secs)
#else
proc
```

```
crlf (gp, g, k, chi, secs)
     GROUP *gp;
     graph *g;
     colors *k;
     word *chi;
     float *secs;
#endif
{
  int i, ii, jj;
  word n;

  CreateColors (g, k);
  memset (ColorsArr (*k), (byte) 0, sizeofColors (*k));

  /* start timing */
  *secs = (float) clock ();

  *chi = i = 1;
  CreateIndex (*g, &index);

#ifdef TRACE
  printf ("\nBeginning coloring:\n");
#endif
  while (index[0].uncolored)
    {
      while (index[0].lowerb == i)
{
  BegColor (ColorsOf (*g), index[0].vertex) = i;
  index[0].uncolored = false;
  *chi = max (*chi, EndColor (ColorsOf (*g), index[0].vertex));
#ifdef TRACE
  printf ("Coloring node %d starting with %d.\n", index[0].vertex,
  *chi);
#endif

  /* update index for sorting */
  for (ii = 1; index[ii].uncolored and ii < GraphOrder (*g); ii++)
    {
      index[ii].u1deg = index[ii].chroma;
      index[ii].u2deg = 0;
      if (Adjacent (*g, index[0].vertex, index[ii].vertex))
{
  index[ii].lowerb = max (index[ii].lowerb,
  EndColor (ColorsOf (*g),
    index[0].vertex) + 1);
}
    }
  for (ii = 1; index[ii].uncolored and ii < GraphOrder (*g); ii++)
    {
      for (jj = ii + 1; index[jj].uncolored and jj < GraphOrder (*g);
     jj++)
{
```

```
  if (Adjacent (*g, index[ii].vertex, index[jj].vertex))
    {
      if (index[jj].lowerb == i)
index[ii].u1deg += NodeChroma (ChromaOf (*g),
       index[jj].vertex);
      else
index[ii].u2deg += NodeChroma (ChromaOf (*g),
       index[jj].vertex);
      if (index[ii].lowerb == i)
index[jj].u1deg += NodeChroma (ChromaOf (*g),
       index[ii].vertex);
      else
index[jj].u2deg += NodeChroma (ChromaOf (*g),
       index[ii].vertex);
    }
}
    }
  qsort (index, GraphOrder (*g), sizeof (sorttype), nextrule);
}
      if (index[0].uncolored)
{
  i = index[0].lowerb;
  for (ii = 0; index[ii].uncolored and ii < GraphOrder (*g); ii++)
    {
      index[ii].u1deg = index[ii].chroma;
      index[ii].u2deg = 0;
      for (jj = ii + 1; index[jj].uncolored and jj < GraphOrder (*g);
   jj++)
{
  if (Adjacent (*g, index[ii].vertex, index[jj].vertex))
    {
      if (index[jj].lowerb == i)
index[ii].u1deg += NodeChroma (ChromaOf (*g),
       index[jj].vertex);
      if (index[ii].lowerb == i)
index[jj].u1deg += NodeChroma (ChromaOf (*g),
       index[ii].vertex);
    }
}
    }
  qsort (index, GraphOrder (*g), sizeof (sorttype), firstrule);
}
    }

  /* stop timing */
  *secs = ((float) clock ()- *secs) / (float) CLK_TCK;

  free (index);

  return;
}
```

```
#ifdef __STDC__
local proc
CreateIndex (graph g, sorttype ** index)
#else
local proc
CreateIndex (g, index)
     graph g;
     sorttype **index;
#endif
{
  int i, j;

  *index = (sorttype *) malloc (sizeof (sorttype) * GraphOrder (g));
  for (i = 0; i < GraphOrder (g); i++)
    {
      (*index)[i].vertex = i;
      (*index)[i].uncolored = true;
      (*index)[i].lowerb = 1;
      (*index)[i].chroma = NodeChroma (ChromaOf (g), i);
      (*index)[i].u1deg = (*index)[i].chroma;
      (*index)[i].u2deg = 0;
    }
  for (i = 0; i < GraphOrder (g); i++)
    for (j = i + 1; j < GraphOrder (g); j++)
      if (Adjacent (g, i, j))
{
  (*index)[i].u1deg += NodeChroma (ChromaOf (g), j);
  (*index)[j].u1deg += NodeChroma (ChromaOf (g), i);
}
  qsort (*index, GraphOrder (g), sizeof (sorttype), firstrule);

  return;
}

#ifdef __STDC__
int
firstrule (const void *a, const void *b)
#else
int
firstrule (a, b)
     const void *a, *b;
#endif
{
  if (((sorttype *) a)->uncolored == ((sorttype *) b)->uncolored)
    if (((sorttype *) a)->lowerb == ((sorttype *) b)->lowerb)
      if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
if (((sorttype *) a)->u1deg == ((sorttype *) b)->u1deg)
  return 0;
else
  return ((sorttype *) b)->u1deg - ((sorttype *) a)->u1deg;
      else
return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
```

```
    else
      return ((sorttype *) a)->lowerb - ((sorttype *) b)->lowerb;
  else
    return ((sorttype *) b)->uncolored - ((sorttype *) a)->uncolored;
}

#ifdef __STDC__
int
nextrule (const void *a, const void *b)
#else
int
nextrule (a, b)
     const void *a, *b;
#endif
{
  if (((sorttype *) a)->uncolored == ((sorttype *) b)->uncolored)
    if (((sorttype *) a)->lowerb == ((sorttype *) b)->lowerb)
      if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
if (((sorttype *) a)->u2deg == ((sorttype *) b)->u2deg)
  if (((sorttype *) a)->u1deg == ((sorttype *) b)->u1deg)
    return 0;
  else
    return ((sorttype *) a)->u1deg -
      ((sorttype *) b)->u1deg;
else
  return ((sorttype *) b)->u2deg - ((sorttype *) a)->u2deg;
      else
return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
    else
      return ((sorttype *) a)->lowerb - ((sorttype *) b)->lowerb;
  else
    return ((sorttype *) b)->uncolored - ((sorttype *) a)->uncolored;
}
```

cvss.c Implements the vertex sequential heuristic coloring algorithms for composite graphs.

```
/* CVSS.C
 *      Description: Composite Vertex Sequential Searching Shell: This is
 *                   used in conjunction with the heuristic methods. This
 *                   is to be called by using the SHELL.C main program.
 */

#include <time.h>
#include "misc.h"
#include "groupo.h"
#include "graph.h"

/* This shell is designed for the execution of several composite graph */
/* coloring heuristics:                                                 */
/* LF1, LF2, CLF, CSL, CDsatur, LF1I, CLFI, CDsaturI                    */
```

```
#define CMPRULE clfrule
/* define SORTINDEX for cdsaturrule only */
#define SORTINDEX
#define INTERCHANGE

#ifdef SORTINDEX
#define NEXTNODE 0
#else
#define NEXTNODE i
#endif

typedef struct
{
  int vertex;                  /* vertex number in adjacency matrix    */
  bool uncolored;              /* false if already colored, else true  */
  int cdeg;                    /* colored degree                       */
  int chroma;                  /* vertex chromaticity                  */
  int ucdeg;                   /* uncolored adjacent chromatic degree  */
  int ucadj;                   /* number of adjacent uncolored nodes   */
} sorttype;

sorttype *index;
int *co;
word maxco;
int maxch;
colors best;

#ifdef INTERCHANGE
int *p, *q;

#endif

local word nextcolor ARGS ((graph g, int i));
local proc CreateIndex ARGS ((graph g, sorttype ** index));
local proc AdjustIndex ARGS ((graph g, int i));
int NodeIndex ARGS ((GROUP * gp, graph g, sorttype * index, int node));
int lf1rule ARGS ((const void *a, const void *b));
int lf2rule ARGS ((const void *a, const void *b));
int clfrule ARGS ((const void *a, const void *b));
int cslrule ARGS ((const void *a, const void *b));
int cdsaturrule ARGS ((const void *a, const void *b));

#ifdef __STDC__
proc
cvss (GROUP * gp, graph * g, colors * k, word * chi, float *secs)
#else
proc
cvss (gp, g, k, chi, secs)
     GROUP *gp;
     graph *g;
     colors *k;
     word *chi;
```

```
    float *secs;

#endif
{
  int i, j, ii, jj, kk, ll;
  word n;
  sorttype temp;

  CreateColors (g, k);
  memset (ColorsArr (*k), (byte) 0, sizeofColors (*k));
  /* initialize */
  maxco = 0;
  maxch = 0;
  for (i = 0; i < GraphOrder (*g); i++)
    {
      maxco += NodeChroma (ChromaOf (*g), i);
      if (NodeChroma (ChromaOf (*g), i) > maxch)
        maxch = NodeChroma (ChromaOf (*g), i);
    }
  /* used when calculating cdeg */
  co = (int *) malloc ((maxco + maxch) * sizeof (int));

#ifdef INTERCHANGE
  p = (int *) malloc ((maxco + maxch) * sizeof (int));
  q = (int *) malloc ((maxco + maxch) * sizeof (int));

#endif

  /* start timing */
  *secs = (float) clock ();

  *chi = 0;
  CreateIndex (*g, &index);
#ifdef SORTINDEX
  /* move the next candidate to position 0 in the index */
  jj = 0;
  for (ii = 1; ii < GraphOrder (*g); ii++)
    if (CMPRULE (&index[jj], &index[ii]) > 0)
      jj = ii;
  if (jj != 0)
    {
      temp = index[jj];
      index[jj] = index[0];
      index[0] = temp;
    }
#else
  qsort (index, GraphOrder (*g), sizeof (sorttype), CMPRULE);
#endif
#ifdef TRACE
  printf ("\nBeginning coloring:\n");
#endif
  for (i = 0; i < GraphOrder (*g); i++)
```

```
    {
      j = NEXTNODE;
#ifdef TRACE
      printf ("Coloring node %d.\n", index[j].vertex);
#endif
#ifdef INTERCHANGE
      n = nextcolor (*g, j);
      if (n + NodeChroma (ChromaOf (*g), index[j].vertex) - 1 > *chi)
        {
          /* initialize p, q */
          for (ii = 1; ii < n; ii++)
            {
              /* stores color-vertex pairs; ii is color, p[ii] is index */
              p[ii] = -1;
              /* stores mincolor for each vertex in p[ii] under interchng */
              q[ii] = 0;
            }
          p[0] = 0;             /* number of candidates for interchange */
          /* calculate p, q */
          for (ii = 1; ii < n; ii++)
            {
              for (jj = 0; jj < GraphOrder (*g); jj++)
                if (Adjacent (*g, index[j].vertex, index[jj].vertex)
                    and not index[jj].uncolored)
                  if ((ii >= BegColor (ColorsOf (*g), index[jj].vertex) and
                       ii <= EndColor (ColorsOf (*g), index[jj].vertex))
                  or (BegColor (ColorsOf (*g), index[jj].vertex) >= ii and
                      BegColor (ColorsOf (*g), index[jj].vertex)
                      < ii + NodeChroma (ChromaOf (*g), index[j].vertex)))
                    if (p[ii] == -1)
                      {         /* first time found */
                        p[0]++;
                        p[ii] = jj;
                      }
                    else
                      {         /* not unique, drop and continue */
                        p[0]--;
                        p[ii] = -1;
                        break;
                      }
              if (p[ii] != -1)
                {               /* found unique vertex, calculate q[ii] */
                  BegColor (ColorsOf (*g), index[j].vertex) = ii;
                  index[j].uncolored = false;
                  jj = BegColor (ColorsOf (*g), index[p[ii]].vertex);
                  kk = nextcolor (*g, p[ii]);
                  index[j].uncolored = true;
                  BegColor (ColorsOf (*g), index[p[ii]].vertex) = jj;
                  if (kk + NodeChroma (ChromaOf (*g), index[j].vertex) <
                      n + NodeChroma (ChromaOf (*g), index[j].vertex))
                    q[ii] = kk;
                  else
```

```
          {             /* does not improve the current color */
            p[0]--;
            p[ii] = -1;
          }
      }
  }
if (p[0] != 0)
  {                     /* p[0] is 0 if no candidates */
    kk = n;
    ll = 0;
    for (ii = 1; ii < n; ii++)
      if (p[ii] != -1)
        {
          jj = max (ii + NodeChroma (ChromaOf (*g),
                                    index[j].vertex) - 1,
                    q[ii] + NodeChroma (ChromaOf (*g),
                                       index[p[ii]].vertex) - 1);
          if (jj <= kk)
            {
              kk = jj;
              ll = ii;
            }
        }
    if (ll != 0)
      {
        n = ll;
        BegColor (ColorsOf (*g), index[p[ll]].vertex) = q[ll];
        /* update cdeg for this new color at this vertex */
        ii = p[ll];
        for (jj = 0; jj < GraphOrder (*g); jj++)
          if (Adjacent (*g, index[ii].vertex, index[jj].vertex))
            {
              memset (co, (byte) 0, sizeof (int) * maxco);

              index[jj].cdeg = 0;
              for (kk = 0; kk < GraphOrder (*g); kk++)
                if (Adjacent (*g, index[jj].vertex,
                              index[kk].vertex)
                    and not index[kk].uncolored)
                  for (ll = BegColor (ColorsOf (*g),
                                      index[kk].vertex);
                       ll <= EndColor (ColorsOf (*g),
                                       index[kk].vertex);
                       ll++)
                    if (co[ll] == 0)
                      {
                        co[ll] = 1;
                        index[jj].cdeg += 1;
                      }
            }
      }
  }
```

```
        }
      BegColor (ColorsOf (*g), index[j].vertex) = n;
#else
      BegColor (ColorsOf (*g), index[j].vertex) = nextcolor (*g, j);
#endif
      index[j].uncolored = false;

      if (*chi < EndColor (ColorsOf (*g), index[j].vertex))
        *chi = EndColor (ColorsOf (*g), index[j].vertex);

#ifdef SORTINDEX
      AdjustIndex (*g, j);
      /* swap the just colored vertex to the bottom of the index */
      temp = index[0];
      index[0] = index[GraphOrder (*g) - i - 1];
      index[GraphOrder (*g) - i - 1] = temp;
      /* move the next candidate to position 0 in the index */
      jj = 0;
      for (ii = 1; ii < GraphOrder (*g) - i - 1; ii++)
        if (CMPRULE (&index[jj], &index[ii]) > 0)
          jj = ii;
      if (jj != 0)
        {
          temp = index[jj];
          index[jj] = index[0];
          index[0] = temp;
        }
#endif
    }

  /* stop timing */
  *secs = ((float) clock ()- *secs) / (float) CLK_TCK;

  free (index);
  free (co);
#ifdef INTERCHANGE
  free (p);
  free (q);
#endif

  return;
}

#ifdef __STDC__
local word
nextcolor (graph g, int i)
#else
local word
nextcolor (g, i)
     graph g;
     int i;
```

```
#endif
{
  int c, j;

  c = BegColor (ColorsOf (g), index[i].vertex) + 1;
  loop
  {
  again:
    for (j = 0; j < GraphOrder (g); j++)
      if (Adjacent (g, index[i].vertex, index[j].vertex) and
          not index[j].uncolored and
          ((c >= BegColor (ColorsOf (g), index[j].vertex) and
            c <= EndColor (ColorsOf (g), index[j].vertex))
           or (BegColor (ColorsOf (g), index[j].vertex) >= c and
               BegColor (ColorsOf (g), index[j].vertex)
               < c + NodeChroma (ChromaOf (g), index[i].vertex))))
        {
          c = max (c, EndColor (ColorsOf (g), index[j].vertex) + 1);
          goto again;
        }
    return c;
  }
}

#ifdef __STDC__
local proc
CreateIndex (graph g, sorttype ** index)
#else
local proc
CreateIndex (g, index)
     graph g;
     sorttype **index;

#endif
{
  int i, j;

  *index = (sorttype *) malloc (sizeof (sorttype) * GraphOrder (g));
  for (i = 0; i < GraphOrder (g); i++)
    {
      (*index)[i].vertex = i;
      (*index)[i].uncolored = true;
      (*index)[i].cdeg = 0;
      (*index)[i].chroma = NodeChroma (ChromaOf (g), i);
      (*index)[i].ucdeg = 0;
      (*index)[i].ucadj = 0;
    }
  for (i = 0; i < GraphOrder (g); i++)
    for (j = i + 1; j < GraphOrder (g); j++)
      if (Adjacent (g, i, j))
        {
          (*index)[i].ucdeg += NodeChroma (ChromaOf (g), j);
```

```
          (*index)[j].ucdeg += NodeChroma (ChromaOf (g), i);
          (*index)[i].ucadj += 1;
          (*index)[j].ucadj += 1;
        }
  /* qsort(*index, GraphOrder(g), sizeof(sorttype), CMPRULE); */
  return;
}

#ifdef SORTINDEX
#ifdef __STDC__
local proc
AdjustIndex (graph g, int i)
#else
local proc
AdjustIndex (g, i)
     graph g;
     int i;

#endif
{
  int j, k, m;

  for (j = 0; j < GraphOrder (g); j++)
    {
      if (Adjacent (g, index[i].vertex, index[j].vertex))
        {
          if (index[i].uncolored)
            {
              index[j].ucdeg += index[i].chroma;
              index[j].ucadj += 1;
            }
          else
            {
              index[j].ucdeg -= index[i].chroma;
              index[j].ucadj -= 1;
            }
          memset (co, (byte) 0, sizeof (int) * maxco);

          index[j].cdeg = 0;
          for (k = 0; k < GraphOrder (g); k++)
            if (Adjacent (g, index[j].vertex, index[k].vertex) and
                not index[k].uncolored)
              for (m = BegColor (ColorsOf (g), index[k].vertex);
                   m <= EndColor (ColorsOf (g), index[k].vertex); m++)
                if (co[m] == 0)
                  {
                    co[m] = 1;
                    index[j].cdeg += 1;
                  }
        }
    }
  /* qsort(*index, GraphOrder(g), sizeof(sorttype), CMPRULE); */
```

```
  return;
}

#endif

#ifdef __STDC__
int
NodeIndex (GROUP * gp, graph g, sorttype * index, int node)
#else
int
NodeIndex (gp, g, index, node)
     GROUP *gp;
     graph g;
     sorttype *index;
     int node;

#endif
{
  int i;

  i = 0;
  while (index[i].vertex != node and i < GraphOrder (g))
    i++;
  if (index[i].vertex != node)
    {
      gprintf (gp, "\nERROR: vertex has no index!\n");
      exit (1);
    }
  return i;
}

/* #if CMPRULE == lf1rule */
#ifdef __STDC__
int
lf1rule (const void *a, const void *b)
#else
int
lf1rule (a, b)
     const void *a, *b;

#endif
{
  if (((sorttype *) a)->uncolored == ((sorttype *) b)->uncolored)
    if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
      if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
        return 0;
      else
        return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
    else
      return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
  else
    return ((sorttype *) b)->uncolored - ((sorttype *) a)->uncolored;
```

```
}

/* #endif */

/* #if CMPRULE == lf2rule */
#ifdef __STDC__
int
lf2rule (const void *a, const void *b)
#else
int
lf2rule (a, b)
     const void *a, *b;

#endif
{
  if (((sorttype *) a)->uncolored == ((sorttype *) b)->uncolored)
    if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
      if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
        return 0;
      else
        return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
    else
      return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
  else
    return ((sorttype *) b)->uncolored - ((sorttype *) a)->uncolored;
}

/* #endif */

/* #if CMPRULE == clfrule */
#ifdef __STDC__
int
clfrule (const void *a, const void *b)
#else
int
clfrule (a, b)
     const void *a, *b;

#endif
{
  if (((sorttype *) a)->uncolored == ((sorttype *) b)->uncolored)
    if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
      if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
        if (((sorttype *) a)->ucadj == ((sorttype *) b)->ucadj)
          return 0;
        else
          return ((sorttype *) b)->ucadj - ((sorttype *) a)->ucadj;
      else
        return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
    else
      return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
  else
```

```
    return ((sorttype *) b)->uncolored - ((sorttype *) a)->uncolored;
}

/* #endif */

/* #if CMPRULE == cslrule */
#ifdef __STDC__
int
cslrule (const void *a, const void *b)
#else
int
cslrule (a, b)
     const void *a, *b;

#endif
{
  if (((sorttype *) a)->uncolored == ((sorttype *) b)->uncolored)
    if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
      if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
        if (((sorttype *) a)->ucadj == ((sorttype *) b)->ucadj)
          return 0;
        else
          return ((sorttype *) a)->ucadj - ((sorttype *) b)->ucadj;
      else
        return ((sorttype *) a)->ucdeg - ((sorttype *) b)->ucdeg;
    else
      return ((sorttype *) a)->chroma - ((sorttype *) b)->chroma;
  else
    return ((sorttype *) b)->uncolored - ((sorttype *) a)->uncolored;
}

/* #endif */

/* #if CMPRULE == cdsaturrule */
#ifdef __STDC__
int
cdsaturrule (const void *a, const void *b)
#else
int
cdsaturrule (a, b)
     const void *a, *b;

#endif
{
  if (((sorttype *) a)->uncolored == ((sorttype *) b)->uncolored)
    if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
      if (((sorttype *) a)->cdeg == ((sorttype *) b)->cdeg)
        if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
          if (((sorttype *) a)->ucadj == ((sorttype *) b)->ucadj)
            return 0;
          else
            return ((sorttype *) b)->ucadj - ((sorttype *) a)->ucadj;
```

```
        else
          return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
      else
        return ((sorttype *) b)->cdeg - ((sorttype *) a)->cdeg;
    else
      return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
  else
    return ((sorttype *) b)->uncolored - ((sorttype *) a)->uncolored;
}

/* #endif */
```

gengraph.pas Oakes [Oa90] program to generate test graphs with slightly modified output.

```
{$N+,E-,M 65520,0,655360}
program GenGraph;
uses CRT;
const
  MaxNumOfVert = 1000;
type
  String11 = string[11];  VertexType = 1..MaxNumOfVert;
  VertexSetType = set of 1..250;  AdjListPtrType = ^AdjListType;
  AdjListType = record
                  VertSetArray : array[1..4] of VertexSetType;
                end;
var
  GraphType : char;  Seed : longint;
  Count,NumOfVert,EdgeCnt,NumOfGraphs : word;
  Param,EdgeDensity : double;  FileName,Extension : String11;
  Vert, Chrom : array[1..MaxNumOfVert] of word;
  AdjVertArray : array[VertexType] of AdjListPtrType;
  DataOut,Out : text;

{*** Generate power with integer exponent ***}
function Power (Base : double; Exponent : integer) : double;
  var
    I : word; Result : double;
  begin
    Result := 1.0;
    for I := 1 to Exponent do
      Result := Result * Base;
    Power := Result;
  end;

{*** Generate N Factorial ***}
function Fact (N : integer) : integer;
  var
    I,Result : word;
  begin
    Result := 1;
    for I := 1 to N do
```

```
      Result := Result * I;
    Fact := Result;
  end;

{*** Random number generator ***}
function Random : double;
  const
    A : longint = 16807;   M : longint = 2147483647;
    Q : longint = 127773;  R : longint = 2836;
  var
    Low,High,Test : longint;
  begin
    High := Seed div Q;  Low := Seed mod Q;
    Test := A*Low - R*High;
    if (Test>0) then
      Seed := Test
    else
      Seed := Test + M;
    Random := Seed/M;
  end;

{*** Initialize variables ***}
procedure Initialize;
  var
    Vert : VertexType;  AdjListPtr : AdjListPtrType;
  begin
    assign (DataOut, FileName + '.dat');
    rewrite (DataOut);
    Param := 1.0;
    Seed := 493544361;
    for Vert := 1 to NumOfVert do
      begin
        new(AdjListPtr);
        AdjVertArray[Vert] := AdjListPtr;
      end;
    writeln (DataOut,'// This group of graphs was created by GenGraph:');
    writeln (DataOut,'//      GRAPHS  : ',NumOfGraphs);
    writeln (DataOut,'//      ORDER   : ',NumOfVert);
    writeln (DataOut,'//      DENSITY : ',EdgeDensity:0:2);
    writeln (DataOut,'//      PDF     : POISSON');
    writeln (DataOut,'//      SEED    : ',493544361);
    writeln (DataOut);
    writeln (DataOut,'BEGIN GROUP 0');
  end;

{*** Cleanup on exit ***}
procedure Cleanup;
  var
    Vert : VertexType;
  begin
    writeln (DataOut,'END GROUP');
    for Vert := 1 to NumOfVert do
```

```
      dispose(AdjVertArray[Vert]);
    close (DataOut);
  end;

{*** Generate vertex chromaticities ***}
procedure GenChrom;
  var
    I,J : word;  CumPDF : double;
  begin
    for I := 1 to NumOfVert do
      begin
        CumPDF := Param/(Exp(Param)-1);
        J := 1;
        while (CumPDF<Random+0.06) do
          begin
            J := J + 1;
            CumPDF := CumPDF + Power(Param,J)/((Exp(Param)-1)*Fact(j));
          end;
        Chrom[I] := J;
      end;
  end;

{*** Add vertex to adjacency set structure ***}
procedure AddNewAdjVert (Vert1,Vert2 : VertexType);
  var
    I : word;  Vert : VertexType;
  begin
    case Vert2 of
      1..250     : begin
                    I := 1;  Vert := Vert2;
                   end;
      251..500   : begin
                    I := 2;  Vert := Vert2 - 250;
                   end;
      501..750   : begin
                    I := 3;  Vert := Vert2 - 500;
                   end;
      751..1000  : begin
                    I := 4;  Vert := Vert2 - 750;
                   end;
    end;
    AdjVertArray[Vert1]^.VertSetArray[I] :=
      AdjVertArray[Vert1]^.VertSetArray[I] + [Vert];
  end;

{*** Generate graph edges ***}
procedure GenEdges;
  var
    Vert,Vert1,Vert2 : word;  Rand : double;
  begin
    for Vert := 1 to NumOfVert do
      begin
```

```
        AdjVertArray[Vert]^.VertSetArray[1] := [];
        AdjVertArray[Vert]^.VertSetArray[2] := [];
        AdjVertArray[Vert]^.VertSetArray[3] := [];
        AdjVertArray[Vert]^.VertSetArray[4] := [];
      end;
    for Vert1 := 1 to NumOfVert - 1 do
        for Vert2 := Vert1 + 1 to NumOfVert do
            if (EdgeDensity>=Random) then
                AddNewAdjVert (Vert1,Vert2);
  end;

{*** Check if Vert1 is adjacent to Vert2 ***}
function IsIn (Vert1,Vert2 : VertexType) : boolean;
  var
    I : word;
    Vert : VertexType;
  begin
    case Vert1 of
      1..250     : begin
                     I := 1;  Vert := Vert1;
                   end;
      251..500   : begin
                     I := 2;  Vert := Vert1 - 250;
                   end;
      501..750   : begin
                     I := 3;  Vert := Vert1 - 500;
                   end;
      751..1000  : begin
                     I := 4;  Vert := Vert1 - 750;
                   end;
    end;
    IsIn := Vert in AdjVertArray[Vert2]^.VertSetArray[I];
  end;

{*** Output file defining random graph ***}
procedure OutputGraph(I : integer);
  var
    Vert1,Vert2,J : word;
  begin
    writeln (DataOut,'BEGIN GRAPH ',I,' NODES ',NumOfVert);
    writeln (DataOut,'ADJACENCY MATRIX');
    for Vert1 := 1 to NumOfVert do
      begin
        (* for J := 1 to Vert1 - 1 do
          write (DataOut,'  '); *)
        write (DataOut,' 0');
        for Vert2 := Vert1 + 1 to NumOfVert do
          if (IsIn(Vert2,Vert1)) then
            write (DataOut,' 1')
          else
            write (DataOut,' 0');
        writeln (DataOut);
```

```
      end;
    writeln (DataOut,'CHROMATICITY VECTOR');
    for Vert1 := 1 to NumOfVert do
      write (DataOut,Chrom[Vert1]:3);
    writeln (DataOut);
    writeln (DataOut,'END GRAPH');
  end;

begin
  write ('Enter the number of vertices > ');
  readln (NumOfVert);
  write ('Enter the edge density      > ');
  readln (EdgeDensity);
  write ('Enter the number of graphs   > ');
  readln (NumOfGraphs);
  write ('Enter the output file        > ');
  readln (FileName);
  Initialize;
  for Count := 1 to NumOfGraphs do
    begin
      GenChrom;
      GenEdges;
      OutputGraph (Count);
    end;
  Cleanup;
end.
```

graph.c Implements the graph abstract data type.

```
/*
        ADT Graph C Package
        $Log: graph.c $
 * Revision 1.8  1992/09/12  14:47:40  Jenness
 * Changed the Copy functions...programmer must deallocate.
 *
 * Revision 1.7  1992/08/21  23:46:03  Jenness
 * Fixed a bug in PackSets
 *
 * Revision 1.6  1992/08/15  23:25:31  Jenness
 * reorganised the graph and colors data structures, changed
 * the CreateColors function parameters and some other code
 * to reflect this change
 * also made some cosmetic changes
 *
 * Revision 1.5  1992/08/01  21:18:37  Jenness
 * Output routines will write files readable by "gverify"
 * Added routines to track colorings of a composite graph
 *
 * Revision 1.4  1992/08/01  14:41:42  Jenness
 * cosmetic changes
 *
 * Revision 1.3  1992/07/24  19:22:04  Jenness
```

```
 * Added the copy functions and general fixups
 *
 * Revision 1.2  1992/07/22  21:16:16  Jenness
 * Added new random number generators
 *
 * Revision 1.1  1992/07/18  23:35:40  Jenness
 * Initial revision
 *
*/
static char rcsid_GRAPH_C[]=
    "$Id: graph.c 1.8 1992/09/12 14:47:40 Jenness Exp $";

#include <stdio.h> /* fprintf */
#ifdef __STDC__
#include <stdlib.h> /* malloc, rand */
#include <assert.h> /* assert */
#endif
#include <string.h> /* memcpy */
#include "misc.h"
#include "random.h"
#include "groupo.h"
#include "graph.h"

/* some global variables used when manipulating graphs */
bool EchoDetails = false, KeepColors = false;

/* local function declarations */
local proc PrimeISets ARGS ((graph * g, sets * s));
local proc CreateISets ARGS ((graph * g, byte * s, sets * t));

/* exported functions */
#ifdef __STDC__
proc
CreateGraph (graph * g, word n)
#else
proc
CreateGraph (g, n)
     graph *g;
     word n;
#endif
{
#ifndef NDEBUG
  printf ("\nCreating a graph of order %d.", n);
#endif
  GraphOrder (*g) = n;
  GraphAdjMat (*g) = mallocGraph (*g);
  if (GraphAdjMat (*g) == NULL)
    error (MEMFAIL, "CreateGraph");
  GraphDensity (*g) = 0.0;
  ChromaPtr (*g) = NULL;
  ColorsPtr (*g) = NULL;
  return;
```

```
}

#ifdef __STDC__
proc
GenerateEdges (graph * g, float d)
#else
proc
GenerateEdges (g, d)
     graph *g;
     float d;
#endif
{
  int i, j;

#ifndef NDEBUG
  printf ("\nGenerating edges for graph with density %0.2f.", d);
#endif
  assert (GraphAdjMat (*g) != NULL);
  GraphDensity (*g) = d;
  for (i = 0; i < GraphOrder (*g); i++)
    {
      EdgeOfGraph (*g, i, i) = 0;
      for (j = i + 1; j < GraphOrder (*g); j++)
EdgeOfGraph (*g, i, j) = ((randf ()< d) ? 1 : 0);
    }
  return;
}

#ifdef __STDC__
proc
CreateChroma (graph * g, chroma * c)
#else
proc
CreateChroma (g, c)
     graph *g;
     chroma *c;
#endif
{
#ifndef NDEBUG
  printf ("\nCreating a chromaticity array of size %d.", GraphOrder (*g));
#endif
  GraphPtr (*c) = g;
  ChromaPtr (*g) = c;
  ChromaArr (*c) = mallocChroma (*c);
  if (ChromaArr (*c) == NULL)
    error (MEMFAIL, "CreateChroma");
  return;
}

#ifdef __STDC__
proc
GenerateChroma (chroma * c, word (*f) (void))
```

```
#else
proc
GenerateChroma (c, f)
     chroma *c;
word (*f) (void);
#endif
{
  int i;

#ifndef NDEBUG
  printf ("\nGenerating values for the chromaticity array ");
#endif
  assert (ChromaArr (*c) != NULL);
  for (i = 0; i < GraphOrder (GraphOf (*c)); i++)
    NodeChroma (*c, i) = (*f) ();
  return;
}

word
TruncPoisson (void)
{
  float random;

#ifndef NDEBUG
  static bool printed = false;
  if (not printed)
    printf ("using the truncated poisson distribution.");
  printed = true;
#endif
  random = randf ();
  if (random <= 0.582)
    return (1);
  else if (random <= 0.873)
    return (2);
  else if (random <= 0.970)
    return (3);
  else if (random <= 0.994)
    return (4);
  else if (random <= 0.999)
    return (5);
  else
    return (6);
}

word
Fixed60_40 (void)
{
  float random;

#ifndef NDEBUG
  static bool printed = false;
  if (not printed)
```

```
    printf ("using the fixed 60-40 distribution.");
  printed = true;
#endif
  random = randf ();
  if (random <= 0.60)
    return (1);
  else
    return (2);
}

word
Fixed75_25 (void)
{
  float random;

#ifndef NDEBUG
  static bool printed = false;
  if (not printed)
    printf ("using the fixed 75-25 distribution.");
  printed = true;
#endif
  random = randf ();
  if (random <= 0.75)
    return (1);
  else
    return (2);
}

word
Uniform4 (void)
{
  float random;

#ifndef NDEBUG
  static bool printed = false;
  if (not printed)
    printf ("using the uniform distribution.");
  printed = true;
#endif
  random = randf ();
  if (random <= 0.25)
    return (1);
  else if (random <= 0.50)
    return (2);
  else if (random <= 0.75)
    return (3);
  else
    return (4);
}

word
Uniform3 (void)
```

```
{
  float random;

#ifndef NDEBUG
  static bool printed = false;
  if (not printed)
    printf ("using the uniform distribution.");
  printed = true;
#endif
  random = randf ();
  if (random <= 0.33)
    return (1);
  else if (random <= 0.66)
    return (2);
  else
    return (3);
}

char *tok_pdistr[]=
{
  "POISSON", "75-25", "60-40", "UNIFORM-4", "UNIFORM-3"
};

word (*(fp_pdistr[]))(void) =
{
  TruncPoisson, Fixed75_25, Fixed60_40, Uniform4, Uniform3
};

const int num_pdistr = sizeof (fp_pdistr) / sizeof (fp_pdistr[0]);

#ifdef __STDC__
proc
CreateColors (graph * g, colors * k)
#else
proc
CreateColors (g, k)
     graph *g;
     colors *k;
#endif
{
#ifndef NDEBUG
  printf ("\nCreating coloring array of size %d.", GraphOrder (*g));
#endif
  GraphPtr (*k) = g;
  ColorsPtr (*g) = k;
  ColorsArr (*k) = mallocColors (*k);
  if (ColorsArr (*k) == NULL)
    error (MEMFAIL, "CreateColors");
  return;
}

#ifdef __STDC__
```

```
proc
GenerateISets (graph * g, sets * s)
#else
proc
GenerateISets (g, s)
     graph *g;
     sets *s;
#endif
{
  int i;
  sets queue, buffer;

#ifndef NDEBUG
  printf ("\nGenerating independent sets.");
#endif
  PrimeISets (g, &queue);
#ifndef NDEBUG
  printf ("\nThe algorithm is primed with the following sets.");
  PrintSets (queue);
#endif

  PtrToSets (*s) = NULL;
  CardOfSets (*s) = 0;
  GraphPtr (*s) = g;

  for (i = 0; i < CardOfSets (queue); i++)
    {
      CreateISets (g, PtrToSetN (queue, i), &buffer);
#ifndef NDEBUG
      printf ("\nUsing set %d a total of %d independents sets were created",
      i, CardOfSets (buffer));
#endif
      PtrToSets (*s) = (byte *) realloc (PtrToSets (*s),
  sizeofSets (*s) + sizeofSets (buffer));
      memcpy (PtrToSetN (*s, CardOfSets (*s)),
      PtrToSets (buffer), sizeofSets (buffer));
      CardOfSets (*s) += CardOfSets (buffer);
#ifndef NDEBUG
      printf ("\nThe current collection of independent sets:");
      PrintSets (*s);
#endif

      DestroySets (&buffer);
    }
  PackSets (s);
#ifndef NDEBUG
  printf ("\nThe final collection of independent sets after packing:");
  PrintSets (*s);
#endif
  DestroySets (&queue);
  return;
}
```

```
#ifdef __STDC__
bool
VerifyColors (graph g, int gid, GROUP * gp)
#else
bool
VerifyColors (g, gid, gp)
     graph g;
     int gid;
     GROUP *gp;
#endif
{
  int i, j;
  bool verified = true;


  if (gp != NULL)
    {
      if (EchoDetails)
OutputDetails (gp, g);
      gprintf (gp, "\nVerifying coloring for graph %d...", gid);
    }
  for (i = 0; i < GraphOrder (g); i++)
    for (j = i + 1; j < GraphOrder (g); j++)
      if (EdgeOfGraph (g, i, j) == 1)
if ((BegColor (ColorsOf (g), i) >= BegColor (ColorsOf (g), j)
     and BegColor (ColorsOf (g), i) <= EndColor (ColorsOf (g), j))
    or (BegColor (ColorsOf (g), j) >= BegColor (ColorsOf (g), i)
   and BegColor (ColorsOf (g), j) <= EndColor (ColorsOf (g), i)))
  {
    verified = false;
    if (gp != NULL)
      gprintf (gp, "\nconflict with nodes %d and %d.", i, j);
  }
  if (gp != NULL and verified)
    gprintf (gp, "verified.");

  return (verified);
}

#ifdef __STDC__
bool
fscanGraph (FILE * fp, graph * g, int *gid, int *group, int *row)
#else
bool
fscanGraph (fp, g, gid, newgroup, row)
     FILE *fp;
     graph *g;
     int *gid, *group, *row;
#endif
{
  int i, j;
```

```
  char line[MAXLINE], *buff;
  word order;

  loop
  {
    if (fgets (line, MAXLINE, fp) == NULL)
      return (false);
    (*row)++;
    strip_comment (line);
    if (strlen (line) == 0)
      continue;
    buff = (char *) strtok (strupper (line), WHITE_SPACE);

    if (strcmp (buff, SET_TOK) == 0)
      {
buff = (char *) strtok (NULL, WHITE_SPACE);

if (strcmp (buff, ECHO_TOK) == 0)
  {
    buff = (char *) strtok (NULL, WHITE_SPACE);
    EchoDetails = (strcmp (buff, ON_TOK) == 0);
    if (strtok (NULL, WHITE_SPACE) != NULL)
      error (BAD_INPUT, *row);
  }

else if (strcmp (buff, COLORS_TOK) == 0)
  {
    buff = (char *) strtok (NULL, WHITE_SPACE);
    KeepColors = (strcmp (buff, ON_TOK) == 0);
    if (strtok (NULL, WHITE_SPACE) != NULL)
      error (BAD_INPUT, *row);
  }

else
  error (BAD_SET, *row, buff);
      }

    else if (strcmp (buff, BEGIN_TOK) == 0)
      {
buff = (char *) strtok (NULL, WHITE_SPACE);

if (strcmp (buff, GRAPH_TOK) == 0)
  {
    buff = (char *) strtok (NULL, WHITE_SPACE);
    *gid = atoi (buff);
    buff = (char *) strtok (NULL, WHITE_SPACE);
    if (strcmp (buff, NODES_TOK) != 0)
      error (NO_NODES, *row);
    buff = (char *) strtok (NULL, WHITE_SPACE);
    order = atoi (buff);
    if (strtok (NULL, WHITE_SPACE) != NULL)
      error (BAD_INPUT, *row);
```

```
    repeat
      if (fgets (line, MAXLINE, fp) == NULL)
      error (NO_ADJMAT, *row);
    (*row)++;
    strip_comment (line);
    buff = (char *) strtok (strupper (line), WHITE_SPACE);
    until (buff != NULL);
    if (strcmp (buff, ADJ_TOK) != 0)
      error (NO_ADJMAT, *row);
    buff = (char *) strtok (NULL, WHITE_SPACE);
    if (strcmp (buff, MATRIX_TOK) != 0)
      error (NO_ADJMAT, *row);
    if (strtok (NULL, WHITE_SPACE) != NULL)
      error (BAD_INPUT, *row);
    CreateGraph (g, order);
    for (i = 0; i < GraphOrder (*g); i++)
      {
EdgeOfGraph (*g, i, i) = 0;
for (j = i; j < GraphOrder (*g); j++)
  {
    if ((buff = strtok (NULL, WHITE_SPACE)) == NULL)
      {
repeat
  if (fgets (line, MAXLINE, fp) == NULL)
  error (BAD_ADJMAT, *row);
(*row)++;
strip_comment (strupper (line));
buff = (char *) strtok (line, WHITE_SPACE);
until (buff != NULL);
      }
    EdgeOfGraph (*g, i, j) = atoi (buff);
  }
      }
    return (true);
  }

if (strcmp (buff, GROUP_TOK) == 0)
  {
    buff = (char *) strtok (NULL, WHITE_SPACE);
    *group = atoi (buff);
    if (strtok (NULL, WHITE_SPACE) != NULL)
      error (BAD_INPUT, *row);
  }

else
  error (NO_BEGIN, *row, GROUP_TOK);
      }

    else if (strcmp (buff, END_TOK) == 0)
      {
buff = (char *) strtok (NULL, WHITE_SPACE);
if (strcmp (buff, GROUP_TOK) != 0)
```

```
  error (NO_END, *row, GROUP_TOK);
if (strtok (NULL, WHITE_SPACE) != NULL)
  error (BAD_INPUT, *row);
      }

    else
      error (NO_BEGIN, *row, GRAPH_TOK);
  }
}

#ifdef __STDC__
bool
fscanChroma (FILE * fp, graph * g, chroma * c, int *row)
#else
bool
fscanChroma (fp, g, c, row)
     FILE *fp;
     graph *g;
     chroma *c;
     int *row;
#endif
{
  int i;
  char line[MAXLINE], *buff;

  repeat
    if (fgets (line, MAXLINE, fp) == NULL)
    error (NO_CHROMA, *row);
  (*row)++;
  strip_comment (strupper (line));
  buff = (char *) strtok (line, WHITE_SPACE);
  until (buff != NULL);
  if (strcmp (buff, CHROMA_TOK) != 0)
    error (NO_CHROMA, *row);
  buff = (char *) strtok (NULL, WHITE_SPACE);
  if (strcmp (buff, VECTOR_TOK) != 0)
    error (NO_CHROMA, *row);
  if (strtok (NULL, WHITE_SPACE) != NULL)
    error (BAD_INPUT, *row);
  CreateChroma (g, c);
  for (i = 0; i < GraphOrder (*g); i++)
    {
      if ((buff = strtok (NULL, WHITE_SPACE)) == NULL)
repeat
  if (fgets (line, MAXLINE, fp) == NULL)
  error (BAD_CHROMA, *row);
      (*row)++;
      strip_comment (strupper (line));
      buff = (char *) strtok (line, WHITE_SPACE);
      until (buff != NULL);
      NodeChroma (*c, i) = atoi (buff);
    }
```

```
  return (true);
}

#ifdef __STDC__
bool
fscanColors (FILE * fp, graph * g, colors * k, int *row)
#else
bool
fscanColors (fp, g, k, row)
     FILE *fp;
     graph *g;
     colors *k;
     int *row;
#endif
{
  int i;
  char line[MAXLINE], *buff;

  loop
  {
    if (fgets (line, MAXLINE, fp) == NULL)
      error (NO_END, *row, GRAPH_TOK);
    (*row)++;
    strip_comment (line);
    if (strlen (line) == 0)
      continue;
    buff = (char *) strtok (strupper (line), WHITE_SPACE);

    if (strcmp (buff, COLOR_TOK) == 0)
      {
buff = (char *) strtok (NULL, WHITE_SPACE);
if (strcmp (buff, VECTOR_TOK) != 0)
  error (NO_COLORS, *row);
if (strtok (NULL, WHITE_SPACE) != NULL)
  error (BAD_INPUT, *row);
CreateColors (g, k);
for (i = 0; i < GraphOrder (*g); i++)
  {
    if ((buff = strtok (NULL, WHITE_SPACE)) == NULL)
      repeat
if (fgets (line, MAXLINE, fp) == NULL)
error (BAD_COLORS, *row);
    (*row)++;
    strip_comment (strupper (line));
    buff = (char *) strtok (line, WHITE_SPACE);
    until (buff != NULL);
    BegColor (*k, i) = atoi (buff);
  }
repeat
  if (fgets (line, MAXLINE, fp) == NULL)
  error (NO_END, *row, GRAPH_TOK);
(*row)++;
```

```
strip_comment (strupper (line));
buff = (char *) strtok (line, WHITE_SPACE);
until (buff != NULL);
if (strcmp (buff, END_TOK) != 0)
  error (NO_END, *row, GRAPH_TOK);
buff = (char *) strtok (NULL, WHITE_SPACE);
if (strcmp (buff, GRAPH_TOK) != 0)
  error (NO_END, *row, GRAPH_TOK);
if (strtok (NULL, WHITE_SPACE) != NULL)
  error (BAD_INPUT, *row);
return (true);
      }

    else if (strcmp (buff, END_TOK) == 0)
      {
buff = (char *) strtok (NULL, WHITE_SPACE);
if (strcmp (buff, GRAPH_TOK) != 0)
  error (NO_END, *row, GRAPH_TOK);
if (strtok (NULL, WHITE_SPACE) != NULL)
  error (BAD_INPUT, *row);
return (false);
      }

    else
      error (NO_COLORS, *row);
  }
}

#ifdef __STDC__
bool
InputDetails (FILE * fp, graph * g, chroma * c, colors * k,
              int *row, int *gid, int *group)
#else
bool
InputDetails (fp, g, c, k, row, gid, group)
     FILE *fp;
     graph *g;
     chroma *c;
     colors *k;
     int *row, *gid, *group;
#endif
{
  if (not fscanGraph (fp, g, gid, group, row))
    return (false);
  fscanChroma (fp, g, c, row);
  fscanColors (fp, g, k, row);
  return (true);
}

#ifdef __STDC__
proc
gprintGraph (GROUP * gp, graph g)
```

```
#else
proc
gprintGraph (gp, g)
     GROUP *gp;
     graph g;
#endif
{
  int i, j;

  for (i = 0; i < GraphOrder (g); i++)
    {
      gprintf (gp, "\n");
      for (j = 0; j < i; j++)
gprintf (gp, "%3s", " ");
      for (j = i; j < GraphOrder (g); j++)
gprintf (gp, "%3d", EdgeOfGraph (g, i, j));
    }
  gprintf (gp, "\n");
  gflush (gp);
  return;
}

#ifdef __STDC__
proc
gprintChroma (GROUP * gp, chroma c)
#else
proc
gprintChroma (gp, c)
     GROUP *gp;
     chroma c;
#endif
{
  int i;

  gprintf (gp, "\n");
  for (i = 0; i < GraphOrder (GraphOf (c)); i++)
    gprintf (gp, "%3d", NodeChroma (c, i));
  gprintf (gp, "\n");
  gflush (gp);
  return;
}

#ifdef __STDC__
proc
gprintColors (GROUP * gp, colors k)
#else
proc
gprintColors (gp, k)
     GROUP *gp;
     colors k;
#endif
{
```

```
  int i;

  gprintf (gp, "\n");
  for (i = 0; i < GraphOrder (GraphOf (k)); i++)
    gprintf (gp, "%3d", BegColor (k, i));
  gprintf (gp, "\n");
  gflush (gp);
  return;
}

#ifdef __STDC__
proc
OutputDetails (GROUP * gp, graph g)
#else
proc
OutputDetails (gp, g)
     GROUP *gp;
     graph g;
#endif
{
  static word gid = 0;

  gprintf (gp, "%s %s %d %s ", BEGIN_TOK, GRAPH_TOK, gid, NODES_TOK);
  if (GraphAdjMat (g) == NULL)
    gprintf (gp, "0\n");
  else
    {
      gprintf (gp, "%d\n%s %s", GraphOrder (g), ADJ_TOK, MATRIX_TOK);
      gprintGraph (gp, g);
    }
  if (ChromaPtr (g) != NULL)
    {
      gprintf (gp, "%s %s", CHROMA_TOK, VECTOR_TOK);
      gprintChroma (gp, ChromaOf (g));
    }
  if (ColorsPtr (g) != NULL)
    {
      gprintf (gp, "%s %s", COLOR_TOK, VECTOR_TOK);
      gprintColors (gp, ColorsOf (g));
    }
  gprintf (gp, "%s %s\n", END_TOK, GRAPH_TOK);
  ++gid;
  return;
}

#ifdef __STDC__
proc
gprintSets (GROUP * gp, sets s)
#else
proc
gprintSets (gp, s)
     GROUP *gp;
```

```
     sets s;
#endif
{
  int i, j;

  gprintf (gp, "\n! Number of Sets: %d", CardOfSets (s));
  gprintf (gp, "\n! List of Sets:");
  for (i = 0; i < CardOfSets (s); i++)
    {
      gprintf (gp, "\n");
      for (j = 0; j < GraphOrder (GraphOf (s)); j++)
gprintf (gp, "%3d", SetElement (s, i, j));
    }
  gprintf (gp, "\n");
  gflush (gp);
  return;

}

#ifdef __STDC__
proc
ComplementGraph (graph g1, graph * g2)
#else
proc
ComplementGraph (g1, g2)
     graph g1;
     graph *g2;
#endif
{
  int i, j;

#ifndef NDEBUG
  printf ("\nCreating the complement of the graph.");
#endif
  CreateGraph (g2, GraphOrder (g1));
  GraphDensity (*g2) = 1.0 - GraphDensity (g1);
  for (i = 0; i < GraphOrder (*g2); i++)
    {
      EdgeOfGraph (*g2, i, i) = 0;
      for (j = i + 1; j < GraphOrder (*g2); j++)
EdgeOfGraph (*g2, i, j) = (Adjacent (g1, i, j) ? 0 : 1);
    }
  return;
}

#ifdef __STDC__
proc
CopyGraph (graph g1, graph * g2)
#else
proc
CopyGraph (g1, g2)
     graph g1;
```

```
     graph *g2;
#endif
{
#ifndef NDEBUG
  printf ("\nMaking a copy of the graph.");
#endif
  CreateGraph (g2, GraphOrder (g1));
  GraphDensity (*g2) = GraphDensity (g1);
  memcpy (GraphAdjMat (*g2), GraphAdjMat (g1), sizeofGraph (g1));
  ChromaPtr (*g2) = ChromaPtr (g1);
  ColorsPtr (*g2) = ColorsPtr (g1);
  return;
}

#ifdef __STDC__
proc
CopyChroma (chroma c1, chroma * c2)
#else
proc
CopyChroma (c1, c2)
     chroma c1;
     chroma *c2;
#endif
{
#ifndef NDEBUG
  printf ("\nMaking a copy of the chromaticity array.");
#endif
  CreateChroma (GraphPtr (c1), c2);
  memcpy (ChromaArr (*c2), ChromaArr (c1), sizeofChroma (c1));
  return;
}

#ifdef __STDC__
proc
CopyColors (colors k1, colors * k2)
#else
proc
CopyColors (k1, k2)
     colors k1;
     colors *k2;
#endif
{
#ifndef NDEBUG
  printf ("\nMaking a copy of the colors array.");
#endif
  CreateColors (GraphPtr (k1), k2);
  memcpy (ColorsArr (*k2), ColorsArr (k1), sizeofColors (k1));
  return;
}

#ifdef __STDC__
proc
```

```
CopySets (sets s1, sets * s2)
#else
proc
CopySets (s1, s2)
     sets s1;
     sets *s2;
#endif
{
#ifndef NDEBUG
  printf ("\nMaking a copy of the sets.");
#endif
  CardOfSets (*s2) = CardOfSets (s1);
  GraphPtr (*s2) = GraphPtr (s1);
  PtrToSets (*s2) = (byte *) malloc (sizeofSets (s1));
  memcpy (PtrToSets (*s2), PtrToSets (s1), sizeofSets (s1));
  return;
}

#ifdef __STDC__
proc
DestroyGraph (graph * g)
#else
proc
DestroyGraph (g)
     graph *g;
#endif
{
#ifndef NDEBUG
  printf ("\nDestroying the graph.");
#endif
  free (GraphAdjMat (*g));
  GraphAdjMat (*g) = NULL;
  GraphOrder (*g) = 0;
  GraphDensity (*g) = 0.0;
  ChromaPtr (*g) = NULL;
  ColorsPtr (*g) = NULL;
  return;
}

#ifdef __STDC__
proc
DestroyChroma (chroma * c)
#else
proc
DestroyChroma (c)
     chroma *c;
#endif
{
#ifndef NDEBUG
  printf ("\nDestroying the chromaticity array.");
#endif
  free (ChromaArr (*c));
```

```
  ChromaArr (*c) = NULL;
  ChromaPtr (GraphOf (*c)) = NULL;
  GraphPtr (*c) = NULL;
  return;
}

#ifdef __STDC__
proc
DestroyColors (colors * k)
#else
proc
DestroyColors (k)
     colors *k;
#endif
{
#ifndef NDEBUG
  printf ("\nDestroying the coloring array.");
#endif
  free (ColorsArr (*k));
  ColorsArr (*k) = NULL;
  ColorsPtr (GraphOf (*k)) = NULL;
  GraphPtr (*k) = NULL;
  return;
}

#ifdef __STDC__
proc
DestroySets (sets * s)
#else
proc
DestroySets (s)
     sets *s;
#endif
{
#ifndef NDEBUG
  printf ("\nDestroying the sets.");
#endif
  free (PtrToSets (*s));
  PtrToSets (*s) = NULL;
  CardOfSets (*s) = 0;
  GraphPtr (*s) = NULL;
  return;
}

#ifdef __STDC__
proc
PackSets (sets * s)
#else
proc
PackSets (s)
     sets *s;
#endif
```

```
{
  int i, j, k, sum;
  bool supset, subset;

#ifndef NDEBUG
  printf ("\nPacking %d sets.", CardOfSets (*s));
#endif
  for (i = 0; i < CardOfSets (*s); i++)
    {
#ifndef NDEBUG
      printf ("\nLooking at set %d in the current sets:", i);
      PrintSets (*s);
#endif
      /* check to see if set(i) is the empty set */
      sum = 0;
      for (j = 0; j < GraphOrder (GraphOf (*s)); j++)
sum += SetElement (*s, i, j);
      /* if set(i) is empty then delete the set from the list */
      if (sum == 0)
{
#ifndef NDEBUG
  printf ("\nSet %d is empty...deleting.", i);
#endif
  memmove (PtrToSetN (*s, i), PtrToSetN (*s, i + 1),
   sizeofSetN (*s, CardOfSets (*s) - i - 1));
  --i;
  --CardOfSets (*s);
}
      else
{
#ifndef NDEBUG
  printf ("\nSet %d is not empty...searching for sub/supersets.", i);
#endif
  /* search for subsets or supersets of set(i) */
  for (j = i + 1; j < CardOfSets (*s); j++)
    {
      subset = supset = true;
      for (k = 0; (supset || subset) &&
   k < GraphOrder (GraphOf (*s)); k++)
{
  if (SetElement (*s, i, k) < SetElement (*s, j, k))
    supset = false;
  if (SetElement (*s, i, k) > SetElement (*s, j, k))
    subset = false;
}
      if (subset)
{
#ifndef NDEBUG
  printf ("\nSet %d is a superset of set %d...deleting set %d"
  ,j, i, i);
#endif
  /* set(i) is a subset of set(j), delete set(i) */
```

```
  memmove (PtrToSetN (*s, i), PtrToSetN (*s, i + 1),
   sizeofSetN (*s, CardOfSets (*s) - i - 1));
  --i;
  --CardOfSets (*s);
  break;
}
      else if (supset)
{
#ifndef NDEBUG
  printf ("\nSet %d is a subset of set %d...deleting set %d."
   ,j, i, j);
#endif
  /* set(i) is a superset of set(j), delete set(j) */
  if (j < CardOfSets (*s) - 1)
    {
      memmove (PtrToSetN (*s, j), PtrToSetN (*s, j + 1),
       sizeofSetN (*s, CardOfSets (*s) - j - 1));
      --j;
    }
  --CardOfSets (*s);
}
    }
}
    }
  /* clean up memory */
  if (CardOfSets (*s) == 0)
    {
      free (PtrToSets (*s));
      PtrToSets (*s) = NULL;
    }
  else if ((PtrToSets (*s) = (byte *) realloc (PtrToSets (*s),
                                               sizeofSets (*s)))
   == NULL)
    error (MEMFAIL, "PackSets");
  return;
}

/* local functions */

#ifdef __STDC__
local proc
PrimeISets (graph * g, sets * s)
#else
local proc
PrimeISets (g, s)
     graph *g;
     sets *s;
#endif
{
  int i, j;

#ifndef NDEBUG
```

```
  printf ("\nPriming generate_independent_sets algorithm.");
#endif
  GraphPtr (*s) = g;
  CardOfSets (*s) = GraphOrder (*g);
#ifndef NDEBUG
  printf ("\nGenerating %d sets for consideration.", CardOfSets (*s));
#endif
  PtrToSets (*s) = (byte *) malloc (sizeofSetN (*s, CardOfSets (*s)));
  if (PtrToSets (*s) == NULL)
    error (MEMFAIL, "PrimeISets");
  for (i = 0; i < CardOfSets (*s); i++)
    {
      for (j = 0; j < i; j++)
SetElement (*s, i, j) = 0;
      SetElement (*s, i, i) = 1;
      for (j = i + 1; j < GraphOrder (GraphOf (*s)); j++)
SetElement (*s, i, j) = (Adjacent (GraphOf (*s), i, j) ? 0 : 1);
    }
#ifndef NDEBUG
  printf ("\nPrior to packing:");
  PrintSets (*s);
#endif
  PackSets (s);
#ifndef NDEBUG
  printf ("\nAfter packing:");
  PrintSets (*s);
#endif
  return;
}

#ifdef __STDC__
local proc
CreateISets (graph * g, byte * s, sets * t)
#else
local proc
CreateISets (g, s, t)
     graph *g;
     byte *s;
     sets *t;
#endif
{
  int i, j, k, first, mark;
  bool copied;

  /* initialize the set structure and copy s into t as the
     first set */
  CardOfSets (*t) = 1;
  GraphPtr (*t) = g;
  PtrToSets (*t) = (byte *) malloc (sizeofSets (*t));
  if (PtrToSets (*t) == NULL)
    error (MEMFAIL, "CreateISets (malloc)");
  memcpy (PtrToSets (*t), s, sizeofSets (*t));
```

```
  /* assume that s is nonempty and find the first element in
     the set (the element with the lowest index) */
  first = (-1);
  for (i = 0; i < GraphOrder (GraphOf (*t)); i++)
    if ((SetElement (*t, 0, i) == 1) and (first == (-1)))
      first = i;
#ifndef NDEBUG
  printf ("\nThe first element in the independent set is %d.", first);
#endif

  SetElement (*t, 0, first) = first;
  for (i = 0; i < CardOfSets (*t); i++)
    {
#ifndef NDEBUG
      printf ("\nCurrently using set %d.", i);
#endif
      mark = SetElement (*t, i, first);
      SetElement (*t, i, first) = 1;
#ifndef NDEBUG
      printf ("\nAll elements left of %d are independent (exclusive).",
              mark);
#endif
      for (j = mark; j < GraphOrder (GraphOf (*t)); j++)
if (SetElement (*t, i, j) == 1)
  {
#ifndef NDEBUG
    printf ("\nExamining element %d in set %d.", j, i);
#endif
    copied = false;
    for (k = j + 1; k < GraphOrder (GraphOf (*t)); k++)
      if ((SetElement (*t, i, k) == 1) and
  (EdgeOfGraph (GraphOf (*t), j, k) == 1))
{
#ifndef NDEBUG
  printf ("\nNode %d is connected to node %d.", k, j);
#endif
  if (not copied)
    {
#ifndef NDEBUG
      printf ("\nSplitting the set.");
#endif
      if ((PtrToSets (*t) =
   (byte *) realloc (PtrToSets (*t),
   sizeofSetN (*t, CardOfSets (*t) + 1)))
  == NULL)
error (MEMFAIL, "CreateISets (realloc)");
      memcpy (PtrToSetN (*t, CardOfSets (*t)),
      PtrToSetN (*t, i), sizeofSetN (*t, 1));
      SetElement (*t, i, k) = 0;
      SetElement (*t, CardOfSets (*t), j) = 0;
      SetElement (*t, CardOfSets (*t), first) = j + 1;
```

```
      ++CardOfSets (*t);
#ifndef NDEBUG
      printf ("\nNumber of independent sets is %d.",
      CardOfSets (*t));
#endif
      copied = true;
    }
  else
    SetElement (*t, i, k) = 0;
}
  }
    }
  PackSets (t);
#ifndef NDEBUG
  printf ("\nIndependent sets after packing:");
  PrintSets (*t);
#endif
  return;
}
```

graph.h The header for using graphs.

```
/*
        ADT Graph Header File
        NOTES: This package was used primarily for composite
               coloring of simple undirected graphs.  As such
               it only keeps the upper triangle of the adjacency
               matrix for the graph representation.
        $Log: graph.h $
 * Revision 1.7  1992/09/11  23:54:20  Jenness
 * A function prototype name was mis-spelled -- compiler didn't complain?
 *
 * Revision 1.6  1992/08/15  23:24:03  Jenness
 * reorganized the graph and colors data structures, the graph
 * data structure connects the chromaticity and colors arrays
 * added MemberOf and made some cosmetic changes
 *
 * Revision 1.5  1992/08/11  21:44:04  Jenness
 * minor changes
 *
 * Revision 1.4  1992/08/01  21:18:37  Jenness
 * Output routines will write files readable by "gverify"
 * Added routines to track colorings of a composite graph
 *
 * Revision 1.3  1992/08/01  14:41:25  Jenness
 * cosmetic changes
 *
 * Revision 1.2  1992/07/24  19:21:35  Jenness
 * Added the copy functions and general fixups
 *
 * Revision 1.1  1992/07/18  23:35:16  Jenness
 * Initial revision
```

```
 *
*/
#ifndef _GRAPH_H
#define _GRAPH_H

static char rcsid_GRAPH_H[] =
    "$Id: graph.h 1.7 1992/09/11 23:54:20 Jenness Exp $";

#include <stdio.h>
#include <stdlib.h>
#include "misc.h"
#include "groupo.h"

/* Macro Definitions:
   GRAPHELEM              - fundamental data type for adjacency matrix
   GRAPHELEMPTR           - data type for use in dynamic allocation
   sizeofGraph(n)         - returns the number of bytes occupied in memory
                            by a graph of order n
   mallocGraph(n)         - function to allocate dynamic memory for a graph
                            of order n
   EDGEOFFSET(n,i,j)      - calculates the offset into the array of edge
                            (i,j) for a graph of order n
   GraphAdjMat(g)         - returns a pointer to adjacency matrix
   GraphOrder(g)          - returns the order of graph g
   GraphDensity(g)        - returns the density of graph g
   EdgeOfGraph(g,i,j)     - returns the edge (i,j) of graph g
   Adjacent(g,i,j)        - boolean function that determines if edge (i,j)
                            of graph g exists
   MallocChroma(n)        - function to allocate dynamic memory for a
                            chromatic array for n nodes of a graph
   ChromaArr(c)           - returns a pointer to the chromaticity array
   AssocGraphPtr(c)       - returns a pointer to the associated graph
   AssocGraph(c)          - the structure associated with the graph
   NodeChroma(c,i)        - returns the chromaticity for node i
   PtrToSets(s)           - returns a pointer to the array of sets
   SetElement(s,i,j)      - returns the element in set i column j
   PtrToSetN(s,i)         - returns a pointer to set i
   CardOfSets(s)          - returns the cardinality of the set of sets
   sizeofSetN(s,n)        - returns the bytes required for a set of n sets
   sizeofSets(s)          - returns the bytes in the set of sets s
*/

#define EDGEOFFSET(n,i,j) ((i)<=(j)?((i)*(n)-(i)*((i)+1)/2+(j))\
                                   :((j)*(n)-(j)*((j)+1)/2+(i)))
#define GraphAdjMat(g) (g).edge
#define GraphOrder(g) (g).order
#define GraphDensity(g) (g).density
#define EdgeOfGraph(g,i,j) (g).edge[EDGEOFFSET(GraphOrder(g),i,j)]
#define Adjacent(g,i,j) (EdgeOfGraph(g,i,j)>0)
#define sizeofGraph(g) ((GraphOrder(g)*(GraphOrder(g)+1)/2)*sizeof(byte))
#define mallocGraph(g) (byte *)malloc(sizeofGraph(g))
#define ColorsPtr(g) (g).co
```

```
#define ColorsOf(g) (*(ColorsPtr(g)))

typedef struct _graph {
  word order;
  float density;
  byte *edge;
  struct _chroma *ch;
  struct _colors *co;
} graph;

#define ChromaArr(c) (c).node
#define GraphPtr(c) (c).g
#define GraphOf(c) (*(GraphPtr(c)))
#define NodeChroma(c,i) (c).node[i]
#define sizeofChroma(c) (GraphOrder(GraphOf(c))*sizeof(byte))
#define mallocChroma(c) (byte *)malloc(sizeofChroma(c))

typedef struct _chroma {
  graph *g;
  byte *node;
} chroma;

#define ColorsArr(k) (k).node
#define ChromaPtr(k) (k).ch
#define ChromaOf(k) (*(ChromaPtr(k)))
#define sizeofColors(k) (GraphOrder(GraphOf(k))*sizeof(byte))
#define mallocColors(k)  (byte *)malloc(sizeofColors(k))
#define BegColor(k,i) (k).node[i]
#define EndColor(k,i) (BegColor(k,i)+NodeChroma(ChromaOf(GraphOf(k)),i)-1)

typedef struct _colors {
  graph *g;
  byte *node;
} colors;

#define PtrToSets(s) (s).element
#define SetElement(s,i,j) (s).element[(i)*GraphOrder(GraphOf(s))+(j)]
#define MemberOf(s,i,j) (SetElement(s,i,j) == 1)
#define PtrToSetN(s,i) (addr(SetElement(s,i,0)))
#define CardOfSets(s) (s).cardinality
#define sizeofSetN(s,n) ((n)*GraphOrder(GraphOf(s))*sizeof(byte))
#define sizeofSets(s) (sizeofSetN(s,CardOfSets(s)))
#define PrintSets(s)

typedef struct _sets {
  graph *g;
  word cardinality;
  byte *element;
} sets;

/* Function declarations for graph.c:
   CreateGraph                  - create space for a graph
```

```
   GenerateEdges              - generate edges for a graph
   CreateChroma               - create space for a chromaticity array
   GenerateChroma             - generate the chromaticity array
   TruncPoisson               - generates chromaticities
   CreateColors               - create space for a coloring array
   GenerateISets              - generate the independent sets
   gprintGraph                - print the graph's adjacency matrix
   gprintChroma               - print the chromaticity array
   gprintColors               - print the coloring array
   OutputDetails              - output a graph following "gdetails.doc"
   gprintSets                 - print the list of sets
   ComplementGraph            - return a pointer to the complement of the
                                graph
   CopyGraph                 - makes a copy of a graph
   CopyChroma                - makes a copy of the chromaticity array
   CopyColors                - makes a copy of the coloring array
   CopySets                  - makes a copy of the sets
   DestroyGraph              - free the memory associated with the graph
   DestroyChroma             - free the memory associated with the
                                chromaticity array
   DestroyColors             - free the memory associated with the
                                coloring array
   DestroySets               - free the memory associated with the sets
*/

extern bool EchoDetails, KeepColors;
proc CreateGraph ARGS((graph *g, word n));
proc GenerateEdges ARGS((graph *g, float d));
proc CreateChroma ARGS((graph *g, chroma *c));
proc GenerateChroma ARGS((chroma *c, word (*f)(void)));
word TruncPoisson ARGS((void));
word Fixed60_40 ARGS((void));
word Fixed75_25 ARGS((void));
word Uniform4 ARGS((void));
word Uniform3 ARGS((void));
extern char *tok_pdistr[];
extern word (*(fp_pdistr[]))(void);
extern const int num_pdistr;
proc CreateColors ARGS((graph *g, colors *k));
proc GenerateISets ARGS((graph *g, sets *i));
bool VerifyColors ARGS((graph g, int gid, GROUP *gp));
bool fscanGraph ARGS((FILE *fp, graph *g, int *gid, int *group, int *row));
bool fscanChroma ARGS((FILE *fp, graph *g, chroma *c, int *row));
bool fscanColors ARGS((FILE *fp, graph *g, colors *k, int *row));
bool InputDetails ARGS((FILE *fp, graph *g, chroma *c, colors *k, int *row,
                        int *gid, int *group));
proc gprintGraph ARGS((GROUP *gp, graph g));
proc gprintChroma ARGS((GROUP *gp, chroma c));
proc gprintColors ARGS((GROUP *gp, colors k));
proc OutputDetails ARGS((GROUP *gp, graph g));
proc gprintSets ARGS((GROUP *gp, sets s));
proc ComplementGraph ARGS((graph g1, graph *g2));
```

```
proc CopyGraph ARGS((graph g1, graph *g2));
proc CopyChroma ARGS((chroma c1, chroma *c2));
proc CopyColors ARGS((colors k1, colors *k2));
proc CopySets ARGS((sets s1, sets *s2));
proc DestroyGraph ARGS((graph *g));
proc DestroyChroma ARGS((chroma *c));
proc DestroyColors ARGS((colors *k));
proc DestroySets ARGS((sets *s));
proc PackSets ARGS((sets *s));

#endif /* _GRAPH_H */
```

grcg.c A program to generate random composite graphs.

```
/* GRCG.C
 * Description:
 *              Generate Random Composite Graphs.
 *
 *              // comment to EOL
 *              SET SEEDS int int
 *              SET ECHO  [ON|OFF]
 *              SET COLORS [ON|OFF]
 *              SET GRAPHS int
 *              SET NODES int
 *              SET DENSITY float
 *              SET DISTRIBUTION [POISSON|75-25|60-40|UNIFORM-3|UNIFORM-4]
 *              GENERATE GRAPHS
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "misc.h"
#include "random.h"
#include "groupo.h"
#include "graph.h"

GROUP *out;
graph g;
chroma ch;

main()
{
    int i, j;
    char line[MAXLINE]; /* buffer for each line from input      */
    char *buff;         /* pointer into the line buffer          */
    int row;            /* current row number being processed    */
    bool echo, colors; /* flags for coloring algorithms          */
    int seed1, seed2; /* seed values for the RNG                 */
    int graphs, nodes;
    float density;
    int pdf;
```

```
int group;

/* initialize */
out = gopen(stdout);
row = 0;
graphs = 1; nodes = 10; /* default parameters              */
density = 0.5;      /* default density                     */
pdf = 0;              /* default distribution POISSON      */
echo = colors = false; /* default is off for these flags   */
seed1 = 1777; seed2 = 1847; /* default seed values         */
group = 0;

while ( fgets(line, MAXLINE, stdin) != NULL ) {
    ++row;
    strip_comment(line);
    if (strlen(line) == 0)
        continue;
    buff = (char *)strtok(strupper(line), WHITE_SPACE);

    if (strcmp(buff, SET_TOK) == 0) {
        buff = (char *)strtok(NULL, WHITE_SPACE);

        if (strcmp(buff, ECHO_TOK) == 0) {
            buff = (char *)strtok(NULL, WHITE_SPACE);
            echo = (strcmp(buff, ON_TOK) == 0);
            if (strtok(NULL, WHITE_SPACE) != NULL)
                error(BAD_INPUT, row);
        }

        else if (strcmp(buff, COLORS_TOK) == 0) {
            buff = (char *)strtok(NULL, WHITE_SPACE);
            colors = (strcmp(buff, ON_TOK) == 0);
            if (strtok(NULL, WHITE_SPACE) != NULL)
                error(BAD_INPUT, row);
        }

        else if (strcmp(buff, SEED_TOK) == 0) {
            buff = (char *)strtok(NULL, WHITE_SPACE);
            seed1 = atol(buff);
            buff = (char *)strtok(NULL, WHITE_SPACE);
            seed2 = atol(buff);
            if (strtok(NULL, WHITE_SPACE) != NULL)
                error(BAD_INPUT, row);
        }

        else if (strcmp(buff, GRAPHS_TOK) == 0) {
            buff = (char *)strtok(NULL, WHITE_SPACE);
            graphs = atol(buff);
            if (strtok(NULL, WHITE_SPACE) != NULL)
                error(BAD_INPUT, row);
        }
```

```
        else if (strcmp(buff, NODES_TOK) == 0) {
            buff = (char *)strtok(NULL, WHITE_SPACE);
            nodes = atoi(buff);
            if (strtok(NULL, WHITE_SPACE) != NULL)
                error(BAD_INPUT, row);
        }

        else if (strcmp(buff, DENSITY_TOK) == 0) {
            buff = (char *)strtok(NULL, WHITE_SPACE);
            density = atof(buff);
            if (strtok(NULL, WHITE_SPACE) != NULL)
                error(BAD_INPUT, row);
        }

        else if (strcmp(buff, DISTR_TOK) == 0) {
            buff = (char *)strtok(NULL, WHITE_SPACE);
            for(pdf = 0; pdf < num_pdistr; pdf++)
                if (strcmp(buff, tok_pdistr[pdf]) == 0) {
                    break;
                }
            if (pdf >= num_pdistr or strtok(NULL, WHITE_SPACE) != NULL)
                error(BAD_DISTR, row);
        }

        else
            error(BAD_SET, row, buff);
    }

    else if (strcmp(buff, GENERATE_TOK) == 0) {
        buff = (char *)strtok(NULL, WHITE_SPACE);
        if (strcmp(buff, GRAPHS_TOK) == 0) {
            printf("// This group of graphs was created by GRCG:\n");
            printf("//\tGRAPHS  : %d\n",graphs);
            printf("//\tORDER   : %d\n",nodes);
            printf("//\tDENSITY : %0.2f\n",density);
            printf("//\tPDF     : %s\n",tok_pdistr[pdf]);
            printf("//\tSEEDS   : %d %d\n",seed1,seed2);
            printf("\nBEGIN GROUP %d\n",group++);
            if (echo)
                printf("SET ECHO ON\n");
            if (colors)
                printf("SET COLORS ON\n");
            srandf(seed1, seed2);
            CreateGraph(&g, nodes);
            CreateChroma(&g, &ch);
            for(i = 0; i < graphs; i++) {
                GenerateEdges(&g, density);
                GenerateChroma(&ch, fp_pdistr[pdf]);
                OutputDetails(out,g);
            }
            DestroyChroma(&ch);
            DestroyGraph(&g);
```

```
                printf("END GROUP\n\n");
            }

            else
                error(BAD_INPUT, row);
        }

        else
            error(UNKNOWN_INSTR, row);

    }
    return NO_ERRORS;
}
```

groupo.c Implements group controlled output functions.

```
/*
        Group Output C Package
        $Log: groupo.c $
 * Revision 1.4  1992/08/15  23:31:52  Jenness
 * added error() function for checking memory allocations
 *
 * Revision 1.3  1992/08/11  21:46:05  Jenness
 * change reflected by change to list package
 *
 * Revision 1.2  1992/08/01  21:20:16  Jenness
 * cosmetic changes
 *
 * Revision 1.1  1992/08/01  15:54:05  Jenness
 * Initial revision
 *
*/

static char rcsid_GROUPO_C[] =
    "$Id: groupo.c 1.4 1992/08/15 23:31:52 Jenness Exp $";


#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <assert.h>
#include "misc.h"
#include "lists.h"
#include "groupo.h"

#ifdef __STDC__
GROUP *gopen(FILE *fp)
#else
GROUP *gopen(fp) FILE *fp;
#endif
{
```

```
    GROUP *gp;

#ifndef NDEBUG
    printf("\nOpening an output group with first file.");
#endif
    gp = (GROUP *)malloc(sizeof(GROUP));
    if (gp == NULL)
        error (MEMFAIL, "gopen");
    NewList(gp, sizeof(FILE));
    AppendToList(gp,fp);
    return gp;
}

#ifdef __STDC__
proc gfadd(GROUP *gp, FILE *fp)
#else
proc gfadd(gp, fp) GROUP *gp; FILE *fp;
#endif
{
#ifndef NDEBUG
    printf("\nAdding file to the group.");
#endif
    AppendToList(gp,fp);
    return;
}

#ifdef __STDC__
proc gfdel(GROUP *gp, FILE *fp)
#else
proc gfdel(gp, fp) GROUP *gp; FILE *fp;
#endif
{
    link *l;

#ifndef NDEBUG
    printf("\nLooking for file to delete ... ");
#endif
    for(l=ListHead(*gp);l!=EndOfList(*gp);l=NextLink(*l))
        if (LinkData(*l)==(pointer)fp) {
#ifndef NDEBUG
    printf("found ... deleting file from the group.");
#endif
    DeleteLink(gp,l);
    break;
}
    return;
}

#ifdef __STDC__
proc gputc(char c, GROUP *gp)
#else
proc gputc(c, gp) char c; GROUP *gp;
```

```
#endif
{
    link *l;

#ifndef NDEBUG
    printf("\nOutputing character '%c' to group.",c);
#endif
    for(l=ListHead(*gp);l!=EndOfList(*gp);l=NextLink(*l))
        fputc(c,LinkData(*l));
    return;
}

#ifdef __STDC__
proc gputs(char *s, GROUP *gp)
#else
proc gputs(s, gp) char *s; GROUP *gp;
#endif
{
    link *l;

#ifndef NDEBUG
    printf("\nOutputing string \"%s\" to group.",s);
#endif
    for(l=ListHead(*gp);l!=EndOfList(*gp);l=NextLink(*l))
        fputs(s,LinkData(*l));
    return;
}

#ifdef __STDC__
proc gprintf(GROUP *gp, char *fmt, ...)
#else
proc gprintf(gp, fmt, ...) GROUP *gp; char *fmt;
#endif
{
    link *l;
    va_list args;

#ifndef NDEBUG
    printf("\nOutputing to group with format \"%s\".",fmt);
#endif
    va_start(args, fmt);
    for(l=ListHead(*gp);l!=EndOfList(*gp);l=NextLink(*l))
        vfprintf(LinkData(*l),fmt,args);
    va_end(args);
    return;
}

#ifdef __STDC__
proc gflush(GROUP *gp)
#else
proc gflush(gp) GROUP *gp;
#endif
```

```
{
    link *l;

#ifndef NDEBUG
    printf("\nFlushing all members of group.");
#endif
    for(l=ListHead(*gp);l!=EndOfList(*gp);l=NextLink(*l))
        fflush(LinkData(*l));
    return;
}

#ifdef __STDC__
proc gclose(GROUP *gp)
#else
proc gclose(gp) GROUP *gp;
#endif
{
    link *l;

#ifndef NDEBUG
    printf("\nClosing group.");
#endif
    for(l=ListHead(*gp);l!=EndOfList(*gp);l=NextLink(*l))
        fclose(LinkData(*l));
    FreeList(gp);
    free(gp);
    return;
}
```

groupo.h Header for using group output.

```
/*
        Group Output Header File
        NOTES: The functions in this package work in a parallel way
       that the ANSI C library works for a single stream.
       This package provides a single function call so that
       output will be directed to multiple streams.  Input
       functions have not been implemented as of yet (no
       good reason has been found to need such functions).
       Future additions include the addition of a group
       merge, return codes that follow the standard library,
       and subgroup "tags" so that the programmer can
       specify a subgroup as "disk" or "screen", etc.
        $Log: groupo.h $
 * Revision 1.3  1992/08/11  21:46:56  Jenness
 * minor changes
 *
 * Revision 1.2  1992/08/01  21:20:16  Jenness
 * cosmetic changes
 *
 * Revision 1.1  1992/08/01  15:54:05  Jenness
 * Initial revision
```

```
 *
*/

#ifndef _GROUPO_H
#define _GROUPO_H

static char rcsid_GROUPO_H[] =
    "$Id: groupo.h 1.3 1992/08/11 21:46:56 Jenness Exp $";

#include <stdio.h>
#include "misc.h"
#include "lists.h"

#define GROUP list

GROUP *gopen ARGS((FILE *fp));
proc gfadd ARGS((GROUP *gp, FILE *fp));
proc gfdel ARGS((GROUP *gp, FILE *fp));
proc gputc ARGS((char c, GROUP *gp));
proc gputs ARGS((char *s, GROUP *gp));
proc gprintf ARGS((GROUP *gp, char *fmt, ...));
proc gflush ARGS((GROUP *gp));
proc gclose ARGS((GROUP *gp));

#endif /* _GROUPO_H */
```

lists.c Implements linked lists.

```
/*
        Linked Lists C Package
        $Log: lists.c $
 * Revision 1.6  1992/09/11  23:51:55  Jenness
 * Forgot to include string.h -- done.
 *
 * Revision 1.5  1992/08/15  23:29:28  Jenness
 * added error() to provide for simple error handling
 *
 * Revision 1.4  1992/08/11  21:42:31  Jenness
 * Added CopyLink and added a second parameter to NewList
 *
 * Revision 1.3  1992/08/01  21:16:57  Jenness
 * cosmetic changes
 *
 * Revision 1.2  1992/08/01  15:48:30  Jenness
 * Added FreeListAndData function.
 *
 * Revision 1.1  1992/08/01  14:53:13  Jenness
 * Initial revision
 *
*/

static char rcsid_LISTS_C[] =
```

```
    "$Id: lists.c 1.6 1992/09/11 23:51:55 Jenness Exp $";

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "misc.h"
#include "lists.h"

#ifdef __STDC__
proc NewList(list *l, word s)
#else
proc NewList(l, s) list *l; word s;
#endif
{
#ifndef NDEBUG
    printf("\nInitializing a linked list.");
#endif
    ListLength(*l) = 0;
    ListHead(*l) = (link *)l;
    ListTail(*l) = (link *)l;
    ListSize(*l) = s;
    return;
}

#ifdef __STDC__
proc InsertAfterLink(list *l, link *n, pointer d)
#else
proc InsertAfterLink(l, n, d) list *l; link *n;  pointer d;
#endif
{
    link *m;

#ifndef NDEBUG
    printf("\nInserting %s into the linked list at position after %p.",d,n);
#endif
    m = (link *)malloc(sizeof(link));
    if (m == NULL)
        error (MEMFAIL, "InsertAfterLink");
    LinkData(*m) = d;
    PrevLink(*m) = n;
    NextLink(*m) = NextLink(*n);

    PrevLink(*NextLink(*n)) = m;
    NextLink(*n) = m;

    ++ListLength(*l);
    return;
}

#ifdef __STDC__
proc InsertBeforeLink(list *l, link *n, pointer d)
#else
```

```
proc InsertBeforeLink(l, n, d) list *l; link *n;  pointer d;
#endif
{
    link *m;

#ifndef NDEBUG
    printf("\nInserting %s into the linked list at position before %p.",
           d,n);
#endif
    m = (link *)malloc(sizeof(link));
    if (m == NULL)
        error (MEMFAIL, "InsertBeforeLink");
    LinkData(*m) = d;
    PrevLink(*m) = PrevLink(*n);
    NextLink(*m) = n;

    NextLink(*PrevLink(*n)) = m;
    PrevLink(*n) = m;

    ++ListLength(*l);
    return;
}

#ifdef __STDC__
proc CopyLink(list *l, link *n, pointer d)
#else
proc CopyLink(l, n, d) list *l; link *n; pointer d;
#endif
{
#ifndef NDEBUG
    printf("\nCopying link at position %p.",n);
#endif
    memcpy(d,LinkData(*n),ListSize(*l));
    return;
}

#ifdef __STDC__
proc DeleteLink(list *l, link *n)
#else
proc DeleteLink(l, n) list *l; link *n;
#endif
{
#ifndef NDEBUG
    printf("\nDeleting link at position %p.",n);
#endif
    if (ListLength(*l) != 0) {
        NextLink(*PrevLink(*n)) = NextLink(*n);
PrevLink(*NextLink(*n)) = PrevLink(*n);
free(LinkData(*n));
free(n);
--ListLength(*l);
    }
```

```
    return;
}

#ifdef __STDC__
link *FindLinkN(list l, word n)
#else
link *FindLinkN(l, n) list l; word n;
#endif
{
    int i;
    link *m;

    m = NULL;
    if (ListLength(l) >= n) {
        m = ListHead(l);
for(i=0; i<n; i++)
    m = NextLink(*m);
    }
#ifndef NDEBUG
    printf("\nFound link %d at position %p.",n,m);
#endif
    return m;
}

#ifdef __STDC__
proc FreeList(list *l)
#else
proc FreeList(l) list *l;
#endif
{
    link *m, *n;

#ifndef NDEBUG
    printf("\nFreeing all links in linked list.");
#endif
    m = ListHead(*l);
    while (ListLength(*l) > 0) {
n = NextLink(*m);
free(m);
m = n;
--ListLength(*l);
    }
    return;
}

#ifdef __STDC__
proc FreeListAndData(list *l)
#else
proc FreeListAndData(l) list *l;
#endif
{
    link *m, *n;
```

```
#ifndef NDEBUG
    printf("\nFreeing all links and data in linked list.");
#endif
    m = ListHead(*l);
    while (ListLength(*l) > 0) {
        free(LinkData(*m));
n = NextLink(*m);
free(m);
m = n;
--ListLength(*l);
    }
    return;
}
```

lists.h The header for using linked lists.

```
/*
        Linked Lists Header File
        NOTES: This packages implements doubly linked lists.
       The list data structure and the links are as
       follows:

       List Data Structure:
         +------------------------------------+
 | First | Last  | Number  | Number  |
 | Link  | Link  |   of    |   of    |
 |  in   |  in   | Links   | Bytes   |
 | List  | List  | in List | in Data |
 +------------------------------------+

       The First Link is a pointer to the first link.
       The Last Link is a pointer to the last link.
       The chain of pointers in the list is somewhat
       circular in that the chain always points back to
       the list data structure itself.  This eliminates
       the need for the special condition needed to add
       a link when the list is empty.

       Link Data Structure:
         +-------------------------+
 | Next  | Prev  | Pointer |
 | Link  | Link  |   to    |
 |  in   |  in   |  Data   |
 | List  | List  |         |
 +-------------------------+

       The pointer to data is generic and the data need
       not be the same size, but only homogeneous lists
       can make use of the CopyLink function.  If Hetero-
       geneous lists are desired then do not use the
       CopyLink function and send anything to the function
```

```
      NewList as the list size parameter (which represents
      the size of data in bytes).
       $Log: lists.h $
 * Revision 1.4  1992/08/11  21:43:38  Jenness
 * Added CopyLink and added a second parameter to NewList
 *
 * Revision 1.3  1992/08/01  21:16:57  Jenness
 * cosmetic changes
 *
 * Revision 1.2  1992/08/01  15:48:30  Jenness
 * Added FreeListAndData function.
 *
 * Revision 1.1  1992/08/01  14:53:13  Jenness
 * Initial revision
 *
*/

#ifndef _LISTS_H
#define _LISTS_H

static char rcsid_LISTS_H[] =
    "$Id: lists.h 1.4 1992/08/11 21:43:38 Jenness Exp $";

#include "misc.h"

#define PrevLink(n) (n).aft
#define NextLink(n) (n).fort
#define LinkData(n) (n).data

typedef struct _link {
    struct _link *fort;
    struct _link *aft;
    pointer data;
} link;

#define ListLength(l) (l).length
#define ListHead(l) (l).fort
#define ListTail(l) (l).aft
#define ListSize(l) (l).size

typedef struct {
    link *fort;
    link *aft;
    dword length;
    word size;
} list;

#define AppendToList(l,d) InsertAfterLink(l,ListTail(*(l)),(pointer)d)
#define PrependToList(l,d) InsertBeforeLink(l,ListHead(*(l)),(pointer)d)
#define BeginOfList(l) (link *)addr(l)
#define EndOfList(l) (link *)addr(l)
```

```
proc NewList ARGS((list *l, word s));
proc InsertAfterLink ARGS((list *l, link *n, pointer d));
proc InsertBeforeLink ARGS((list *l, link *n, pointer d));
proc CopyLink ARGS((list *l, link *n, pointer d));
proc DeleteLink ARGS((list *l, link *n));
link *FindLinkN ARGS((list l, word n));
proc FreeList ARGS((list *l));
proc FreeListAndData ARGS((list *l));

#endif /* _LISTS_H */
```

misc.c Implements the error function and other utilities.

```
/*
        Miscellany C Package
        $Log: misc.c $
 * Revision 1.4  1992/09/11  23:52:55  Jenness
 * error() was not ANSI compatible -- changed.
 *
 * Revision 1.3  1992/08/18  19:34:24  Jenness
 * only comments changed
 *
 * Revision 1.2  1992/08/15  23:23:22  Jenness
 * added a simple error handler
 *
 * Revision 1.1  1992/08/01  14:52:48  Jenness
 * Initial revision
 *
*/

static char rcsid_MISC_C[] =
    "$Id: misc.c 1.4 1992/09/11 23:52:55 Jenness Exp $";

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include "misc.h"

/* scans for 'Y' or 'N' and returns it */
char getYN(void)
{
    char c;

    do
        c = (getchar() & 0xDF);
    while (c != 'Y' and c != 'N');

    return c;
}
```

```
/* convert a string to uppercase */
#ifdef __STDC__
char *strupper(char *s)
#else
char *strupper(s) char *s;
#endif
{
    int i;

    for (i = 0; i < strlen(s); i++)
        s[i] = toupper(s[i]);

    return s;
}

/* strips comments from the line of input */
#ifdef __STDC__
proc strip_comment(char *p)
#else
proc strip_comment(p) char *p;
#endif
{
    char *c;

    if ((c = (char *)strstr(p, COMMENT_TOK)) != NULL)
        *c = (char)0;
    else if ((c = (char *)strstr(p, EOL_TOK)) != NULL)
        *c = (char)0;
}

/* augmented error list */
const char *aug_errlist[] =
  {
    /*               */ "Unknown error code: %d",
    /* OPENFAIL      */ "Open failed for file \"%s\"",
    /* MEMFAIL       */ "Memory allocation failed in function \"%s\"",
    /* UNKNOWN_INSTR */ "Line %d: Unknown instruction",
    /* OUT_OF_RANGE  */ "Line %d: Data out of range",
    /* BAD_INPUT     */ "Line %d: Invalid instruction format",
    /* BAD_SET       */ "Line %d: Unknown SET variable %s",
    /* BAD_DISTR     */ "Line %d: Unknown distribution function",
    /* NO_BEGIN      */ "Line %d: BEGIN %s expected",
    /* NO_END        */ "Line %d: END %s expected",
    /* NO_NODES      */ "Line %d: NODES not specified",
    /* NO_ADJMAT     */ "Line %d: ADJACENCY MATRIX expected",
    /* BAD_ADJMAT    */ "Line %d: Bad data reading adjacency matrix",
    /* NO_CHROMA     */ "Line %d: CHROMATICITY VECTOR expected",
    /* BAD_CHROMA    */ "Line %d: Bad data reading chromaticity vector",
    /* NO_COLORS     */ "Line %d: COLORS VECTOR expected",
    /* BAD_COLORS    */ "Line %d: Bad data reading colors vector"
  };
```

```
const int aug_nerr = sizeof (aug_errlist) / sizeof (aug_errlist[0]) - 1;

/* error handler */
#define ERRMSGLEN 100

#ifdef __STDC__
proc error(int errnum, ...)
#else
proc error(errnum, ...) int errnum;
#endif
{
  char errmsg[ERRMSGLEN];
  va_list args;

  va_start(args, errnum);
  if (errnum > 0 and errnum <= aug_nerr)
    {
      vsprintf(errmsg,aug_errlist[errnum],args);
      if (errno != 0) {
          strncat(errmsg, ": ", ERRMSGLEN - strlen(errmsg));
          strncat(errmsg, strerror(errno), ERRMSGLEN - strlen(errmsg));
      }
    }
  else
    sprintf(errmsg, aug_errlist[0], errnum);
  fprintf(stderr, "\nERROR(%d): %s\n", errnum, errmsg);
  va_end(args);
  exit(errnum);
}
```

misc.h Header file for the miscellaneous functions.

```
/*
        Miscellany Header File
        NOTES: This file makes up for deficient C compilers as well as
               providing more useful macros and data types.

               DECLARATION macros:
               ARGS(x) is used in function declarations in both ANSI
                       as well as KR C.
               proc    is a designation for procedures.
               pointer is a generic memory pointer.
               addr(x) is a generic address of x.
               void    is used in empty function parameters lists.
               local   is a way to declare local functions and restrict
                       scope.  Turn of this by -DNLOCAL.

               DATA TYPES:
               byte  is an unsigned 8-bit integer.
               word  is an unsigned 16-bit integer.
               dword is an unsigned 32-bit integer.
```

```
                bool  is a logical data type.

                LANGUAGE macros:
                and  is logical 'and'.
                or   is logical 'or'.
                not  is logical 'not'.
                loop is a nonterminal loop construct.

                FUNCTIONS:
                getYN() reads 'y','Y','n', or 'N' from stdin, returns 'Y','N'.
                error(errnum, ...) provides for handling errors simply.

        $Log: misc.h $
 * Revision 1.5  1992/08/15  23:22:30  Jenness
 * added a simple error handler and provided for some common
 * macros found in stdio and stdlib
 *
 * Revision 1.4  1992/08/11  21:41:05  Jenness
 * cosmetic changes
 *
 * Revision 1.3  1992/08/01  21:18:21  Jenness
 * cosmetic changes
 *
 * Revision 1.2  1992/08/01  14:52:27  Jenness
 * Added misc.c prototypes and generic pointer type
 *
 * Revision 1.1  1992/07/18  23:34:27  Jenness
 * Initial revision
 *
*/
#ifndef _MISC_H
#define _MISC_H

static char rcsid_MISC_H[] =
    "$Id: misc.h 1.5 1992/08/15 23:22:30 Jenness Exp $";

#define NDEBUG

#ifdef __STDC__

#define ARGS(x) x
#define proc void
#define pointer void *
#define addr(x) ((pointer)&(x))

#else /* ! __STDC__ */

#define const
#define void
#define ARGS(x) ()
#define proc int
#define assert(s)
```

```
#define pointer char *
#define addr(x) ((pointer)&(x))

#endif /* __STDC__ */

#ifndef max
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))
#endif

#ifndef NULL
#define NULL (pointer)0
#endif /* NULL */

/* #ifndef RAND_MAX
#define RAND_MAX 0x7FFF
#endif /* RAND_MAX */

#ifndef SHRT_MAX
#define CHAR_MAX +127
#define CHAR_MIN -127
#define SHRT_MAX +32767
#define SHRT_MIN -32767
#define LONG_MAX +2147483647L
#define LONG_MIN -2147483647L

#define BYTE_MAX 0xFFU
#define UCHAR_MAX BYTE_MAX
#define WORD_MAX 0xFFFFU
#define USHRT_MAX WORD_MAX
#define DWORD_MAX 0xFFFFFFFFUL
#define ULONG_MAX DWORD_MAX

/* beware, int types are usually not portable */
#define INT_MAX LONG_MAX
#define INT_MIN LONG_MIN
#define UINT_MAX ULONG_MAX
#endif

#ifndef FLT_MAX
#define FLT_MAX 1E+37
#endif

#ifndef CLK_TCK
#define CLK_TCK 1000
#endif

/* generic data types */
typedef unsigned char byte;  /* 8-bits */
typedef unsigned short word; /* 16-bits */
typedef unsigned long dword; /* 32-bits */
typedef enum {
```

```
  false = 0, true = 1
} bool;

/* logical operators */
#define not !
#define and &&
#define or ||

/* generic constructs */
#define loop for(;;)
#define repeat do {
#define until(x) } while (not(x))

#ifndef NLOCAL
#define local static
#else
#define local
#endif

/* error codes */
enum {
    NO_ERRORS, OPENFAIL, MEMFAIL, UNKNOWN_INSTR, OUT_OF_RANGE,
    BAD_INPUT, BAD_SET, BAD_DISTR, NO_BEGIN, NO_END, NO_NODES,
    NO_ADJMAT, BAD_ADJMAT, NO_CHROMA, BAD_CHROMA, NO_COLORS, BAD_COLORS
};

/* implementation defined limits */
#define MAXLINE 4096

/* string tokens */
#define BEGIN_TOK       "BEGIN"
#define END_TOK         "END"
#define GROUP_TOK       "GROUP"
#define GRAPH_TOK       "GRAPH"
#define ADJ_TOK         "ADJACENCY"
#define MATRIX_TOK      "MATRIX"
#define COLOR_TOK       "COLOR"
#define CHROMA_TOK      "CHROMATICITY"
#define VECTOR_TOK      "VECTOR"
#define SET_TOK         "SET"
#define ON_TOK          "ON"
#define OFF_TOK         "OFF"
#define SEED_TOK        "SEEDS"
#define ECHO_TOK        "ECHO"
#define COLORS_TOK      "COLORS"
#define GRAPHS_TOK      "GRAPHS"
#define NODES_TOK       "NODES"
#define DENSITY_TOK     "DENSITY"
#define DISTR_TOK       "DISTRIBUTION"
#define GENERATE_TOK    "GENERATE"
#define COMMENT_TOK     "//"
#define EOL_TOK         "\n"
```

```
#define WHITE_SPACE     " \t\n"

/* miscellaneous function prototypes from MISC C Package */
char getYN(void);
char *strupper ARGS((char *s));
proc strip_comment ARGS((char *p));
proc error ARGS((int, ...));

#ifndef NTRACE
#define trace(fmt,v) printf("TRACE: %s(%d): %s = " fmt "\n", \
                            __FILE__,__LINE__,#v,v)
#else
#define trace(fmt,v)
#endif /* NTRACE */

#endif /* _MISC_H */
```

random.c Implements integer and float random number generators.

```
/*
        Random Number Generators C Package
        $Log: random.c $
 * Revision 1.2  1992/08/01  21:17:28  Jenness
 * cosmetic changes
 *
 * Revision 1.1  1992/07/22  21:15:21  Jenness
 * Initial revision
 *
*/
#include <stdio.h>
#include <stdlib.h>
#include "misc.h"

static char rcsid_RANDOM_C[] =
    "$Id: random.c 1.2 1992/08/01 21:17:28 Jenness Exp $";

/* local declarations */
static float u[98], c, cd, cm;
static int i97, j97;
static bool init00f = false;

/* exported functions */
proc srand00f(int ij, int kl)
{
/*
  NOTE: The seed variables can have values between:    0 <= IJ <= 31328
                                                       0 <= KL <= 30081
  Use IJ = 1802 & KL = 9373 to test the random number generator. The
  subroutine rand00f should be used to generate 20000 random numbers.
  Then display the next six random numbers generated multiplied by 4096*4096
  If the random number generator is working properly, the random numbers
  should be:
```

```
            6533892.0  14220222.0  7275067.0
            6172232.0  8354498.0   10633180.0
*/
int i, j, k, l, ii, jj, m;
float s, t;

if (ij<0 || ij>31328 || kl<0 || kl>30081) {
puts("\nERROR:(srand00f) seed values out of range.");
exit(1);
}

i = (ij/177)%177 + 2;
j = ij%177 + 2;
k = (kl/169)%178 + 1;
l = kl%169;

for (ii=1; ii<=97; ii++) {
s = 0.0;
t = 0.5;
for (jj=1; jj<=24; jj++) {
m = (((i*j)%179)*k) % 179;
i = j;
j = k;
k = m;
l = (53*l + 1) % 169;
if ((l*m)%64 >= 32) s += t;
t *= 0.5;
}
u[ii] = s;
}

c = 362436.0 / 16777216.0;
cd = 7654321.0 / 16777216.0;
cm = 16777213.0 / 16777216.0;

i97 = 97;
j97 = 33;

init00f = true;
}

float rand00f(void)
/*
  This is the random number generator proposed by George Marsaglia in
  Florida State University Report: FSU-SCRI-87-50
  It was slightly modified by F. James to produce an array of pseudorandom
  numbers.
*/
{
float uni;

if (init00f==false) {
```

```
puts("\nERROR:(rand00f) not initialized before use.\n");
exit(2);
        }
uni = u[i97] - u[j97];
if (uni < 0.0) uni += 1.0;
u[i97] = uni;
i97--;
if (i97==0) i97 = 97;
j97--;
if (j97==0) j97 = 97;
c -= cd;
if (c<0.0) c += cm;
uni -= c;
if (uni<0.0) uni += 1.0;
return uni;
}

/* local functions */
```

random.h The header for using the random number generators.

```
/*
        Random Number Generators Header File
        NOTES: Implements a suite of Pseudo-Random Number Generators,
               both float and integer types.  The naming scheme is as
               follows:
                        rand<digit><digit><type>
               <type> is f (float) or i (integer)
               <digit> is 0,1,...,9

               Macros are implemented to default to one of the RNG's
               which closely follows the ANSI C naming convention.
                       randi, srandi, randf, srandf

               Also for added readability the following macros are
               defined:
                       rrandi, rrandf

               These return a random number within a range.
                      lower <= rrand_(lower,upper) < upper

        $Log: random.h $
 * Revision 1.4  1992/08/11  21:38:37  Jenness
 * Added the macros to generate random numbers on an interval
 *
 * Revision 1.3  1992/08/01  21:17:28  Jenness
 * cosmetic changes
 *
 * Revision 1.2  1992/08/01  14:09:11  Jenness
 * cosmetic changes
 *
 * Revision 1.1  1992/07/22  21:14:50  Jenness
```

```
 * Initial revision
 *
*/

#ifndef _RANDOM_H
#define _RANDOM_H

static char rcsid_RANDOM_H[] =
    "$Id: random.h 1.4 1992/08/11 21:38:37 Jenness Exp $";

#include "misc.h"

#define randi rand
#define srandi srand
#define rrandi(l,u) (randi()%((u)-(l))+(l))
/*
  To use the ANSI C builtin function rand() use:
                    (float)rand()/(float)RAND MAX
*/
#define randf rand00f
#define srandf srand00f
#define rrandf(l,u) (randf()*((u)-(l))+(l))

proc srand00f ARGS((int ij, int kl));
float.rand00f(void);

#endif /* _RANDOM_H */
```

shell.c The shell for all of the coloring algorithms.

```
/* SHELL.C
 *      Description: This is a shell for dropping in both exact
 *                   and heuristic graph coloring functions.
 *                   DEFINE a macro COLORFUNC with the function
 *                   name you wish to test.  The prototype for
 *                   this function is:
 *
 * proc COLORFUNC (GROUP *gp, graph *g, colors *k, word *maxk, float *secs);
 */

#ifndef COLORFUNC
#error COLORFUNC undefined
#endif

#include <stdio.h>
#include "misc.h"
#include "stats.h"
#include "groupo.h"
#include "graph.h"

#define NOGRID -1
```

```
graph g;                         /* graph to be colored */
chroma c;                        /* chromaticity vector */
colors k;                        /* current coloring    */
word maxk;                       /* number of colors    */

float secs;                      /* time to color in secs */
statistic cstat, tstat;          /* color and time stats  */

int grid, gid;                   /* current group and graph id */
int newid;                       /* new group id               */

proc COLORFUNC (GROUP *gp, graph *g, colors *k, word *maxk, float *secs);

main ()
{
    GROUP *gp;                   /* group output         */
    FILE *in, *out;              /* input output files   */
    char input[40], output[40];  /* input output names   */
    int row;                     /* current row of input */

    /* get input output names */
    printf("\nEnter the input file name  > ");
    scanf("%s",input);
    if ((in = fopen(input, "r")) == NULL)
error(OPENFAIL, input);
printf("Enter the output file name > ");
scanf("%s",output);
if ((out = fopen(output, "w")) == NULL)
error(OPENFAIL, output);
gp = gopen(out);
printf("Do you wish to echo to the screen ? ");
if (getYN() == 'Y')
gfadd(gp, stdout);

row = 0;    /* no row has been read */
grid = NOGRID;  /* hopefully no group will have this id */
/* read until no more graphs in the input file */
while (InputDetails(in, &g, &c, &k, &row, &gid, &newid)) {

COLORFUNC (gp, &g, &k, &maxk, &secs);

if (newid != grid) { /* new group has been found */
if (grid != NOGRID) { /* don't print if grid is invalid */
gprintf(gp, "%s Statistics for coloring of group %d:\n",
COMMENT_TOK, grid);
gprintf(gp, "%s Maximum Colors:\n", COMMENT_TOK);
gprintStat(gp, cstat);
gprintf(gp, "%s Time Required:\n", COMMENT_TOK);
gprintStat(gp, tstat);
}
/* initialize for new group */
grid = newid;
```

```
gprintf(gp, "%s Beginning group %d of graphs.\n",
COMMENT_TOK, grid);
InitStat(&cstat);
InitStat(&tstat);
}

gprintf(gp, "%s Coloring graph %3d :", COMMENT_TOK, gid);
gprintf(gp, "used %3d colors; time required %7.2fs.\n", maxk, secs);
TallyStat(&cstat, (float)maxk);
TallyStat(&tstat, secs);
if (EchoDetails)
OutputDetails(gp, g);
gflush(gp);
/* cleanup memory */
DestroyColors(&k);
DestroyChroma(&c);
DestroyGraph(&g);
}
/* print out statistics on final group */
gprintf(gp, "%s Statistics for coloring of group %d:\n",
COMMENT_TOK, grid);
gprintf(gp, "%s Maximum Colors:\n", COMMENT_TOK);
    gprintStat(gp, cstat);
    gprintf(gp, "%s Time Required:\n", COMMENT_TOK);
    gprintStat(gp, tstat);
    fclose(in);
    gclose(gp);
    return 0;
}
```

stats.c Implements the statistics gathering functions.

```
/*
        Statistics C Package
        $Log: stats.c $
 * Revision 1.6  1992/08/22  00:45:55  Jenness
 * cosmetic changes to output by gprintStat
 *
 * Revision 1.5  1992/08/15  23:30:53  Jenness
 * minor changes
 *
 * Revision 1.4  1992/08/11  21:40:43  Jenness
 * Added group output to the print function
 * cosmetic changes
 *
 * Revision 1.3  1992/08/01  21:17:54  Jenness
 * cosmetic changes
 *
 * Revision 1.2  1992/08/01  14:51:49  Jenness
 * minor cleanup of code
 *
 * Revision 1.1  1992/07/24  21:23:33  Jenness
```

```
 * Initial revision
 *
*/

static char rcsid_STATS_C[] =
    "$Id: stats.c 1.6 1992/08/22 00:45:55 Jenness Exp $";

#include <stdio.h>      /* printf */
#include <float.h>      /* FLT_MAX FLT_MIN */
#include "misc.h"
#include "groupo.h"
#include "stats.h"

#ifdef __STDC__
proc _InitStat(statistic *v)
#else
proc _InitStat(v) statistic *v;
#endif
{
#ifndef NDEBUG
    printf("\nInitializing a simple statistical variable.");
#endif
    StatNum(*v) = 0;
    StatMax(*v) = FLT_MIN;
    StatMin(*v) = FLT_MAX;
    StatSum(*v) = 0.0;
    StatSSQ(*v) = 0.0;
    return;
}

#ifdef __STDC__
proc _TallyStat(statistic *v, float d)
#else
proc _TallyStat(v, d) statistic *v; float d;
#endif
{
#ifndef NDEBUG
    printf("\nRecording data item %G.",d);
#endif
    ++StatNum(*v);
    StatMin(*v) = min(StatMin(*v),d);
    StatMax(*v) = max(StatMax(*v),d);
    StatSum(*v) += d;
    StatSSQ(*v) += d*d;
    return;
}

#ifdef __STDC__
proc _gprintStat(GROUP *g, statistic v)
#else
proc _gprintStat(g, v) GROUP *g; statistic v;
#endif
```

```
{
    if (StatNum(v) > 0) {
        gprintf(g,"%s Number   of samples : %d\n", COMMENT_TOK, StatNum(v));
        gprintf(g,"%s Minimum  of samples : %0.4G\n", COMMENT_TOK,
                StatMin(v));
        gprintf(g,"%s Maximum  of samples : %0.4G\n", COMMENT_TOK,
                StatMax(v));
        gprintf(g,"%s Average  of samples : %0.4G\n", COMMENT_TOK,
                StatSum(v)/StatNum(v));
        gprintf(g,"%s Variance of samples : %0.4G\n", COMMENT_TOK,
                (StatSSQ(v)-(StatSum(v)*StatSum(v))/StatNum(v))/StatNum(v));
    } else
        gprintf(g,"%s No statistics recorded!\n", COMMENT_TOK);
    gflush(g);
    return;
}
```

stats.h Header for using the statistics functions.

```
/*
        Statistics Header File
        NOTES: Provides a way to gather simple statistics on a single
               variable.
        $Log: stats.h $
 * Revision 1.4  1992/08/15  23:31:02  Jenness
 * made gprintStat function into a macro call
 *
 * Revision 1.3  1992/08/11  21:40:15  Jenness
 * Added group output to the print function
 * cosmetic changes
 *
 * Revision 1.2  1992/08/01  21:17:54  Jenness
 * cosmetic changes
 *
 * Revision 1.1  1992/07/24  21:23:33  Jenness
 * Initial revision
 *
*/

#ifndef _STATS_H
#define _STATS_H

static char rcsid_STATS_H[] =
    "$Id: stats.h 1.4 1992/08/15 23:31:02 Jenness Exp $";

#include <stdio.h>
#include "misc.h"
#include "groupo.h"

#define StatNum(v) (v).n
#define StatMin(v) (v).min
#define StatMax(v) (v).max
```

```
#define StatSum(v) (v).sum
#define StatSSQ(v) (v).ssq

#ifndef NSTATS
#define InitStat(v) _InitStat(v)
#define TallyStat(v,d) _TallyStat(v,d)
#define gprintStat(g,v) _gprintStat(g,v)
#else /* NSTATS */
#define InitStat(v)
#define TallyStat(v,d)
#define gprintStat(g,v)
#endif /* NSTATS */

typedef struct {
    int n;
    float min;
    float max;
    float sum;
    float ssq;
} statistic;

proc _InitStat ARGS((statistic *v));
proc _TallyStat ARGS((statistic *v, float d));
proc _gprintStat ARGS((GROUP *g, statistic v));

#endif /* _STATS_H */
```

tabu.c Implements the Tabu Search technique.

```
/* TABU C
 *      Description: TABU Search Shell: This implements the functions
 *                   of the TABU search method.
 */

#include <stdlib.h>
#include <string.h>
#include <stdc.h>
#include <time.h>
#include "misc.h"
#include "tabu.h"

/* function pointers */
move (*MoveSelect)(move *pool, int size);
proc (*MoveToCfg)(config c, config d, move m);
proc (*StartCfg)(config c);
bool (*EndOfSearch)(void);

#include "uservar.c"

int aspiration, iteration = 0;
config currentcfg, bestcfg; /* configurations           */
int currentcost;    /* cost for the current iteration    */
```

```
int tsize;          /* tabu-list size   */
int thits;          /* stats on the number of tabu hits  */
move *mpool;     /* move pool for candidate selection */
int psize;       /* move pool size   */

proc TSShell (GROUP *gp, graph *g, colors *k, word *chi, float *secs)
{
  /* Initialize the problem specific structures and variables */
  ggp = gp;
  gr = g;
  ko = k;

    thits = 0;
    TabuSearch(chi, secs);
    gprintf(ggp, "%s Tabu List Hits  : %d\n", COMMENT_TOK, thits);
    gprintf(ggp, "%s Interations     : %d\n", COMMENT_TOK, iteration);

    return;
}

proc TabuSearch(word *best, float *secs)
{
config nextcfg, candidate;
int nextcost, potential;
move nextmove;
tabulist ts;

Initialize();

ts = CreateTL(tsize);

    CreateCfg(&bestcfg);
CreateCfg(&currentcfg);
    CreateCfg(&nextcfg);
CreateCfg(&candidate);

/* start timing */
    *secs = (float) clock ();

    StartCfg(currentcfg);
    currentcost = CostOfCfg(currentcfg,0);
    SaveCfg(currentcfg);

    aspiration = currentcost;
    trace("%d (initial)",currentcost);

    iteration = 0;
while(not EndOfSearch()) {
++iteration;
do { /* get the first move -- ASSUMING NOT EXHAUSTED */
nextmove = GenerateMove(currentcfg);
            /* Test if nbrhd exhausted -- used in sampling */
```

```
if (MoveCmp(nextmove, NullMove) == true)
nextmove = GenerateMove(currentcfg);
MoveToCfg(nextcfg,currentcfg,nextmove);
nextcost = CostOfCfg(nextcfg,firstpos(nextmove));
} while(TabuStatus(ts, nextmove) and nextcost >= aspiration);

        psize = 1;
        mpool[0] = nextmove;

        /* find the best neighbor */
nextmove = GenerateMove(currentcfg);
while (MoveCmp(nextmove, NullMove) == false) {
            MoveToCfg(candidate,currentcfg,nextmove);
            potential = CostOfCfg(candidate,firstpos(nextmove));

if (potential < nextcost)
if (not TabuStatus(ts, nextmove) or
potential < aspiration) {
psize = 1;
mpool[0] = nextmove;
nextcost = potential;
}
else if (potential == nextcost and
not TabuStatus(ts, nextmove))
mpool[psize++] = nextmove;

nextmove = GenerateMove(currentcfg);
}
        /* take the best move */
nextmove = MoveSelect(mpool,psize);

MoveToCfg(currentcfg,currentcfg,nextmove);
currentcost = CostOfCfg(currentcfg,firstpos(nextmove));

        /* update the best configuration */
if (currentcost < aspiration) {
aspiration = currentcost;
SaveCfg(currentcfg);
trace("%d",iteration);
trace("%d (update)",currentcost);
#ifndef NTRACE
        } else if (iteration%UPDATE == 0) {
            trace("%d",iteration);
            trace("%d",currentcost);
#endif
        }

UpdateTL(&ts, nextmove);
}
RestoreCfg(currentcfg);
*best = CostOfCfg(currentcfg,0);
```

```
/* stop timing */
*secs = ((float) clock () - *secs) / (float) CLK_TCK;

    DestroyCfg(&bestcfg);
DestroyCfg(&currentcfg);
DestroyCfg(&nextcfg);
DestroyCfg(&candidate);
    DestroyTL(&ts);
    free(mpool);

    CleanUp();

trace("%d (total)",iteration);
}

tabulist CreateTL (int d)
{
    int i;
tabulist t;
    move m;

    m = NullMove;
    t.duration = d;
t.moves = (move *)malloc(sizeof(move)*t.duration);
    for(i = 0; i < t.duration; i++)
t.moves[i] = m;
    t.next = 0;

    return t;
}

bool TabuStatus (tabulist t, move m)
{
int i;
    bool found;

for(i = 0, found = false; not found and i < t.duration; i++)
        if (MoveCmp(m, t.moves[i]) == true)
            found = true;

    if (found)
        ++thits;
    return found;
}

proc UpdateTL (tabulist *t, move m)
{
t->moves[t->next] = MoveReverse(m);
    t->next = (t->next + 1) % t->duration;

    return;
}
```

```
proc SetDuration (tabulist *t, int d)
{
int i, j;
move *m;

    m = (move *)malloc(sizeof(move)*d);
    i = t->next - min(d, t->duration);
    if (i < 0)
        i += t->duration;
    for (j = 0; j < min(d, t->duration); j++, i = (i + 1) % t->duration)
        m[j] = t->moves[i];
    if (d > t->duration) /* initialize all uninitialized moves */
for (j = t->duration; j < d; j++)
            m[j] = NullMove;
    free(t->moves);
    t->moves = m;
    t->next = min(d, t->duration);
    t->duration = d;

return;
}

proc DestroyTL (tabulist *t)
{
    free(t->moves);
    t->moves = NULL;
    t->next = 0;
    t->duration = 0;

    return;
}

proc SaveCfg (config c)
{
    memcpy(bestcfg, c, sizeof(short int)*SizeOfCfg);

return;
}

proc RestoreCfg (config c)
{
    memcpy(c, bestcfg, sizeof(short int)*SizeOfCfg);

    return;
}

#include "userfunc.c" /* the user contributed functions */
```

tabu.h The header file for the Tabu Search technique.

```
/* TABU.H
```

```
 *          Description - Configuration file for the Tabu Search Shell
 */

/* The record structures used in the search */

#define UPDATE 1
#define MAXPOS 4

typedef struct {
short int v[2];       /* vertex number          */
short int p[MAXPOS];  /* index of swap          */
} move;

typedef struct {
int duration;
move *moves;
int next;
} tabulist;

typedef short int *config;

#define randomize() srand((unsigned)time(NULL))
#define NullMove nullmove()
#define SizeOfCfg sizeofcfg

/* function declarations */
proc TabuSearch (word *best, float *secs);
proc Initialize (void);
proc CleanUp (void);

tabulist CreateTL (int d);
bool TabuStatus (tabulist t, move m);
proc UpdateTL (tabulist *t, move m);
proc SetDuration (tabulist *t, int d);
proc DestroyTL (tabulist *t);

proc CreateCfg (config *c);
proc SaveCfg (config c);
proc RestoreCfg (config c);
int CostOfCfg (config c, int p);
move nullmove (void);
move GenerateMove (config c);
bool MoveCmp (move m1, move m2);
move MoveReverse (move m);
proc DestroyCfg (config *c);
```

tabu2.c Implements the Tabu Search using the PositionRelocation neighborhood.

```
/* TABU2.C
 *      Description: TABU Search Version 2: This implements the functions
 *                   of the TABU search method specific to CGCP.
```

```
 */

#include <stdlib.h>
#include <string.h>
#include <stdc.h>
#include <time.h>
#include "groupo.h"
#include "graph.h"
#include "stats.h"
#include "misc.h"
#include "tabu2.h"

#define percent(a,b) (((a)*(b))/100)

/* some local types */
typedef struct
{
  short int vertex;   /* vertex number in adjacency matrix    */
  short int chroma;   /* vertex chromaticity                  */
  short int ucdeg;    /* uncolored adjacent chromatic degree  */
  short int ucadj;    /* number of adjacent uncolored nodes   */
} sorttype;

/* global variables that must be set in Initialize() */
GROUP *ggp; /* global pointer to the output group  */
graph *gr;  /* global pointer to graph  */
colors *ko;  /* global pointer to colors  */
int tabustyle;  /* designates Simple(0) or Prob(1)      */
int srate;      /* neighborhood sample rate in percent */
bool sdyna;     /* use dynamic sampling                */
int sdelta;     /* change of sampling rate             */
int sstop;      /* ceiling of sampling rate            */
bool tdyna;     /* use dynamic tabu list size          */
int tdelta;     /* change in tabu list size            */
int tstop;      /* ceiling of tabu list size           */
int iterations; /* the number of iterations to perform */
int i_count = 0;/* iteration counter for eos???()      */
move *nbrhd;  /* holds all moves in neighborhood  */
int nsize;  /* neighborhood size  */
int ssize; /* neighborhood sample size */
short int *pi; /* permuted index for sampling nbrhd */
int sizeofcfg; /* used for the define SizeOfCfg  */
statistic istat;/* iteration statistics                */
short int *pos; /* position array                      */
int changup;    /* when to make changes in dynamic vars */

/* some local functions */
bool eosSTA (void);
bool eosABS (void);
proc startLF2 (config c);
proc startCLF (config c);
proc startRAN (config c);
```

```
move selectRAN(move *p, int n);
move selectLST(move *p, int n);
move selectFST(move *p, int n);
int clfrule ARGS ((const void *a, const void *b));
int lf2rule (const void *a, const void *b);
word nextcolor ARGS ((graph g, int i));
proc CreateIndex (graph g, sorttype ** index,
  int (*rule)(const void *a, const void *b));
proc RandPerm ARGS ((short int *p, int n));
proc MakeTabu(move m);
proc StartUp (int ss, int ts);

int aspiration, iteration = 0;
config currentcfg, bestcfg; /* configurations            */
int currentcost;    /* cost for the current iteration    */
int tsize;          /* tabu-list size    */
int tslast;         /* index to the oldest tabu move     */
move *mpool;     /* move pool for candidate selection */
int psize;       /* move pool size   */
move NullMove = {-1,-1,-1};

/* function pointers */
typedef proc (*PROC)();
typedef move (*MFUNC)();
typedef bool (*BFUNC)();
move (*MoveSelect)(move *pool, int size);
proc (*StartCfg)(config c);
bool (*EndOfSearch)(void);
PROC  startfunc[] = {startRAN, startCLF, startLF2};
MFUNC selectfunc[] = {selectFST, selectLST, selectRAN};
BFUNC eosfunc[] = {eosABS, eosSTA};

char *tsstr[] = { "Simple", "Probabilistic" };
char *eosstr[] = { "Absolute", "Stabilized" };
char *scstr[] = { "Random", "CLFOrder", "LF2Order" };
char *msstr[] = { "First", "Last", "Random" };

proc TSShell (GROUP *gp, graph *g, colors *k, word *chi, float *secs)
{
config nextcfg, candidate;
int nextcost, potential;
move nextmove;

  /* Initialize the problem specific structures and variables */
  ggp = gp;
  gr = g;
  ko = k;

Initialize();

    CreateCfg(&bestcfg);
CreateCfg(&currentcfg);
```

```
    CreateCfg(&nextcfg);
CreateCfg(&candidate);

/* start timing */
    *secs = (float) clock ();

    StartCfg(currentcfg);
    currentcost = CostOfCfg(currentcfg);
    SaveCfg(currentcfg);

    aspiration = currentcost;
    trace("%d (initial)",currentcost);

    iteration = 0;
while(not EndOfSearch()) {
++iteration;
nextmove = GenerateMove(currentcfg);
MoveToCfg(nextcfg,currentcfg,nextmove);
nextcost = CostOfCfg(nextcfg);

        psize = 1;
        mpool[0] = nextmove;

        /* find the best neighbor */
nextmove = GenerateMove(currentcfg);
while (MoveCmp(nextmove, NullMove) == false) {
            MoveToCfg(candidate,currentcfg,nextmove);
            potential = CostOfCfg(candidate);

if (potential < nextcost) {
psize = 1;
mpool[0] = nextmove;
nextcost = potential;
} else if (potential == nextcost)
mpool[psize++] = nextmove;

nextmove = GenerateMove(currentcfg);
}
        /* take the best move */
nextmove = MoveSelect(mpool,psize);

MoveToCfg(currentcfg,currentcfg,nextmove);
currentcost = CostOfCfg(currentcfg);

        /* update the best configuration */
if (currentcost < aspiration) {
aspiration = currentcost;
SaveCfg(currentcfg);
trace("%d",iteration);
trace("%d (update)",currentcost);
#ifndef NTRACE
        } else if (iteration%UPDATE == 0) {
```

```
            trace("%d",iteration);
            trace("%d",currentcost);
#endif
        }
        MakeTabu(nextmove);
}
RestoreCfg(currentcfg);
    *chi = CostOfCfg(currentcfg);

/* stop timing */
*secs = ((float) clock () - *secs) / (float) CLK_TCK;

    DestroyCfg(&bestcfg);
DestroyCfg(&currentcfg);
DestroyCfg(&nextcfg);
DestroyCfg(&candidate);
    free(mpool);

    CleanUp();

trace("%d (total)",iteration);
    gprintf(ggp, "%s Interations     : %d\n", COMMENT_TOK, iteration);
}

proc MakeTabu(move m)
{
    short int temp;

    temp = pos[m.id];
    pos[m.id] = pos[tslast];
    pos[tslast] = temp;

    temp = SizeOfCfg - tsize;
    tslast = (tslast-temp+1)%tsize + temp;

    return;
}

proc SaveCfg (config c)
{
    memcpy(bestcfg, c, sizeof(short int)*SizeOfCfg);

return;
}

proc RestoreCfg (config c)
{
    memcpy(c, bestcfg, sizeof(short int)*SizeOfCfg);

    return;
}
```

```
/* Tabu Search User Functions */

proc Initialize (void)
{
int i, n;
static bool initialized = false;

if (not initialized) { /* initialize for a group of graphs */
        InitStat(&istat);  /* initialize the statistics for iteration */
     SizeOfCfg = GraphOrder(*gr);
/* Get the user input for this run */
printf("TABUSEARCH:\n Simple(0), Probabilistic(1)\n"
   "Enter the style of search      > ");
scanf("%d",&tabustyle);
gprintf(ggp,"%s Tabu Search            : %s ",
COMMENT_TOK, tsstr[tabustyle]);
printf("Enter initial tabu list size   > ");
scanf("%d",&tsize);
printf("Do you wish to use dynamic tabu list size? ");
tdyna = (getYN() == 'Y');
if (tdyna) {
printf("Enter the rate of change       > ");
scanf("%d",&tdelta);
     printf("Enter the stopping rate        > ");
     scanf("%d",&tstop);
            tstop = max(5,tstop);
            tstop = min(tstop,SizeOfCfg-5);
            if (tdelta > 0)
                tstop = max(tstop,tsize);
            else
                tstop = min(tstop,tsize);
            /* make sure that everything is okay */
            tdyna = (tdelta * (tstop - tsize - tdelta) >= 0);
        }
if (tabustyle == 1) {
printf("Enter the initial sample rate  > ");
scanf("%d",&srate);
     srate = max(1,srate);
     srate = min(srate,100);
printf("Do you wish to use dynamic sampling? ");
sdyna = (getYN() == 'Y');
if (sdyna) {
printf("Enter the rate of change       > ");
scanf("%d",&sdelta);
printf("Enter the stopping rate        > ");
scanf("%d",&sstop);
                sstop = max(0,sstop);
                sstop = min(sstop,100);
                if (sdelta > 0)
                    sstop = max(sstop,srate);
                else
                    sstop = min(sstop,srate);
```

```
                /* make sure that everything is okay */
                sdyna = (sdelta * (sstop - srate - sdelta) >= 0);
            }
}
gprintf(ggp,"\n%s Initial Tabu List Size : %d\n",
COMMENT_TOK, tsize);
if (tdyna == 1) {
gprintf(ggp,"%s Tabu List Size Change  : %d\n",
COMMENT_TOK, tdelta);
gprintf(ggp,"%s Final Tabu List Size   : %d\n",
COMMENT_TOK, tstop);
}
if (tabustyle == 1) {
gprintf(ggp,"%s Initial Sample Rate    : %d\n",
COMMENT_TOK, srate);
if (sdyna == 1) {
gprintf(ggp,"%s Sample Change Rate     : %d\n",
COMMENT_TOK, sdelta);
gprintf(ggp,"%s Stopping Sample Rate   : %d\n",
COMMENT_TOK, sstop);
}
}
gprintf(ggp,"%s Neighborhood           : %s\n",
COMMENT_TOK, "PositonRelocation");
printf("START CONFIGURATION:\n Random(0), CLFOrder(1), LF2Order"
   "(2)\nEnter the start configuration  > ");
scanf("%d", &n);
        StartCfg = startfunc[n]; /* initialize function */
gprintf(ggp,"%s Starting Configuration : %s\n",
COMMENT_TOK, scstr[n]);
printf("MOVE SELECTION:\n First(0), Last(1), Random(2)"
   "\nEnter the move selection       > ");
scanf("%d", &n);
        MoveSelect = selectfunc[n]; /* initialize function */
gprintf(ggp,"%s Move Selection         : %s\n",
COMMENT_TOK, msstr[n]);
printf("END OF SEARCH:\n Absolute(0), Stabilized(1)\n"
   "Enter the end of search       > ");
scanf("%d", &n);
        EndOfSearch = eosfunc[n]; /* initialize function */
gprintf(ggp,"%s End Of Search          : %s\n",
COMMENT_TOK, eosstr[n]);
if (n == 0)
printf("Enter the maximum iterations   > ");
else
printf("Enter iterations before stable > ");
scanf("%d",&iterations);
gprintf(ggp,"%s Number of iterations   : %d\n",
COMMENT_TOK, iterations);
        changup = iterations;
        if (sdyna or tdyna) {
            printf("Enter iterations before change > ");
```

```
            scanf("%d",&changup);
            changup = max(1,changup);
            changup = min(changup,iterations);
     gprintf(ggp,"%s Iterations to change   : %d\n",
     COMMENT_TOK, changup);

        }
        gflush(ggp);

initialized = true;
}
    /* set up for first search */
    nbrhd = mpool = NULL;
    pi = NULL;
    pos = (short int *)malloc(sizeof(short int)*SizeOfCfg);
    RandPerm(pos, SizeOfCfg);
    StartUp(srate, tsize);

CreateColors(gr,ko);
randomize();

return;
}

bool eosABS (void)
{
/* end search after an absolute number of iterations */
if (++i_count > iterations) {
        i_count = 0;
return true;
}

    return false;
}

bool eosSTA (void)
{
/* end search after no change in best cost */
    static int best = INT_MAX;

if (++i_count > iterations) { /* end of search */
i_count = 0;
        best = INT_MAX;
return true;
} else if (aspiration < best) { /* not stable */
i_count = 1;
            best = aspiration;
    }

return false;
}
```

```
proc CleanUp (void)
{
    TallyStat(&istat, (float)iteration);
    gprintf(ggp, "%s Total Iterations:\n", COMMENT_TOK);
    gprintStat(ggp, istat);
free(nbrhd);
free(pi);
    free(pos);
}

bool MoveCmp (move m1, move m2)
{
return (bool)(memcmp(&m1, &m2, sizeof(move)) == 0);
}

proc CreateCfg (config *c)
{
*c = (config)malloc(sizeof(short int)*SizeOfCfg);

return;
}

int CostOfCfg (config c)
{
int i, j, m, beg;

BegColor (ColorsOf(*gr), c[0]) = 1;
m = EndColor (ColorsOf(*gr), c[0]);
    for (i = 1; i < GraphOrder(*gr); ++i) {
beg = 1;
redo:
for (j = 0; j < i; j++)
if (Adjacent(*gr,c[i],c[j]) and beg <=
EndColor(ColorsOf(*gr),c[j]) and
BegColor(ColorsOf(*gr),c[j]) <
beg+NodeChroma(ChromaOf(*gr),c[i])) {
beg = EndColor (ColorsOf (*gr), c[j]) + 1;
goto redo;
}
BegColor(ColorsOf(*gr),c[i]) = beg;
m = max(m, EndColor(ColorsOf(*gr), c[i]));
}

return m;
}

proc StartUp (int ss, int ts)
{
    srate = ss;
    tsize = ts;
    tslast = SizeOfCfg - tsize;
/*  Don't allow the vertex to remain fixed
```

```
    nsize = SizeOfCfg - tsize - 1;
*/
    nsize = SizeOfCfg - tsize;
    nbrhd = (move *)realloc(mpool, sizeof(move)*nsize);
mpool = (move *)realloc(mpool, sizeof(move)*nsize);
    if (tabustyle == 1) {
        ssize = percent(nsize,srate);
        pi = (short int *)realloc(pi, sizeof(short int)*nsize);
    } else
        ssize = nsize;

    return;
}

proc MoveToCfg (config c, config d, move m)
{
    short int temp;

memmove(c, d, sizeof(short int)*SizeOfCfg);
temp = d[m.p[1]];
c[m.p[1]] = d[m.p[0]];
c[m.p[0]] = temp;

return;
}

move GenerateMove (config c)
{
static int n=-1; /* the next move to make */
int i, j;

    /* reinitialize for tabulist size or sampling */
    if (n < 0) { /* beginning of an iteration */
        if (i_count%changup == 0) { /* time for a change */
            if (sdyna or tdyna) { /* do dynamic changes */
                trace("%d",iteration);
                if (tdyna) { /* check conditions for tabu list change */
                    tsize += tdelta;
                    tdyna = (tdelta * (tstop - tsize - tdelta) >= 0);
                    trace("%d",tsize);
                }
                if (sdyna) { /* check conditions for sample rate change */
                    srate += sdelta;
                    ssize = percent(nsize,srate);
                    sdyna =  (sdelta * (sstop - srate - sdelta) >= 0);
                    trace("%d",srate);
                }
                StartUp(srate, tsize);
         }
        }
        /* initialize the neighborhood */
        i = rand()%(SizeOfCfg - tsize - 1);
```

```
        n = 0;
/* Don't allow the vertex to remain fixed
        for (j = 0; j < i; ++j) {
            nbrhd[n].id = j;
            nbrhd[n].p[0] = pos[i];
            nbrhd[n].p[1] = pos[j];
            ++n;
        }
        for (j = i+1; j < SizeOfCfg - tsize; ++j) {
*/
        for (j = 0; j < SizeOfCfg - tsize; ++j) {
            nbrhd[n].id = j;
            nbrhd[n].p[0] = pos[i];
            nbrhd[n].p[1] = pos[j];
            ++n;
        }
        n = -1;
     /* generate a new permuted index for each configuration */
    if (tabustyle == 1) /* check if probabilistic tabu */
    RandPerm(pi, nsize);
    }

++n;

    if (tabustyle == 1) /* use the permuted index */
        i = pi[n];
    else
        i = n;
if (n >= ssize) { /* only upto the sample size */
n = -1;
return NullMove;
} else
return nbrhd[i];
}

move selectFST(move *p, int n)
{
return p[0];
}

move selectLST(move *p, int n)
{
return p[n-1];
}

move selectRAN(move *p, int n)
{
return p[rand()%n];
}

proc DestroyCfg(config *c)
{
```

```
free(*c);
*c = NULL;

return;
}

proc startRAN (config c)
{
   RandPerm(c,SizeOfCfg);

   return;
}

proc startCLF (config c)
{
   int i;
   sorttype *index;

   CreateIndex(*gr, &index, clfrule);
   for (i = 0; i < GraphOrder(*gr); i++)
  c[i] = index[i].vertex;
   free(index);

   return;
}

proc startLF2 (config c)
{
   int i;
   sorttype *index;

   CreateIndex(*gr, &index, lf2rule);
   for (i = 0; i < GraphOrder(*gr); i++)
  c[i] = index[i].vertex;
   free(index);

   return;
}

int clfrule (const void *a, const void *b)
{
if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
if (((sorttype *) a)->ucadj == ((sorttype *) b)->ucadj)
return 0;
else
return ((sorttype *) b)->ucadj - ((sorttype *) a)->ucadj;
else
return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
else
return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
}
```

```
int lf2rule (const void *a, const void *b)
{
if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
return 0;
else
return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
else
return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
}

proc CreateIndex (graph g, sorttype ** index,
int (*rule)(const void *a, const void *b))
{
int i, j;

*index = (sorttype *) malloc (sizeof (sorttype) * GraphOrder (g));
for (i = 0; i < GraphOrder (g); i++) {
(*index)[i].vertex = i;
(*index)[i].chroma = NodeChroma (ChromaOf (g), i);
(*index)[i].ucdeg = 0;
(*index)[i].ucadj = 0;
}
for (i = 0; i < GraphOrder (g); i++)
for (j = i + 1; j < GraphOrder (g); j++)
if (Adjacent (g, i, j)) {
(*index)[i].ucdeg += NodeChroma (ChromaOf (g), j);
(*index)[j].ucdeg += NodeChroma (ChromaOf (g), i);
(*index)[i].ucadj += 1;
(*index)[j].ucadj += 1;
}
qsort(*index, GraphOrder(g), sizeof(sorttype), rule);
return;
}

/* Generate a random permutation of {0,..,n-1} */
proc RandPerm(short int *p, int n)
{
int i, j, temp;

for(i = 0; i < n; i++)
p[i] = i;
for(i = n; i > 1; i--) {
j = rand()%i;
temp = p[i-1];
p[i-1] = p[j];
p[j] = temp;
}

return;
}
```

tabu2.h The header file for the Tabu Search using the PositionRelocation neighborhood.

```
/* TABU2.H
 *          Description - Configuration file for the Tabu Search Shell V.2
 */

/* The record structures used in the search */

#define UPDATE 1
#define MAXPOS 2
typedef struct {
        short int id;       /* index into pos array   */
short int p[MAXPOS];/* index of swap         */
} move;

typedef short int *config;

#define randomize() srand((unsigned)time(NULL))
#define SizeOfCfg sizeofcfg

/* function declarations */
proc Initialize (void);
proc CleanUp (void);

proc CreateCfg (config *c);
proc SaveCfg (config c);
proc RestoreCfg (config c);
int CostOfCfg (config c);
proc MoveToCfg (config c, config d, move m);
move GenerateMove (config c);
bool MoveCmp (move m1, move m2);
move MoveReverse (move m);
proc DestroyCfg (config *c);
```

userfunc.c Implements the user contributed functions to model composite graphs for Tabu Search.

```
/* Tabu Search User Functions */

proc Initialize (void)
{
int i, n;
static bool initialized = false;
bool intvneeded = false; /* interval needed */
    bool fencneeded = false; /* fence needed */

if (not initialized) { /* initialize for a group of graphs */
        InitStat(&istat);  /* initialize the statistics for iteration */
```

```
     SizeOfCfg = GraphOrder(*gr);
/* Get the user input for this run */
printf("TABUSEARCH:\n Simple(0), Probabilistic(1)\n"
   "Enter the style of search      > ");
scanf("%d",&tabustyle);
gprintf(ggp,"%s Tabu Search            : %s ",
COMMENT_TOK, tsstr[tabustyle]);
printf("Enter initial tabu list size   > ");
scanf("%d",&tsize);
if (tabustyle == 1) {
printf("Enter the initial sample rate  > ");
scanf("%d",&sstart);
     sstart = max(1,sstart);
     sstart = min(sstart,100);
printf("Do you wish to use dynamic sampling? ");
sdyna = (getYN() == 'Y');
if (sdyna) {
printf("Enter the rate of change       > ");
scanf("%d",&sdelta);
printf("Enter the stopping rate        > ");
scanf("%d",&sstop);
                sstop = max(0,sstop);
                sstop = min(sstop,100);
                if (sdelta > 0)
                    sstop = max(sstop,sstart);
                else
                    sstop = min(sstop,sstart);
                /* make sure that everything is okay */
                sdyna = (sdelta * (sstop - sstart - sdelta) >= 0);
            }
}
printf("NEIGHBORHOODS:\n PairWiseSwaps(0), Combinations(1), "
   "VertexCycling(2),\n ColorReductions(3), VertexInsertion(4)"
   ", Shuffling(5),\n FenceHopping(6)\n"
               "Enter the neighborhood search  > ");
scanf("%d", &search[0]);
nsearches = 1;
intvneeded = (search[0] == 0 or search[0] == 1 or
                      search[0] == 5 or search[0] == 6);
        fencneeded = (search[0] == 6);
printf("Do you wish to use multiple neighborhoods? ");
if (getYN() == 'Y') {
printf("Enter the number of additional > ");
scanf("%d",&n);
n = max(0,n);
n = min(n,MAXSEARCHES);
nsearches += n;
            if (nsearches > 1)
     gprintf(ggp,"with Multiple Neighborhoods ");
for (i = 1; i <= n; ++i) {
printf("Enter neighborhood search %2d   > ",i);
scanf("%d",&search[i]);
```

```
intvneeded = (intvneeded or
search[i] == 0 or search[i] == 1 or
                    search[i] == 5 or search[i] == 6);
                fencneeded = (fencneeded or search[i] == 6);
}
        }
        rmax = -1;
printf("Do you wish to use multistart method? ");
if (getYN() == 'Y') {
printf("Enter max number of restarts   > ");
scanf("%d",&rmax);
            rmax = max(1, rmax);
    gprintf(ggp,"with Multistart");
}
gprintf(ggp,"\n%s Initial Tabu List Size : %d\n",
COMMENT_TOK, tsize);
if (tabustyle == 1) {
gprintf(ggp,"%s Initial Sample Rate    : %d\n",
COMMENT_TOK, sstart);
if (sdyna == 1) {
gprintf(ggp,"%s Sample Change Rate     : %d\n",
COMMENT_TOK, sdelta);
gprintf(ggp,"%s Stopping Sample Rate   : %d\n",
COMMENT_TOK, sstop);
}
}
gprintf(ggp,"%s Neighborhood           : %s",
COMMENT_TOK, nbrstr[search[0]]);
for (i = 1; i < nsearches; ++i)
gprintf(ggp,", %s", nbrstr[search[i]]);
        gprintf(ggp,"\n");
if (rmax > 0)
gprintf(ggp,"%s Maximum restarts       : %d\n",
COMMENT_TOK, rmax);
interval = 2; /* the default size for i_temp */
if (intvneeded) {
printf("Enter the interval size        > ");
scanf("%d",&istart);
istart = max(2,istart);
istart = min(istart,SizeOfCfg);
gprintf(ggp,"%s Interval Size          : %d\n",
COMMENT_TOK, istart);
printf("Do you wish to use dynamic intervals? ");
idyna = (getYN() == 'Y');
if (idyna) {
printf("Enter the size of change       > ");
scanf("%d",&idelta);
gprintf(ggp,"%s Interval Delta         : %d\n",
COMMENT_TOK, idelta);
printf("Enter the stopping size        > ");
scanf("%d",&istop);
                istop = max(2,istop);
```

```
                istop = min(istop,SizeOfCfg);
                if (idelta > 0)
                    istop = max(istop,istart);
                else
                    istop = min(istop,istart);
gprintf(ggp,"%s Stopping Interval Size : %d\n",
COMMENT_TOK, istop);
                /* make sure that everything is okay */
                idyna = (idelta * (istop - istart - idelta) >= 0);
}
}
        pc = NULL;
        if (fencneeded) {
            printf("FENCES:\n FixedPosition(0), RandomPosition(1),\n"
                   " ForwardCycle(2), BackwardCycle(3),\n PermutedCycle(4)\n"
                   "Enter the fence type           > ");
            scanf("%d", &fence);
            CalcFence = fenfunc[fence];
            pc = (short int *)malloc(sizeof(short int)*(SizeOfCfg-2));
            RandPerm(pc,SizeOfCfg-2);
            gprintf(ggp,"%s Fence Type              : %s\n",
                        COMMENT_TOK, fenstr[fence]);
            if (fence == 0) {
                printf("Enter the fence position       > ");
                scanf("%d", &fence);
                fence = max(1,fence);
                fence = min(fence,SizeOfCfg-1);
                gprintf(ggp,"%s Fence Position          : %d\n",
                            COMMENT_TOK, fence);
            }
        }
printf("START CONFIGURATION:\n Random(0), CLFOrder(1), LF2Order"
   "(2)\nEnter the start configuration  > ");
scanf("%d", &n);
        StartCfg = startfunc[n]; /* initialize function */
gprintf(ggp,"%s Starting Configuration : %s\n",
COMMENT_TOK, scstr[n]);
printf("MOVE SELECTION:\n First(0), Last(1), Random(2)"
   "\nEnter the move selection       > ");
scanf("%d", &n);
        MoveSelect = selectfunc[n]; /* initialize function */
gprintf(ggp,"%s Move Selection         : %s\n",
COMMENT_TOK, msstr[n]);
printf("END OF SEARCH:\n Absolute(0), Stabilized(1)\n"
   "Enter the end of search        > ");
scanf("%d", &n);
        EndOfSearch = eosfunc[n]; /* initialize function */
gprintf(ggp,"%s End Of Search          : %s\n",
COMMENT_TOK, eosstr[n]);
if (n == 0)
printf("Enter the maximum iterations   > ");
else
```

```
printf("Enter iterations before stable > ");
scanf("%d",&iterations);
gprintf(ggp,"%s Number of iterations   : %d\n",
COMMENT_TOK, iterations);
        changup = iterations;
        if (sdyna or idyna or nsearches > 1 or rmax > 0) {
            printf("Enter iterations before change > ");
            scanf("%d",&changup);
            changup = max(1,changup);
            changup = min(changup,iterations);
     gprintf(ggp,"%s Iterations to change   : %d\n",
     COMMENT_TOK, changup);

        }
        gflush(ggp);

initialized = true;
}
    /* set up for first search */
    nbrhd = mpool = NULL;
    pi = NULL;
    r_count = 0; /* initialize refresh count for multistart */
    StartUp(sstart, sdyna, istart, idyna, 0);
i_temp = (short int *)malloc(sizeof(short int)*MAXPOS);

CreateColors(gr,ko);
randomize();

return;
}

proc initPWS (void)
{
    int i,j,n;

nsize=(interval-1)*(SizeOfCfg-interval+1)+((interval-2)*(interval-1))/2;
nbrhd = (move *)realloc(nbrhd, sizeof(move)*nsize);
n = 0;
for (i = 0; i < SizeOfCfg; ++i)
for (j = i+1; j < i+interval && j < SizeOfCfg; ++j) {
nbrhd[n].p[0] = i;
nbrhd[n].p[1] = j;
++n;
}

return;
}

proc initCO (void)
{
    int i,j,n,psize;
    short int *perms;
```

```
psize = ifact(interval);
perms = AllPerms(interval);
nsize=(psize-1)*(SizeOfCfg-interval+1);
nbrhd = (move *)realloc(nbrhd, sizeof(move)*nsize);
n = 0;
for (i = 0; i < SizeOfCfg-interval+1; ++i)
for (j = 0; j < psize-1; ++j) {
nbrhd[n].v[0] = i;
memmove(&(nbrhd[n].p),&perms[j*interval],sizeof(short int)*interval);
++n;
}
free(perms);

return;
}

proc initVC(void)
{
    int i,n;

nsize=2*(SizeOfCfg-1)-1;
nbrhd = (move *)realloc(nbrhd, sizeof(move)*nsize);
n = 0;
/* swap the first with any other position */
for (i = 1; i < SizeOfCfg; ++i) {
nbrhd[n].p[0] = 0;
nbrhd[n].p[1] = i;
++n;
}
/* swap the last with any other position */
for (i = 1; i < SizeOfCfg-1; ++i) {
nbrhd[n].p[0] = i;
nbrhd[n].p[1] = SizeOfCfg-1;
++n;
}
    iflag = false;

return;
}

proc initCR (void)
{
nsize = bsize = (SizeOfCfg*(SizeOfCfg-1))/2;
nbrhd = (move *)realloc(nbrhd, sizeof(move)*nsize);
    iflag = false;

return;
}

proc initVI (void)
{
```

```
    int i,n;

nsize=2*(SizeOfCfg-1);
nbrhd = (move *)realloc(nbrhd, sizeof(move)*nsize);
n = 0;
/* insert the first into any other position */
for (i = 1; i < SizeOfCfg; ++i) {
nbrhd[n].p[0] = 0;
nbrhd[n].p[1] = i;
++n;
}
/* insert the last into any other position */
for (i = 0; i < SizeOfCfg-1; ++i) {
nbrhd[n].p[0] = SizeOfCfg-1;
nbrhd[n].p[1] = i;
++n;
}
    iflag = false;

return;
}

proc initSH (void)
{
    int i,j,n,m;

    m = (interval-1)/2;
nsize=2*(m*SizeOfCfg-(m*(m+1))/2)-SizeOfCfg+1;
nbrhd = (move *)realloc(nbrhd, sizeof(move)*nsize);
n = 0;
for (i = 0; i < SizeOfCfg; ++i)
for (j = max(i-m,0); j < min(i+m+1,SizeOfCfg); ++j)
if (j != i-1 and j != i) {
nbrhd[n].p[0] = i;
nbrhd[n].p[1] = j;
++n;
}

return;
}

proc initFH (void)
{
    nsize = (SizeOfCfg*SizeOfCfg)/4; /* odd(SizeOfCfg) == false */
nbrhd = (move *)realloc(nbrhd, sizeof(move)*nsize);

    return;
}

bool eosABS (void)
{
/* end search after an absolute number of iterations */
```

```
if (++i_count > iterations) {
        i_count = 0;
return true;
}

    return false;
}

bool eosSTA (void)
{
/* end search after no change in best cost */
    static int best = INT_MAX;

if (++i_count > iterations) { /* end of search */
i_count = 0;
        best = INT_MAX;
return true;
} else if (aspiration < best) { /* not stable */
i_count = 1;
            best = aspiration;
    }

return false;
}

proc CleanUp (void)
{
    TallyStat(&istat, (float)iteration);
    gprintf(ggp, "%s Total Iterations:\n", COMMENT_TOK);
    gprintStat(ggp, istat);
free(nbrhd);
free(i_temp);
    free(pc);
if (tabustyle == 1)
free(pi);
}

bool MoveCmp (move m1, move m2)
{
return (bool)(memcmp(&m1, &m2, sizeof(move)) == 0);
}

move nullmove (void)
{
    int i;
move m;

    for (i = 0; i < MAXPOS; ++i)
    m.p[i] = -1;
m.v[0] = m.v[1] = -1;

return m;
```

```
}

move MoveReverse (move m)
{
move r = m;

if (search[sidx] == 0 or search[sidx] == 2 or search[sidx] == 6) {
r.v[0] = m.v[1];
r.v[1] = m.v[0];
}

return r;
}

proc CreateCfg (config *c)
{
*c = (config)malloc(sizeof(short int)*SizeOfCfg);

return;
}

int firstpos(move m)
{
    if (search[sidx] == 1)
        return m.v[0];
    else
        return min(m.p[0],m.p[1]);
}

int CostOfCfg (config c, int p)
{
int i, j, m, beg;

BegColor (ColorsOf(*gr), c[0]) = 1;
m = EndColor (ColorsOf(*gr), c[0]);
    for (i = 1; i < max(p,1); ++i)
        m = max(m, EndColor(ColorsOf(*gr), c[i]));
    for (i = max(p,1); i < GraphOrder(*gr); ++i) {
beg = 1;
redo:
for (j = 0; j < i; j++)
if (Adjacent(*gr,c[i],c[j]) and beg <=
EndColor(ColorsOf(*gr),c[j]) and
BegColor(ColorsOf(*gr),c[j]) <
beg+NodeChroma(ChromaOf(*gr),c[i])) {
beg = EndColor (ColorsOf (*gr), c[j]) + 1;
goto redo;
}
BegColor(ColorsOf(*gr),c[i]) = beg;
m = max(m, EndColor(ColorsOf(*gr), c[i]));
}
```

```
return m;
}

proc movePWS (config c, config d, move m)
{
memmove(c, d, sizeof(short int)*SizeOfCfg);
i_temp[0] = d[m.p[1]];
i_temp[1] = d[m.p[0]];
c[m.p[0]] = i_temp[0];
c[m.p[1]] = i_temp[1];

return;
}

proc moveCO (config c, config d, move m)
{
int i;

memmove(c, d, sizeof(short int)*SizeOfCfg);
for (i = 0; i < interval; ++i)
i_temp[i] = d[m.v[0]+m.p[i]];
for (i = 0; i < interval; ++i)
c[m.v[0]+i] = i_temp[i];

return;
}

proc moveSH (config c, config d, move m)
{
memmove(c, d, sizeof(short int)*SizeOfCfg);
if (m.p[0] < m.p[1])
memmove(&c[m.p[0]],&c[m.p[0]+1],sizeof(short int)*(m.p[1]-m.p[0]));
else
memmove(&c[m.p[1]+1],&c[m.p[1]],sizeof(short int)*(m.p[0]-m.p[1]));
c[m.p[1]] = m.v[0];

return;
}

proc StartUp (int ss, bool sd, int is, bool id, int x)
{
    sidx = x;
    srate = ss;
    interval = is;
    sflag = sd;
    iflag = id;
    MoveToCfg = movefunc[search[sidx]]; /* initialize function */
    initfunc[search[sidx]](); /* call the nbrhd initialization */
mpool = (move *)realloc(mpool, sizeof(move)*nsize);
    if (tabustyle == 1) {
        ssize = percent(nsize,srate);
        pi = (short int *)realloc(pi, sizeof(short int)*nsize);
```

```
    } else
        ssize = nsize;

    return;
}

move GenerateMove (config c)
{
static int n=-1; /* the next move to make */
int idx;

    /* reinitialize for nbrhd, multistart, multisearch, or sampling */
    if (n < 0) { /* beginning of an iteration */
        if (i_count%changup == 0) { /* time for a change */
            if (sflag or iflag) { /* do dynamic changes first */
                trace("%d",iteration);
                if (iflag) { /* check conditions for interval change */
                    StartUp(srate, sflag, interval+idelta, iflag, sidx);
                    iflag = (idelta * (istop - interval - idelta) >= 0);
                    trace("%d",interval);
                }
                if (sflag) { /* check conditions for sample rate change */
                    srate += sdelta;
                    ssize = percent(nsize,srate);
                    sflag =  (sdelta * (sstop - srate - sdelta) >= 0);
                    trace("%d",srate);
                }
            } else if (sidx < nsearches - 1) { /* check for multiple search */
                StartUp(sstart, sdyna, istart, idyna, sidx+1);
                RestoreCfg(c);
                currentcost = CostOfCfg(c,0);
                trace("%d",iteration);
                trace("%s",nbrstr[search[sidx]]);
         } else if (r_count < rmax + 1) { /* check for multistart */
                ++r_count;
                StartUp(sstart, sdyna, istart, idyna, 0);
         RandPerm(c,SizeOfCfg);
         currentcost = CostOfCfg(c,0); /* be sure to calculate new cost */
                trace("%d",iteration);
        trace("%d (refresh)", currentcost);
         }
        }
        /* initialize the neighborhood */
        genfunc[search[sidx]](c);
     /* generate a new permuted index for each configuration */
    if (tabustyle == 1) /* check if probabilistic tabu */
    RandPerm(pi, nsize);
    }

++n;

    if (tabustyle == 1) /* use the permuted index */
```

```
        idx = pi[n];
    else
        idx = n;
if (n >= ssize) { /* only upto the sample size */
n = -1;
return NullMove;
} else
return nbrhd[idx];
}

proc genPWS (config c)
{
    int i,j,n;

n = 0;
for (i = 0; i < SizeOfCfg; ++i)
for (j = i+1; j < i+interval && j < SizeOfCfg; ++j) {
nbrhd[n].v[0] = c[i];
nbrhd[n].v[1] = c[j];
++n;
}

return;
}

proc genCO (config c)
{
    /* do nothing */
return;
}

proc genVC (config c)
{
    int i,n;

n = 0;
/* swap the first with any other position */
for (i = 1; i < SizeOfCfg; ++i) {
nbrhd[n].v[0] = c[0];
nbrhd[n].v[1] = c[i];
++n;
}
/* swap the last with any other position */
for (i = 1; i < SizeOfCfg-1; ++i) {
nbrhd[n].v[0] = c[i];
nbrhd[n].v[1] = c[SizeOfCfg-1];
++n;
}

return;
}
```

```
proc genCR (config c)
{
    int i,j,n;

n = 0;
/* look for the vertices with the highest color assigned */
for (i = 0; i < SizeOfCfg; ++i)
if (EndColor(ColorsOf(*gr),c[i]) == currentcost)
for (j = 0; j < SizeOfCfg; ++j)
if (j != i) {
if (n >= bsize) { /* grab more memory */
bsize += BLOCK;
nbrhd = (move *)realloc(nbrhd,sizeof(move)*bsize);
mpool = (move *)realloc(mpool,sizeof(move)*bsize);
}
nbrhd[n].v[0] = c[i];
nbrhd[n].p[0] = i;
nbrhd[n].p[1] = j;
++n;
}
nsize = n;
/* trace("%d",nsize); */
if (tabustyle == 1) {
pi = (short int *)realloc(pi,sizeof(short int)*nsize);
ssize = percent(nsize,srate);
    } else
        ssize = nsize;
    /* trace("%d",ssize); */

return;
}

proc genVI (config c)
{
    int i,n;

n = 0;
/* insert the first into any other position */
for (i = 1; i < SizeOfCfg; ++i) {
nbrhd[n].v[0] = c[0];
++n;
}
/* insert the last into any other position */
for (i = 0; i < SizeOfCfg-1; ++i) {
nbrhd[n].v[0] = c[SizeOfCfg-1];
++n;
}

return;
}

proc genSH (config c)
```

```
{
    int i,j,n,m;

    m = (interval-1)/2;
n = 0;
for (i = 0; i < SizeOfCfg; ++i)
for (j = max(i-m,0); j < min(i+m+1,SizeOfCfg); ++j)
if (j != i-1 and j != i) {
nbrhd[n].v[0] = c[i];
++n;
}

return;
}

proc genFH (config c)
{
    int i,j,n;
    int leftbound, rightbound;

    fence = CalcFence(fence);
    leftbound = max(0,fence-interval/2);
    leftbound = min(leftbound,SizeOfCfg-interval);
    rightbound = min(fence+interval/2,SizeOfCfg);
    rightbound = max(rightbound,interval);
    n = 0;
    /* generate all possible swaps over the fence */
    for (i = leftbound; i < fence; ++i)
        for (j = fence; j < rightbound; ++j) {
nbrhd[n].p[0] = i;
nbrhd[n].p[1] = j;
nbrhd[n].v[0] = c[i];
nbrhd[n].v[1] = c[j];
            ++n;
        }
    nsize = n;

if (tabustyle == 1) {
pi = (short int *)realloc(pi,sizeof(short int)*nsize);
ssize = percent(nsize,srate);
    } else
        ssize = nsize;

    return;
}

int fenceFP (int fence)
{
    return fence;
}

int fenceRP (int fence)
```

```
{
    return rand()%(SizeOfCfg-1) + 1;
}

int fenceFC (int fence)
{
    return fence%(SizeOfCfg-1) + 1;
}

int fenceBC (int fence)
{
    --fence;
    if (fence < 0)
        fence = SizeOfCfg-1;
    return fence;
}

int fencePC (int fence)
{
    static int n = -1;

    ++n;
    if (n > SizeOfCfg-2) {
        n = 0;
        RandPerm(pc,SizeOfCfg-2);
    }

    return pc[n]+1;
}

move selectFST(move *p, int n)
{
return p[0];
}

move selectLST(move *p, int n)
{
return p[n-1];
}

move selectRAN(move *p, int n)
{
return p[rand()%n];
}

proc DestroyCfg(config *c)
{
free(*c);
*c = NULL;

return;
}
```

```
proc startRAN (config c)
{
   RandPerm(c,SizeOfCfg);

   return;
}

proc startCLF (config c)
{
   int i;
   sorttype *index;

   CreateIndex(*gr, &index, clfrule);
   for (i = 0; i < GraphOrder(*gr); i++)
  c[i] = index[i].vertex;
   free(index);

   return;
}

proc startLF2 (config c)
{
   int i;
   sorttype *index;

   CreateIndex(*gr, &index, lf2rule);
   for (i = 0; i < GraphOrder(*gr); i++)
  c[i] = index[i].vertex;
   free(index);

   return;
}

int clfrule (const void *a, const void *b)
{
if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
if (((sorttype *) a)->ucadj == ((sorttype *) b)->ucadj)
return 0;
else
return ((sorttype *) b)->ucadj - ((sorttype *) a)->ucadj;
else
return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
else
return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
}

int lf2rule (const void *a, const void *b)
{
if (((sorttype *) a)->ucdeg == ((sorttype *) b)->ucdeg)
if (((sorttype *) a)->chroma == ((sorttype *) b)->chroma)
```

```
return 0;
else
return ((sorttype *) b)->chroma - ((sorttype *) a)->chroma;
else
return ((sorttype *) b)->ucdeg - ((sorttype *) a)->ucdeg;
}

proc CreateIndex (graph g, sorttype ** index,
int (*rule)(const void *a, const void *b))
{
int i, j;

*index = (sorttype *) malloc (sizeof (sorttype) * GraphOrder (g));
for (i = 0; i < GraphOrder (g); i++) {
(*index)[i].vertex = i;
(*index)[i].chroma = NodeChroma (ChromaOf (g), i);
(*index)[i].ucdeg = 0;
(*index)[i].ucadj = 0;
}
for (i = 0; i < GraphOrder (g); i++)
for (j = i + 1; j < GraphOrder (g); j++)
if (Adjacent (g, i, j)) {
(*index)[i].ucdeg += NodeChroma (ChromaOf (g), j);
(*index)[j].ucdeg += NodeChroma (ChromaOf (g), i);
(*index)[i].ucadj += 1;
(*index)[j].ucadj += 1;
}
qsort(*index, GraphOrder(g), sizeof(sorttype), rule);
return;
}

int ifact (int i)
{
   int f = 1;
   for (;i > 1; --i)
f *= i;
   return f;
}

/* Generate all permutations of {0,..,n-1} */
short int *AllPerms (int n)
{
int i,j,k,rows,cols,nrows,ncols;
    short int *perms, *nperms;

perms = NULL;
if (n > 0) {
rows = 1; cols = 1;
perms = (short int *)malloc(sizeof(short int));
perms[0] = 0;
while (cols < n) {
ncols = cols+1;
```

```
nrows = ncols*rows;
nperms = (short int *)malloc(ncols*nrows*sizeof(short int));
for (i = 0; i < rows; ++i)
for (j = 0; j < ncols; ++j) {
nperms[(i*ncols+j)*ncols+cols] = j;
for (k = 0; k < cols; ++k)
nperms[(i*ncols+j)*ncols+k] = perms[i*cols+k]>=j?
perms[i*cols+k]+1:perms[i*cols+k];
} /* endfor */
cols = ncols;
rows = nrows;
free(perms);
perms = nperms;
} /* endwhile */
}

return perms;
}

/* Generate a random permutation of {0,..,n-1} */
proc RandPerm(short int *p, int n)
{
int i, j, temp;

for(i = 0; i < n; i++)
p[i] = i;
for(i = n; i > 1; i--) {
j = rand()%i;
temp = p[i-1];
p[i-1] = p[j];
p[j] = temp;
}

return;
}
```

uservar.c The user contributed variables used in the Tabu Search.

```
#include "groupo.h"
#include "graph.h"
#include "stats.h"

#define BLOCK 10
#define MAXSEARCHES 10
#define percent(a,b) (((a)*(b))/100)

/* some local types */
typedef struct
{
  short int vertex;   /* vertex number in adjacency matrix    */
  short int chroma;   /* vertex chromaticity                  */
  short int ucdeg;    /* uncolored adjacent chromatic degree  */
```

```
  short int ucadj;    /* number of adjacent uncolored nodes    */
} sorttype;

/* output strings */
char *tsstr[] = { "Simple", "Probabilistic" };
char *eosstr[] = { "Absolute", "Stabilized" };
char *scstr[] = { "Random", "CLFOrder", "LF2Order" };
char *nbrstr[] = { "PairWiseSwaps", "Combinations",
   "VertexCycling", "ColorReductions",
   "VertexInsertion", "Shuffling", "FenceHopping" };
char *msstr[] = { "First", "Last", "Random" };
char *fenstr[] = { "FixedPosition", "RandomPosition",
                   "ForwardCycle", "BackwardCycle",
                   "PermutedCycle" };

/* function types */
typedef proc (*PROC)();
typedef move (*MFUNC)();
typedef bool (*BFUNC)();
typedef int (*IFUNC)();

/* global variables that must be set in Initialize() */
GROUP *ggp; /* global pointer to the output group  */
graph *gr;  /* global pointer to graph  */
colors *ko;  /* global pointer to colors  */
int search[MAXSEARCHES]; /* multiple search array       */
int nsearches;  /* number of searches to perform        */
int s_count;    /* number of searches performed         */
int sidx;       /* index of the current search          */
int rmax; /* maximum multistarts                  */
int r_count;    /* counter for restarts                 */
int tabustyle;  /* designates Simple(0) or Prob(1)      */
int srate;      /* neighborhood sample rate in percent */
int sstart;     /* initial sample rate                  */
bool sdyna;     /* use dynamic sampling                 */
bool sflag;     /* determines if dynamic sampling done  */
int sdelta;     /* change of sampling rate              */
int sstop;      /* ceiling of sampling rate             */
int iterations; /* the number of iterations to perform */
int i_count = 0;/* iteration counter for eos???()       */
int changup;    /* number of iteration to change search */
move *nbrhd;  /* holds all moves in neighborhood  */
int nsize;  /* neighborhood size  */
int interval; /* interval size for generating nbrhd   */
short int *i_temp; /* used in the MoveToCfg() function */
int istart;     /* initial interval size                */
bool idyna;     /* use dynamic interval sizes */
bool iflag;     /* determines if dynamic intervals done */
int idelta;     /* change in interval size              */
int istop;      /* ceiling of interval size             */
int bsize;    /* memory block size for ColorReductions*/
int ssize; /* neighborhood sample size */
```

```
short int *pi; /* permuted index for sampling nbrhd */
int sizeofcfg; /* used for the define SizeOfCfg  */
statistic istat;/* iteration statistics              */
int fence;      /* for use in FenceHopping           */
int short *pc;  /* permuted index for cycling        */
int (*CalcFence)(int fence); /* function pointer     */

/* some local functions */
bool eosSTA (void);
bool eosABS (void);
proc initSH (void);
proc initVI (void);
proc initCR (void);
proc initVC (void);
proc initCO (void);
proc initFH (void);
proc initPWS (void);
proc moveSH (config c, config d, move m);
proc moveCO (config c, config d, move m);
proc movePWS (config c, config d, move m);
proc genSH (config c);
proc genVI (config c);
proc genCR (config c);
proc genVC (config c);
proc genCO (config c);
proc genFH (config c);
proc genPWS (config c);
proc StartUp (int ss, bool sd, int is, bool id, int x);
proc startLF2 (config c);
proc startCLF (config c);
proc startRAN (config c);
move selectRAN(move *p, int n);
move selectLST(move *p, int n);
move selectFST(move *p, int n);
int fenceFP (int fence);
int fenceRP (int fence);
int fenceFC (int fence);
int fenceBC (int fence);
int fencePC (int fence);
int clfrule ARGS ((const void *a, const void *b));
int lf2rule (const void *a, const void *b);
word nextcolor ARGS ((graph g, int i));
proc CreateIndex (graph g, sorttype ** index,
  int (*rule)(const void *a, const void *b));
int ifact ARGS ((int i));
short int *AllPerms ARGS ((int i));
proc RandPerm ARGS ((short int *p, int n));
int firstpos(move m);

/* arrays of function pointers */
PROC  initfunc[] = {initPWS, initCO, initVC, initCR, initVI, initSH, initFH};
PROC  genfunc[] = {genPWS, genCO, genVC, genCR, genVI, genSH, genFH};
```

```
PROC  movefunc[] = {movePWS, moveCO, movePWS, moveSH, moveSH, moveSH, movePWS};
PROC  startfunc[] = {startRAN, startCLF, startLF2};
MFUNC selectfunc[] = {selectFST, selectLST, selectRAN};
BFUNC eosfunc[] = {eosABS, eosSTA};
IFUNC fenfunc[] = {fenceFP, fenceRP, fenceFC, fenceBC, fencePC};
```

vrcg.c A program to verify colorings of a graph.

```
/*  VRCG.C
 *
 *      DESCRIPTION:
 *      This file will verify that a given specification for coloring a
 *      composite graph will color the graph.  It assumes the file is
 *      specified according to the rules in "filedocs.txt".
 *
 *      $Log$
 */

static char rcsid[] = "$Id$";

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "misc.h"
#include "groupo.h"
#include "graph.h"

GROUP *gp;
graph g;
chroma c;
colors k;
int gid, group;

main()
{
    int row = 0;

    gp = gopen(stdout);
    while (InputDetails(stdin, &g, &c, &k, &row, &gid, &group)) {
        VerifyColors(g, gid, gp);
    }
    gclose(gp);
    return (0);
}
```

VITA

Jeffrey Wayne Jenness was born on February 21, 1960 in Joplin, Missouri. In 1965, his family moved to Neosho, Missouri where he received his elementary and secondary education. During the summer of his freshman year in high school his family moved back to Joplin where he completed his sophomore and junior years at Memorial High School. He graduated from Central Christian Academy his senior year in 1978.

Starting in the spring of 1981, he attended Missouri Southern State College while working as a field technician and computer programmer at White Industrial Seismology. During this time he met Theresa Bowling and was married December 1982. He graduated Cum Laude from Missouri Southern State College in May 1986 with a double major in Mathematics and Computer Science and a minor in Computing Analysis. In the fall of 1986 he enrolled in the Master's Degree program at the University of Missouri in Rolla and received a Master's Degree in Applied Mathematics in the summer of 1988. He enrolled in the Doctoral program in Computer Science in the fall of 1988.

In the fall of 1991, he accepted a position at Arkansas State University as Assistant Professor of Computer Science. He currently resides in Jonesboro, Arkansas with his wife and three children.