Missouri S&T Library and Learning Resources

Scholars' Mine

Doctoral Dissertations

Student Theses and Dissertations

Summer 1989

Graph coloring algorithms on random graphs

Shi-Jen Lin

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

Part of the Computer Sciences Commons

Department: Computer Science

Recommended Citation

Lin, Shi-Jen, "Graph coloring algorithms on random graphs" (1989). *Doctoral Dissertations*. 736.
https://scholarsmine.mst.edu/doctoral_dissertations/736

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. 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.

GRAPH COLORING ALGORITHMS ON RANDOM GRAPHS

BY

SHI-JEN LIN, 1955-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI - ROLLA

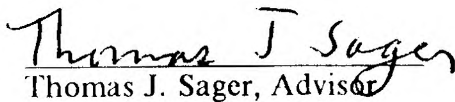
In Partial Fulfillment of the Requirements for the Degree

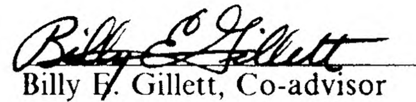
DOCTOR OF PHILOSOPHY

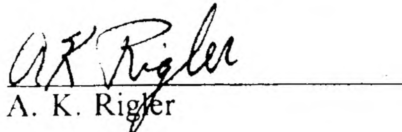
in

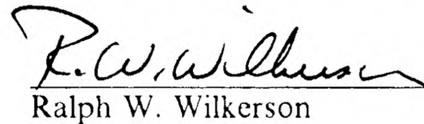
COMPUTER SCIENCE

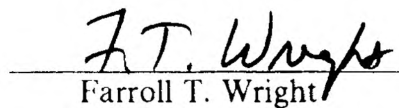
1989


Thomas J. Sager, Advisor


Billy E. Gillett, Co-advisor


A. K. Rigler


Ralph W. Wilkerson


Farroll T. Wright

ABSTRACT

The graph coloring problem, which is to color the vertices of a simple undirected graph with the minimum number of colors such that no adjacent vertices are assigned the same color, arises in a variety of scheduling problems. This dissertation focuses attention on vertex sequential coloring. Two basic approaches, backtracking and branch-and-bound, serve as a foundation for the developed algorithms. The various algorithms have been programmed and applied to random graphs. This dissertation will present several variatons of the Korman algorithm, Korw2, Pactual, and Pactmaxw2, which produce exact colorings quicker than the Korman algorithm in the average for some classes of graphs. In addition to exact algorithms, we also look at some heuristic algorithms, limit, epsilon, and branch-and-bound.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Thomas J. Sager, for his guidance and a great deal of moral encouragement in preparation of this work. I am also grateful for the assistance of the other members of my advisory committee: Dr. Billy E. Gillett, Dr. A. K. Rigler, and Dr. Farroll T. Wright.

I especially thank my parents for all their love and their financial assistance.

Finally, I would like to thank my wife, Tan Li, and my daughter, I-Shen, for their love and encouragement. Without them, this work would not have been completed.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF ILLUSTRATIONS vii

LIST OF TABLES x

I. INTRODUCTION 1

II. PRELIMINARIES 5

A. GRAPH THEORETIC TERMS 5

B. GRAPH COLORING TERMS 7

C. ASYMPTOTIC NOTATION 10

D. RANDOM GRAPH 11

E. BASIC PROPERTIES 12

III. LITERATURE REVIEW 16

A. EXACT ALGORITHMS 16

1. 0-1 Integer Programming Approach 16

2. Dichotomous Search Approach 18

3. Dynamic Programming Approach 20

4. Implicit Enumeration Approach 23

B. HEURISTIC ALGORITHMS 27

1. Vertex-color Sequential Algorithm 29

2. Color-vertex Sequential Algorithm 31

3. Vertex-vertex pair scanning 34

C. APPLICATIONS 36

1. Loading Problem 36

2. Timetable Scheduling 37

3. Resource Allocation 37

IV. VARIATIONS ON THE IMPLICIT ENUMERATION APPROACH 38

A. BACKGROUND 38

1. Terminology of Trees 38

2. Tree Search Techniques 40

a. Depth-first Search (DFS) 40

b. Breadth-first Search (BFS) 41

B. BACKTRACKING 42

C. BRANCH-AND-BOUND 47

D. PROGRAMMING FRAME OF BACKTRACKING ON THE GCP 50

E. PROGRAMMING FRAME OF BRANCH-AND-BOUND ON THE GCP 54

V. IMPLEMENTATION NOTES 59

A. GRAPHS REPRESENTATIONS 59

B. MEMORY SWAPPING 62

C. HEAPSORT 64

D. RANDOM NUMBER GENERATOR 73

VI. SELECTION FUNCTIONS 77

A. TERMS 77

B. NEXT-UNCOLORED-VERTEX SELECTION FUNCTIONS . 78

C. NEXT-COLORED-VERTEX SELECTION FUNCTIONS 80

D. WEIGHTED SCALE ON THE SCORE OF THE NEW COLOR 83

E. SWAPPING BETWEEN THE CORE AND THE PERIPHERY 83

F. HEURISTIC ALGORITHMS 88

1. Limit 88

2. Epsilon 88

G. COMPUTATIONAL RESULTS 89

H. DISCUSSION 93

VII. A BRANCH-AND-BOUND PREFERENCE FUNCTION 190

A. CONSTRUCTION OF A PREFERENCE FUNCTION 190

B. COMPUTATIONAL RESULTS 194

C. DISCUSSION 196

VIII. CONCLUSIONS 205

A. SUMMARY 205

B. FURTHER RESEARCH TOPICS 208

REFERENCES 209

VITA 213

LIST OF ILLUSTRATIONS

Figure		Page
1	Example of graphs	6
2	A graph and its subgraphs	7
3	Example of a graph	9
4	Mycielski's graphs	13
5	Zykov's Algorithm.	20
6	Example of Zykov-tree	21
7	Christofides's Algorithm.	22
8	Implicit Enumeration Approach.	25
9	Brown's Algorithm.	28
10	Approximately Maximum Independent Set	32
11	Leighton's Algorithm	33
12	Wood's Algorithm.	35
13	Dutton-Brigham's Algorithm.	35
14	Example of tree TS	39
15	General Backtracking Algorithm	44
16	Tree representation of the 4-queens problem	45
17	Backtrack-tree of 4-queens problem	46
18	General Branch-and-bound Algorithm	48
19	Examples of BNB-tree	49
20	Intelligent Branch-and-bound Algorithm	50
21	Procedures newcolor and mergecolor	51
22	Backtracking scheme on the GCP	52
23	Procedure node_to_process in backtracking and branch-and-bound	53
24	Procedure forward in backtracking	53

25	Procedure backward in backtracking	54
26	Branch-and-bound scheme on the GCP	56
27	Procedure getnextstate in branch-and-bound	56
28	Procedure savestatus in branch-and-bound	57
29	Procedure forward in branch-and-bound	57
30	Procedure backward in branch-and-bound	58
31	Representations of a graph	60
32	Memory swapping in the backtracking scheme	63
33	Procedure node_to_process of memory swapping	64
34	Procedure backward of memory swapping	65
35	Example of full tree	67
36	Procedure heapup	67
37	Procedure heapdown	68
38	Procedure heapsort	68
39	Part 1 of the example 5-2	69
40	Part 2 of the example 5-2	70
41	Part 3 of the example 5-2	71
42	Part 4 of the example 5-2	72
43	A multiplicative congruential generator	76
44	heuristic colors for edgeload = 0.3	153
45	heuristic colors for edgeload = 0.5	154
46	heuristic colors for edgeload = 0.7	155
47	heuristic colors for edgeload = 0.9	156
48	running time for edgeload = 0.3	157
49	running time for edgeload = 0.5	158
50	running time for edgeload = 0.7	159
51	running time for edgeload = 0.9	160

52 forward moves for edgeload = 0.3 161

53 forward moves for edgeload = 0.5 162

54 forward moves for edgeload = 0.7 163

55 forward moves for edgeload = 0.9 164

56 forward moves on log scale for edgeload = 0.3 165

57 forward moves on log scale for edgeload = 0.5 166

58 forward moves on log scale for edgeload = 0.7 167

59 forward moves on log scale for edgeload = 0.9 168

60 exact colors 169

61 part 1 of the preference function 193

62 part 2 of the preference function 194

63 part 3 of the preference function 195

LIST OF TABLES

Table		Page
I	VERTICES = 40, EDGELOAD = 0.3	95
II	VERTICES = 40, EDGELOAD = 0.5	96
III	VERTICES = 40, EDGELOAD = 0.7	97
IV	VERTICES = 40, EDGELOAD = 0.9	98
V	VERTICES = 40, EDGELOAD = 0.3	99
VI	VERTICES = 40, EDGELOAD = 0.5	100
VII	VERTICES = 40, EDGELOAD = 0.7	101
VIII	VERTICES = 40, EDGELOAD = 0.9	102
IX	VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2	103
X	VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2	104
XI	VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2	105
XII	VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2	106
XIII	VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2	107
XIV	VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2	108
XV	VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2	109
XVI	VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2	110
XVII	VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2	111
XVIII	VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2	112
XIX	VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2	113
XX	VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2	114
XXI	VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2	115
XXII	VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2	116
XXIII	VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2	117
XXIV	VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2	118

XXV VERTICES = 40, EDGELOAD = 0.3 119

XXVI VERTICES = 40, EDGELOAD = 0.5 120

XXVII VERTICES = 40, EDGELOAD = 0.7 121

XXVIII VERTICES = 40, EDGELOAD = 0.9 122

XXIX VERTICES = 40, EDGELOAD = 0.3 AND 0.5 123

XXX VERTICES = 40, EDGELOAD = 0.7 AND 0.9 124

XXXI VERTICES = 44, EDGELOAD = 0.3 AND 0.5 125

XXXII VERTICES = 44, EDGELOAD = 0.7 AND 0.9 126

XXXIII VERTICES = 40, EDGELOAD = 0.3 127

XXXIV VERTICES = 40, EDGELOAD = 0.5 128

XXXV VERTICES = 40, EDGELOAD = 0.7 129

XXXVI VERTICES = 40, EDGELOAD = 0.9 130

XXXVII VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2 131

XXXVIII VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2 132

XXXIX VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2 133

XL VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2 134

XLI VERTICES = 44, EDGELOAD = 0.3 AND 0.5, WEIGHT = 2 135

XLII VERTICES = 44, EDGELOAD = 0.7 AND 0.9, WEIGHT = 2 136

XLIII VERTICES = 48, EDGELOAD = 0.3, WEIGHT = 2 137

XLIV VERTICES = 48, EDGELOAD = 0.5, WEIGHT = 2 138

XLV VERTICES = 48, EDGELOAD = 0.7, WEIGHT = 2 139

XLVI VERTICES = 48, EDGELOAD = 0.9, WEIGHT = 2 140

XLVII VERTICES = 52, EDGELOAD = 0.3 AND 0.5, WEIGHT = 2 141

XLVIII VERTICES = 52, EDGELOAD = 0.7 AND 0.9, WEIGHT = 2 142

XLIX VERTICES = 56, EDGELOAD = 0.3 AND 0.5, WEIGHT = 2 143

L VERTICES = 56, EDGELOAD = 0.7 AND 0.9, WEIGHT = 2 144

LI	COMPARISON KORW2 WITH KORMAN IN RUNNING TIME	145
LII	COMPARISON KORW2 WITH KORMAN IN RUNNING TIME	146
LIII	COMPARISON PACTUAL WITH KORMAN IN RUNNING TIME	147
LIV	COMPARISON PACTUAL WITH KORMAN IN RUNNING TIME	148
LV	COMPARISON PACTMAXW2 WITH KORMAN IN RUNNING TIME	149
LVI	COMPARISON PACTMAXW2 WITH KORMAN IN RUNNING TIME	150
LVII	COMPARISON PACTMAXW2 WITH PACTUAL IN RUNNING TIME	151
LVIII	COMPARISON PACTMAXW2 WITH PACTUAL IN RUNNING TIME	152
LIX	VERTICES = 40, EDGELOAD = 0.3	170
LX	VERTICES = 40, EDGELOAD = 0.5	171
LXI	VERTICES = 40, EDGELOAD = 0.7	172
LXII	VERTICES = 40, EDGELOAD = 0.9	173
LXIII	VERTICES = 40, EDGELOAD = 0.3	174
LXIV	VERTICES = 40, EDGELOAD = 0.5	175
LXV	VERTICES = 40, EDGELOAD = 0.7	176
LXVI	VERTICES = 40, EDGELOAD = 0.9	177
LXVII	VERTICES = 40, EDGELOAD = 0.3	178
LXVIII	VERTICES = 40, EDGELOAD = 0.5	179
LXIX	VERTICES = 40, EDGELOAD = 0.7	180
LXX	VERTICES = 40, EDGELOAD = 0.9	181
LXXI	VERTICES = 40, EDGELOAD = 0.3	182
LXXII	VERTICES = 40, EDGELOAD = 0.5	183
LXXIII	VERTICES = 40, EDGELOAD = 0.7	184

LXXIV VERTICES = 40, EDGELOAD = 0.9 185
LXXV VERTICES = 40, EDGELOAD = 0.3 186
LXXVI VERTICES = 40, EDGELOAD = 0.5 187
LXXVII VERTICES = 40, EDGELOAD = 0.7 188
LXXVIII VERTICES = 40, EDGELOAD = 0.9 189
LXXIX VERTICES = 30, EDGELOAD = 0.3 197
LXXX VERTICES = 30, EDGELOAD = 0.5 198
LXXXI VERTICES = 30, EDGELOAD = 0.7 199
LXXXII VERTICES = 30, EDGELOAD = 0.9 200
LXXXIII VERTICES = 40, EDGELOAD = 0.3 201
LXXXIV VERTICES = 40, EDGELOAD = 0.5 202
LXXXV VERTICES = 40, EDGELOAD = 0.7 203
LXXXVI VERTICES = 40, EDGELOAD = 0.9 204

I. INTRODUCTION

The graph coloring problem (GCP), which is to color the vertices of a simple undirected graph with the minimum number of colors such that no adjacent vertices are assigned the same color, arises in a variety of problems of optimal partitioning of mutually exclusive events or objects. For instance, the timetable scheduling problem is to find the schedule of persons such that no two persons are using the same hardware tool in the same period of time, and all jobs are executed with the minimum process time. Christofides employed graph coloring to solve the resource allocation problem [*Ch*75], and Elion and Christofides used it to solve the loading problem [*EC*71]. The partitioning of mutually exclusive objects is usually embodied through a graph in such a way that each object is represented by a node, and every pair of exclusive objects is connected by an edge. Thus, optimal solutions to such problems can be found by applying a graph coloring algorithm to the corresponding graph.

Even though the GCP has received considerable attention because of its various applications, there is no known algorithm which will optimally color *every* graph in a polynomial time function in the number of vertices of the given graph. Moreover, the GCP has been shown to be $\mathbb{NP}$-complete [*AH*74] [*GJ*79] ; that is, it seems unlikely that any such algorithm can be found. Therefore, it is necessary to seek a heuristic algorithm with a polynomial time bound. For some $\mathbb{NP}$-complete problems such as the bin packing problem, there is a *nlogn* heuristic algorithm obtaining the solution within a small multiple of the optimal solution (actually solution ≤ 1.7 * (optimal solution) + 2) [*HS*78]. Unfortunately, many known heuristic algorithms have extremely bad worst case behavior [*Jo*74]: the ratio $A(G)/\chi(G)$, where $A(G)$ denotes the number of colors used by algorithm A to color a graph G and $\chi(G)$ denotes the minimal number of colors with which G can be colored, can be shown to grow linearly with the number of vertices in G. Furthermore, Garey and Johnson showed that even

finding an efficient algorithm which colors every graph G in $r * \chi(G) + d$ colors, where $r < 2$, remains a $\mathbb{NP}$-complete problem [*GJ*76].

Johnson showed that many popular heuristic algorithms will color a graph by using the number of colors which is linear to the order of the graph, but how often does the worst cases appear? Grimmett and McDiarmid showed that the vertex-color sequential algorithm, VS, (refer to chapter 3) has the property, for *almost all* graphs G_n, $VS(G_n)/\chi(G_n) \leq 2 + \varepsilon$ as $n \to \infty$ [*GM*75]. Therefore, we know that the average behavior of some heuristic algorithms such as the vertex-color sequential algorithm is much better than their worst behavior. McDiarmid [*Mc*79] showed that all variations of the vertex-color sequential algorithm have the aforementioned property. These variations act as follows: whenever a new color is introduced, a checking and recoloring procedure is called in order to find a color, which has already been used, for the current vertex. This procedure then recolors some of the previously colored vertices. The goal is to find a fast and accurate algorithm that includes the checking and recoloring technique.

The *chrome-degree* of vertex v is the number of colors used so far on vertices adjacent to v. The Korman algorithm without backtracking (refer to chapter 6) sequentially picks the vertex with the highest chrome-degree and colors it. Manvel proposed that the Korman algorithm without backtracking is superior to the vertex-color sequential algorithm and the vertex-color sequential algorithm with pre-ordering by the largest-first principle [*Ma*81]. In terms of running speed, Korman presented experimental results showing that the Korman algorithm is also superior to the vertex-color sequential algorithm and the LF vertex-color sequential algorithm in the backtracking scheme [*Ko*79]. This dissertation will present several variations of the vertex-color sequential algorithm which perform better than the Korman algorithm in the mean for some graphs. These algorithms, Korw, Pactual, and

Pactmaxw2, will be described in chapter 6. In addition to exact algorithms, we will look at some variations of the Korman algorithm, *limit*, *epsilon*, and *branch-and-bound*, which produce heuristic colorings.

Chapter 2 introduces basic graph-theoretic terms and graph coloring related terms. It also presents some well-known properties on graph colorability.

Chapter 3 reviews four different exact approaches: 0-1 integer programming, dichotomous search, dynamic programming, and implicit enumeration, three different heuristic algorithms: a vertex-color sequential algorithm, a color-vertex sequential algorithm, and a vertex-vertex pair scanning; and finally three basic applications: a loading problem, timetable scheduling and resource allocation.

In chapter 4, two distinct programming schemes, backtracking and branch-and-bound, serve as a foundation for the developed algorithms. The profiles for both programming schemes are presented.

Chapter 5 describes some implementation notes on data structures, memory swapping, and a random number generator.

Chapter 6 proposes a number of next-uncolored-vertex selection (nucv-selection) functions and next-color-vertex selection (ncv-selection) functions, which are the heart of the developed algorithms. The experimental results of the backtracking scheme with various combinations of nucv-selection and ncv-selection functions are also included. Finally two heuristic algorithms which are based on the backtracking scheme are presented.

Chapter 7 introduces the procedure of constructing a preference function of the branch-and-bound scheme. The computational results of various algorithms are included.

Chapter 8 deals with conclusions and further research directions.

II. PRELIMINARIES

A. GRAPH THEORETIC TERMS

A *graph* G = (V, E) is a finite set of vertices V, together with a set of undirected or directed edges E. An *undirected graph* is a graph with undirected edges, and a *directed graph* is a graph with directed edges. The elements of E are unordered (ordered) pairs of vertices. The number of vertices in a graph is called the *order* of the graph, denoted by |V|.

For the edge e = <x, y>, vertex x is called its *initial endpoint*, and vertex y is called its *terminal endpoint*. Of course, there is no difference between initial endpoint and terminal endpoint in an undirected graph. A *loop* is an edge whose initial endpoint and terminal endpoint are the same.

An undirected graph (or multigraph) is called a *simple graph* if it has (1) no loops; (2) no more than one edge joining any two vertices.

Example 2.1. In Figure 1-1, V = {a, b, c}, E = {<b, a>, <c, a>}.
In Figure 1-2, V = {a, b, c}, E = {<a, a>, <a, b>, <a, c>, <a, c>}. <a, a> is a loop, and there are two edges between a and c. In this case, E is a multiset. In Figure 1-3, V = {a, b, c}, E = {<a, b>, <a, c>}.

Hereafter we refer to an undirected graph as a graph. Two edges are called *adjacent* if they have at least one endpoint in common. The *degree* of the vertex v is the number of edges with v as endpoint. The degree of v is denoted by $d_G(v)$. Note that each loop is counted twice. A graph is *k-regular* if each vertex has the same degree k.

A graph K_n = (V, E) is said to be *complete* if every pair of distinct vertices has an associated edge and $|V| = n$. A graph is *k-partite* if its vertices can be partitioned into k subsets such that no two vertices in the same subset are adjacent. For k = 2,

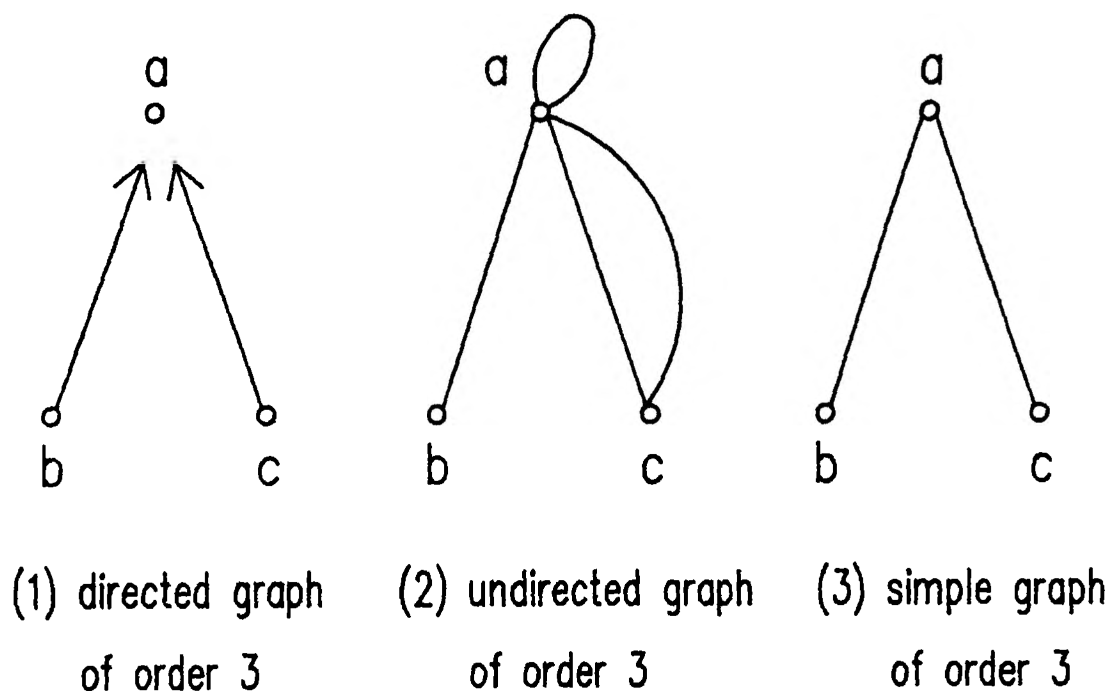


Figure 1. Example of graphs

such graphs are called *bipartite*. A *null graph* is a graph with no edges. An *empty graph* is a graph which has no vertices and no edge.

A *path* is a sequence of vertices such that every pair of consecutive vertices is an edge. A path is *simple* if all vertices on the path are distinct with exception that the initial endpoint and the terminal endpoint may be the same. The *length* of a path is the number of edges along the path. A *cycle* is a simple path that connects a vertex to itself.

A *connected graph* is a graph that contains a path for every pair of distinct vertices. A graph is *cyclic* if it contains at least one cycle. A connected acyclic graph is sometimes called a *tree*. A *cycle graph* of order n, denoted by C_n, is a connected graph whose edges form a cycle of length n.

A subgraph of G = (V, E) is a graph G′ = (V′, E′) such that (1) V′ is a subset of V; and (2) E′ consists of edges <x, y> in E such that x as well as y are in V′. G′ is an *induced subgraph* of G if G′ contains all edges in E such that both endpoints of the edge are in V′.

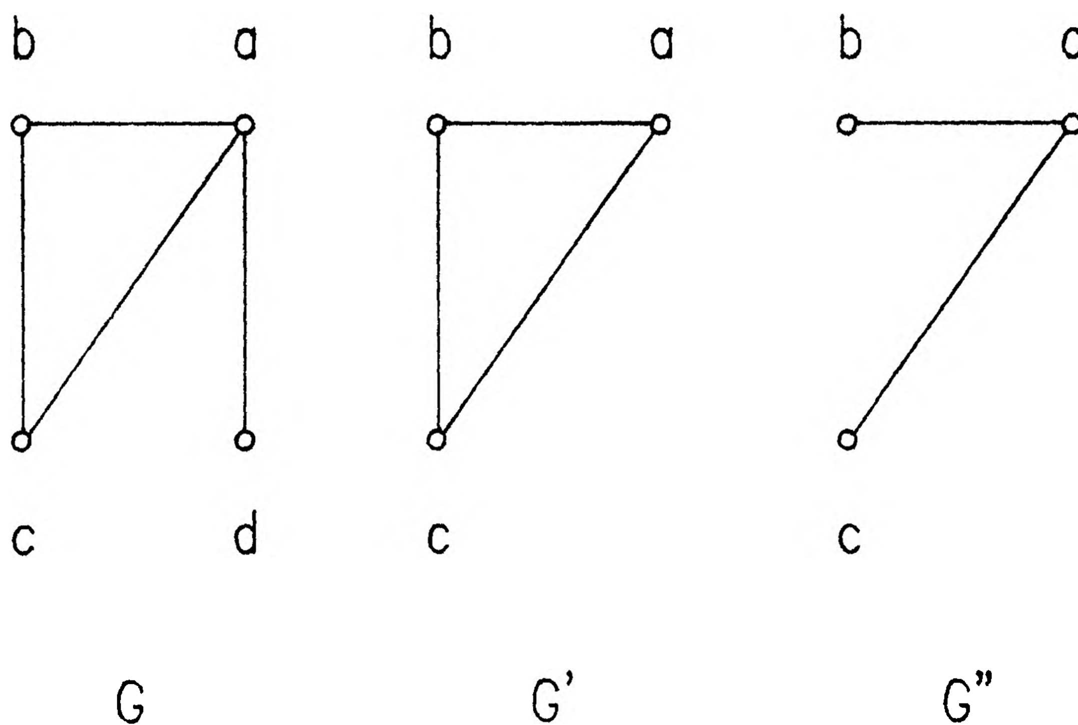


Figure 2. A graph and its subgraphs

Example 2.2. In Figure 2, both G′ and G″ are subgraphs of G. G′ is an induced subgraph of G. However, G″ is not an induced subgraph of G because it does not contain the edge <b, c>.

B. GRAPH COLORING TERMS

An assignment of color to each vertex of a graph G such that no adjacent vertices have been assigned the same color is called a *coloring* of G. We refer to this coloring

as a *complete coloring*, in contrast to a *partial coloring*, which is an assignment of colors to part of the vertices of G.

f is a *k-coloring function* of a graph G = (V, E) if f is an onto function from V to the set $\{1, 2, \ldots, k\}$ such that f(v) = j if and only if vertex v is colored with color j. G is said to be *k-colorable* if G has a k-coloring function. The minimum k for which G is k-colorable is called the *chromatic number* of G and is denoted by $\chi(G)$.

An *independent set* of a graph G = (V, E) is a subset of V, no two vertices of which are adjacent. An independent set is a *maximal independent set*, denoted by MIS, if it is not a proper subset of an independent set. In graph coloring, each subset of vertices of the same color is called a *color class*. From the definition of the independent set, a color class is an independent set. The *independence number* of G, denoted $\alpha(G)$, is the number of vertices of the largest MIS in G.

In contrast, a *completely connected set* of a graph G = (V, E) is a subset of V, every two vertices of which are adjacent. A maximal completely connected set is called a *clique*. It is seen that a complete graph is a clique of itself. We know that a MIS of a graph G is a clique of $\overline{G}$, its complement graph, and vice versa. The *clique number* of G, denoted $\omega(G)$, is the number of vertices of the largest clique in G.

Example 2.3 In Figure 3, G = (V, E), V = {a, b, c, d, e}, $E = \{\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle, \langle a, e\rangle, \langle c, e\rangle\}$. We can define a 3-coloring function f as follow:

f(a) = 1, f(b) = 2, f(c) = 2,

f(d) = 2, f(e) = 3.

We say G is k-colorable for all $k \geq 3$.

The following sets are all completely connected sets of G:

{a}, {b}, {c}, {d}, {e},

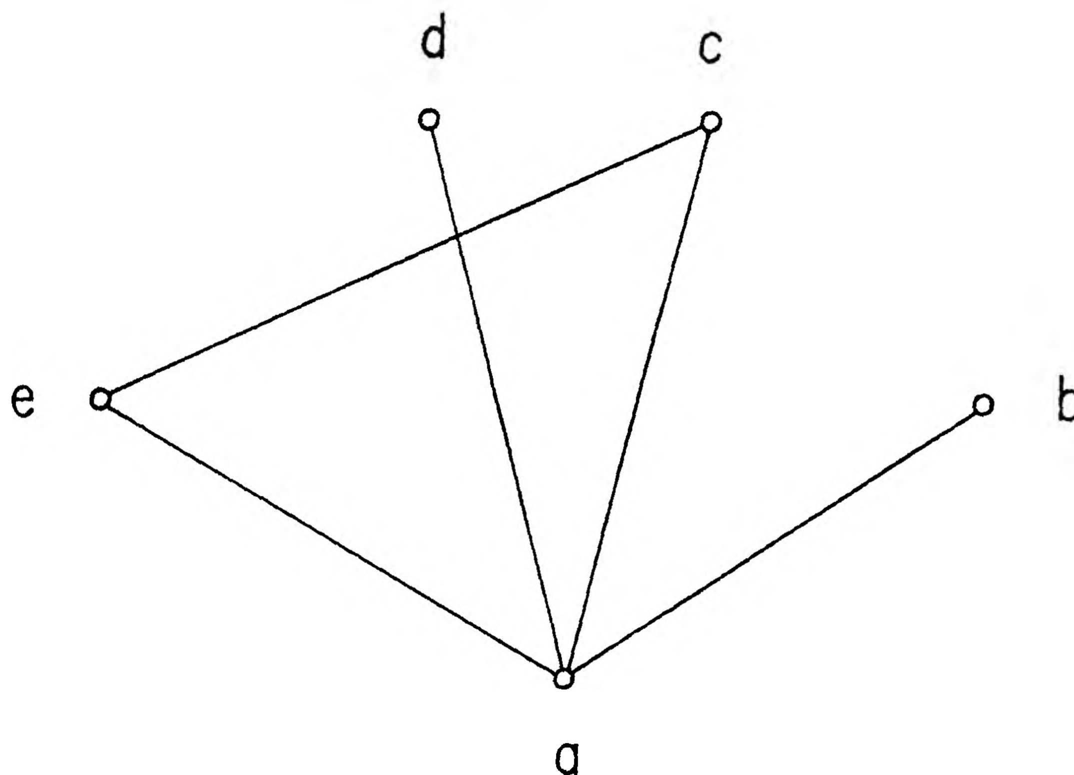


Figure 3. Example of a graph

{a, b}, {a, c}, {a, d}, {a, e}, {c, e},

{a, c, e}.

{a, b}, {a, d}, and {a, c, e} are clique. $\omega(G) = 3$.

ℙ is the class of problems solvable by deterministic Turing machines which operate in polynomial time. ℕℙ is the class of problems solvable by non-deterministic Turing machines which operate in polynomial time. A problem A is *reducible to* a problem B if there is a polynomial-time bounded function f from A to B such that for every instance x, $f(x) \in B$ if and only if $x \in A$. A problem L is ℕℙ-*complete* if (1) $L \in$ ℕℙ ; and (2) every problem in ℕℙ is reducible to L. A problem L is ℕℙ-*hard* if every problem in ℕℙ is reducible to L. As we see all ℕℙ-complete problems are ℕℙ-hard but all ℕℙ-hard problems are not ℕℙ-complete. Garey and Johnson offered a complete picture of ℕℙ-completeness in their publication [*GJ*79].

Some examples of classes of problems that are in ℕ or ℕℙ are as follow:

Minimum Spanning Tree ∈ ℙ [*Kr*56]

Given a connected graph G = (V, E), a cost function C on E, and an integer m. Is there a spanning tree of G with cost m or less?

Shortest Path ∈ ℙ [*Di*59]

Given a graph G = (V, E), a weight function on E, source and destination in V, and an integer w. Is there a path between source and destination of G with weight w or less?

Clique ∈ ℕℙ-complete [*AH*74] [*PS*82]

Does an undirected graph have a clique of order k?

Satisfiability ∈ ℕℙ-complete [*Co*71] [*AH*74] [*PS*82]

Is a boolean expression satisfiable?

Colorability ∈ ℕℙ-complete [*AH*74]

Is an undirected graph k-colorable for $k \geq 3$?

Context-sensitive Recognition ∈ ℕℙ-hard [*Ka*72]

Given a context-sensitive grammar C and a string s. Is s in the language generated by C? Note that this problem is not known to be ℕℙ-complete because its membership in ℕℙ is presently in doubt.

An *exact algorithm* can always produce an optimal solution. A *heuristic algorithm* will sometimes find an optimal solution. In general, a heuristic algorithm is trying to find a *good* solution within an acceptable amount of time.

C. ASYMPTOTIC NOTATION

Here we introduce two mathematical notations which will be used for the analysis of the complexity.

Definition $f(n) = O(g(n))$ if and only if there exist two positive integers c and n_0 such that $|f(n)| \leq c * |g(n)|$, for all $n \geq n_0$.

O-notation can be used to express the upper bound of the performance of an algorithm. It ignores the constant term of the leading term (as $g(n)$ in the definition). However, sometimes we need a more precise description, which also takes into account the constant term, of the algorithm's performance. A stronger mathematical notation is given below.

Definition $f(n) = o(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

Example 2.4 Let $f(n) = n^2 + n$. $f(n) = O(n^2)$ by picking $c = 2$, and $n_0 = 1$, but $f(n) = o(n^2)$. Consider $g(n) = 2n^2$. $g(n) = O(n^2)$ by picking $c = 2$, $n_0 = 1$. However $g(n) \neq o(n^2)$ since $\lim_{n\to\infty} \frac{g(n)}{n^2} = 2$. But $g(n) = O(n^2)$.

D. RANDOM GRAPH

A *random graph* , $G_{n,P(n)}$, has n vertices. Its edges are *independently* chosen from $\binom{n}{2}$ possible edges by the probability function P(n), where $0 \leq P(n) \leq 1$. We include the edge with probability P(n). It is seen that $G_{n,1}$ is a complete graph, and $G_{n,0}$ is a null graph. Suppose that $P(n) = \lambda$ is a constant. λ is called the *edgeload* of $G_{n,\lambda}$. The number of edges of $G_{n,\lambda}$ is *binomially distributed* with parameters λ and $\binom{n}{2}$, and thus has mean $\lambda\binom{n}{2}$ and variance $\lambda(1 - \lambda)\binom{n}{2}$.

For programming, each potential edge is obtained by getting a uniformly distributed pseudo random number between 0 and 1 from the pseudo random number generator; if the obtained pseudo random number is less than the desirable edgeload, the edge is included. The experimental results presented later are obtained by giving the edgeload λ and the total number of edges $\lambda\binom{n}{2}$ chosen randomly from the $\binom{n}{2}$ possible edges.

E. BASIC PROPERTIES

For the clique G′ of order m, since all vertices are adjacent, the chromatic number of G′ is m. It is seen that the clique number of a graph G is an lower bound of $\chi(G)$. However, this lower bound can be very poor. Consider the following construction of Mycielski [$My55$] [$SD83$]. Initially, $M_1 = K_1$, $M_2 = K_2$ and for $k \geq 2$, constructing M_{k+1} from M_k. Suppose $|M_k| = m$, $v_1, \dots, v_m$ are the vertices of M_k. Let $|M_{k+1}|$ = 2m + 1 and have vertices $v_1, \dots, v_m$, $v'_1, \dots, v'_m$, w_{k+1}, The edge set of M_{k+1} consists of (1) all edges of M_k, (2) all edges between v'_i and the neighbors of v_i, i = $1, \dots, m$, and (3) $< w_{k+1}, v'_i >$, $i = 1, \dots, m$. Figure 4 shows the construction of M_3 from M_2, and M_4 from M_3. In the sequence M_1, M_2, ..., $\chi(M_i) = i$ but $\omega(M_i) = 2$. Therefore, the lower bound $\omega(M_i)$ for $\chi(M_i)$ can be very bad as i becomes very large.

We now present some well-known theorems on graph colorability.

Theorem 2.1 If G is a graph of order n, then

$$\chi(G) \geq \frac{n}{\alpha(G)}$$

$$\chi(G) \leq n + 1 - \alpha(G).$$

(*See* [*Be*73]).

Theorem 2.2 If G is a simple graph with n vertices and m edges, then

$$\chi(G) \geq \frac{n^2}{n^2 - 2m}.$$

(*See* [*Be*73]).

Theorem 2.3 G = (V, E) is a simple graph, then

$$\chi(G) \leq 1 + \max\{d_G(v)\}, \qquad \text{for } v \in V.$$

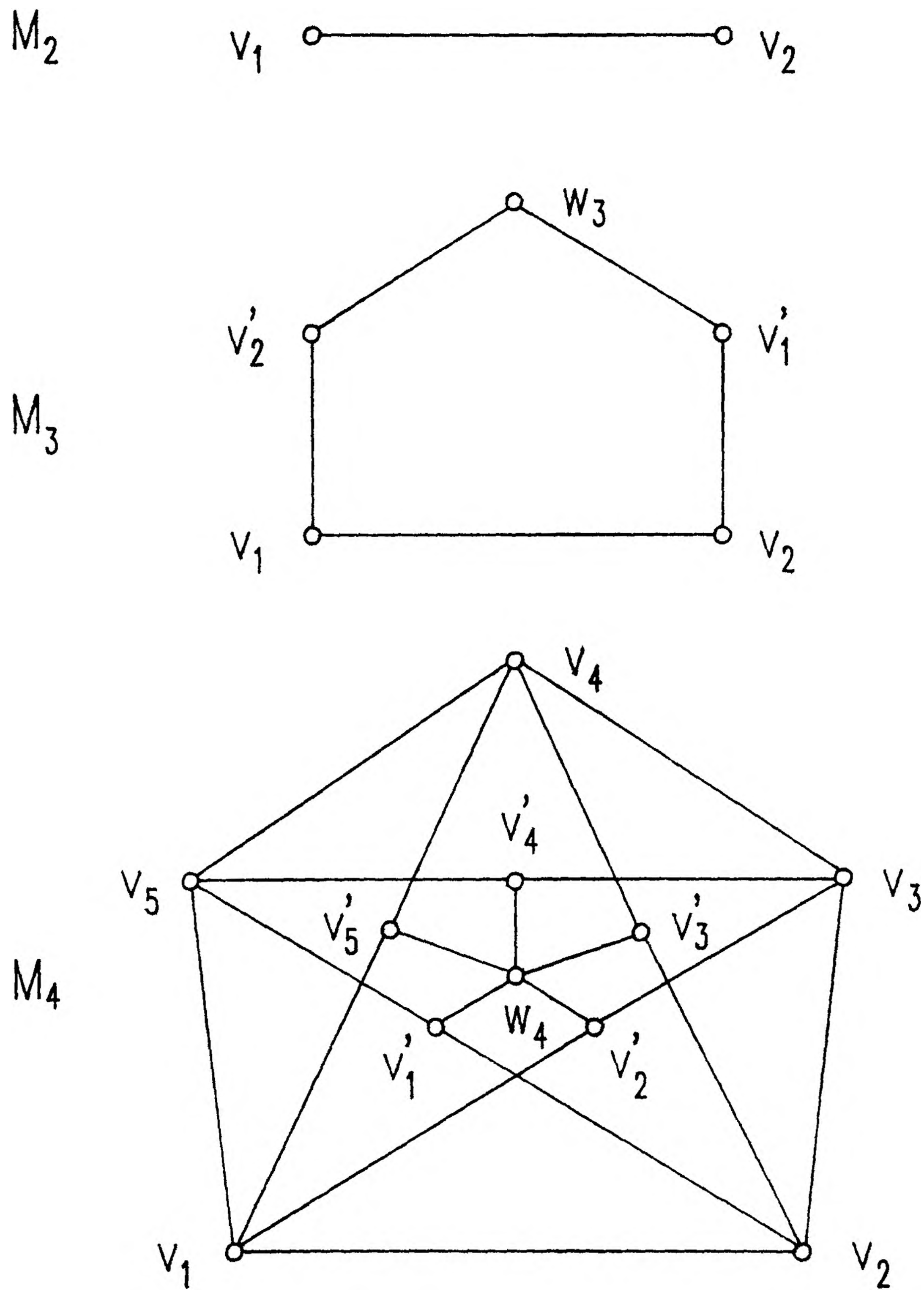


Figure 4. Mycielski's graphs

(*See* [*Br*41], [*Be*73]).

There are only two kinds of graphs for which Theorem 2.3 holds with equality above: odd cycle graphs and complete graphs. This result is known as Brook's Theorem

Theorem 2.4 Any planar graph is 4-colorable.

(*See* [*AH*77*a*], [*AH*77*b*], [*Ha*77], [*Be*77]).

A *map coloring* is to find an assignment of a color to each region so that all contiguous regions have different colors. For a map M in the plane we can construct a dual graph G whose vertices are in one-to-one correspondence with the regions of M and whose edges are in one-to-one correspondence with the border lines between regions. Thus, the map coloring is equivalent to the *planar graph coloring*. The conjecture, any planar graph is 4-colorable, had been a well-known unsolved problem in mathematics for over a century. Kempe [*Ke*79] is the first person known to have attacked the 4-coloring conjecture. Although Kempe's work contained a flaw which Heawood [*He*90] pointed out, it contained a valuable contribution which became the basis of many later attempts to prove the conjecture. In 1972, Appel, Haken, and Koch took three and a half years to develop the method based on Kempe's works to solve the 4-coloring conjecture and another 6 months to verify the 4-color reducibility of nearly 1900 cases. The verification part was done by computer.

Theorem 2.5 The coloring problem is $\mathbb{NP}$-complete.

(*See* [*AH*74]).

Theorem 2.6 Consider the graph coloring of a graph G. If for some constant r < 2 and constant d there exists a polynomial time bounded algorithm A such that $A(G) \leq r * \chi(G) + d$, where A(G) denotes the number of colors when A is applied to G, then there is a polynomial time bounded exact algorithm B.

(*See* [*GJ*76]).

Theorem 2.6 states that even finding an efficient near-optimal graph coloring algorithm is as hard as discovering an efficient exact algorithm.

III. LITERATURE REVIEW

The graph coloring problem has received considerable attention for many years. This chapter reviews exact as well as heuristic algorithms known earlier, and describes a number of applications of the graph coloring problem.

A. EXACT ALGORITHMS

In this section we describe four classic approachs which are exact in the sense that they guarantee the correct value of the chromatic number for any arbitrary graph.

1. 0-1 Integer Programming Approach.

The GCP can be formulated as a number of 0-1 integer programming problems [*Ch*75] based on different kinds of variables. Unfortunately, not every formulated 0-1 integer programming problem can be solved efficiently because of the huge size of both variable set and constraints. Let us consider the following two models. Let the graph G = (V, E) be of order n, $[a_{ij}]_{n \times n}$ be the *adjacency matrix* of G with all diagonal elements set to zero, and $[c_{ij}]_{n \times q}$ be a *coloring matrix* of G such that

$$c_{ij} = 1 \quad \text{if vertex } v_i \text{ is assigned color } j\,; \qquad = 0 \quad \text{otherwise,}$$

where q is an upper bound of $\chi(G)$. The GCP can be rewritten as

$$\text{Minimize} \quad z = \sum_{j=1}^{q} \sum_{i=1}^{n} w_j c_{ij} \tag{3.1}$$

subject to

$$\sum_{j=1}^{q} c_{ij} = 1, \quad \text{for all } i = 1, \dots, n \tag{3.2}$$

$$n * c_{ij} + \sum_{k=1}^{n} a_{ik} c_{kj} \le n \tag{3.3}$$

for all $i = 1, \dots, n$, and $j = 1, \dots, q$.

In the object function (3.1), w_j is a *weight* of the color j; the weight satisfies $w_{j+1} > n * w_j$. Constraint (3.2) ensures that any vertex can be colored with one and only one color. Condition (3.3) simulates the requirement, every pair of adjacent vertices cannot have the same color. If vertex v_i is colored with color j, then the first term is n. Thus the second term must be zero; if $a_{ik} = 1$ then $c_{kj} = 0$. In other words, any adjacent vertex v_i is not colored with color j, the second term of (3.3) can go up to $\deg(v_i)$, which is less than n. So the condition (3.3) is satisfied. Note that if x and y are both adjacent to v_i and also adjacent to each other, condition (3.3) of v_i can not avoid coloring x and y with the same color. However, conditon (3.3) of x (or y) can prevent x and y from being assigned the same color.

Alternatively, let $M_1, M_2, \ldots, M_p$ be all the MIS's of G and define the *inclusion* matrix $[m_{ij}]_{n \times p}$ such that

$$m_{ij} \;\; = 1 \quad \text{if vertex } v_i \in M_j\,; \qquad = 0 \quad \text{otherwise.}$$

We also define the *cost* variable c_j associated with M_j such that

$$c_j \;\; = 1 \quad \text{if } M_j \text{ includes a color class of the optimal coloring;} \qquad = 0 \quad \text{otherwise.}$$

We have the following 0-1 integer programming problem which is equivalent to the GCP:

Minimize $\quad z = \sum_{j=1}^{p} c_j$

subject to

$$\sum_{j=1}^{p} m_{ij} c_j \geq 1 \qquad \text{for all } i = 1, \ldots, n.$$

Note that the inequalities mean the *over-coloring* possibility. If the over-coloring occurs, we arbitrarily pick one from the feasible color set.

The former programming model has nq variables in the coloring matrix and $n + nq$ constraints; the latter has $np + p$ variables (including matrix plus cost variables) and n constraints. Both models require pre-processing work. The upper bound of the first

model can be computed by a heuristic coloring algorithm or simply set to $|V|$, and the MIS enumeration of the second one can be solved approximately by an efficient algorithm proposed by Bron and Kerbosch [*BK*73]. In terms of the complexity, the latter model is much better than the former.

Example 3.1 Consider Figure 3 again. Let $M_1 = \{a\}$, $M_2 = \{b, c, d\}$, and $M_3 = \{b, d, e\}$ be all MIS's of G.

The inclusion matrix is as follow:

M_{ij}	M_1	M_2	M_3
a	1		
b		1	1
c		1	
d		1	1
e			1

The 0's in the inclusion matrix are represented by blanks. The GCP is to find the minimum number of columns which cover all rows. The optimal solution is $c_1 = 1$, $c_2 = 1$, and $c_3 = 1$, where b and d are over-colored.

2. Dichotomous Search Approach.

This method was proposed by Zykov in 1952 [*Zy*52]. The basic idea of this approach is to reduce a graph to a complete graph by continuously applying the following basic step to any pair of non-adjacent vertices of the corresponding reduced graph. For these two vertices, there are only two choices; one is to color them with the same color; another is to color them with distinct colors. In the Zykov-tree, if we represent the pre-work graph as the father node, the two choices mentioned earlier are two branches, and the corresponding reduced graphs are sons of the father node. The method terminates whenever all leaves of the Zykov-tree become complete. Therefore the chromatic number of the graph is the minimum of chromatic numbers of leaves.

Before describing Zykov's algorithm, we introduce two terms from Zykov.

Definition Let $G = (V, E)$ be a graph with non-adjacent vertices x and y. $G/xy = (V', E')$ is a *join* of G by adding the edge $<x,y>$. That is, $V' = V$; $E' = E \cup \{<x,y>\}$.

Definition Let $G = (V, E)$ be a graph with non-adjacent vertices x and y. $G{:}xy = (V', E')$ is a *contraction* of G by *identifying* x and y. That is,

$$V' = V - \{y\};$$

$$E = \{<u,v> \mid u \neq y,\ v \neq y, \text{ and } <u,v> \in E\} \cup \{<u,x> \mid <u,y> \in E\}.$$

Theorem 3.1 If x and y are non-adjacent vertices in G, then $\chi(G) = \min\{\chi(G/xy), \chi(G{:}xy)\}$.

Proof. For a proof of this theorem, see [*Br77*].

Theorem 3.1 can be applied to a graph recursively such that

$$\chi(G) = \min\{|G_1|, |G_2|, \ldots, |G_m|\},$$

where each G_i is an irreducible (complete) graph; that is, G_i is a leaf of the Zykov-tree.

We now present Zykov's algorithm in Figure 5. We can easily improve Zykov's algorithm by the branch-and-bound method. If there exists an irreducible subgraph K in the Zykov-tree, any reducible subgraph H in the Zykov-tree containing a clique of order $\chi(K)$ need not be reduced further. Unfortunately, this kind of branch-and-bound approach is not effective because the clique finding problem is also $\mathbb{NP}$-complete [*GJ*79]. A depth-first search with a branch-and-bound heuristic algorithm was proposed by Corneil and Graham in 1973 [*CG*73], which, instead of looking for an α-clique, found an α-cluster, a highly dense graph of order α, in $O(n^3)$ time.

```
Procedure Reduce(H: graph)
   begin
      if H is complete then return
      else begin
         choose any two nonadjacent vertices x, y in H;
         construct H:xy and H/xy;
         Reduce(H:xy);
         Reduce(H/xy);
      end
   end; (* Reduce *)

Program Zykov_tree_of_G
   begin
      Reduce(G);
   end.
```

Figure 5. Zykov's Algorithm.

Example 3.2 Figure 6 shows the Zykov-tree of graph G_0, where $G_1 = G_0/bd$, $G_2 = G_1/bc$, $G_3 = G_1{:}bc$, and $G_4 = G_0{:}bd$.

3. Dynamic Programming Approach.

As we mentioned earlier, a color class is an independent set. The dynamic programming approach employs the following property of MIS: for every k-coloring of a graph G there exists an p-coloring of G, where $p \leq k$, and the family of color classes of the p-coloring contains at least one MIS. Christofides first published this method in 1971 [$Ch71$]. So we would like to introduce the following theorem [$Ch71$] in advance.

Theorem 3.2 Any graph $G = (V, E)$ can be optimally colored by first coloring a MIS M_1 of G, next coloring a MIS M_2 of G_{V-M_1} and so on until all vertices are colored.

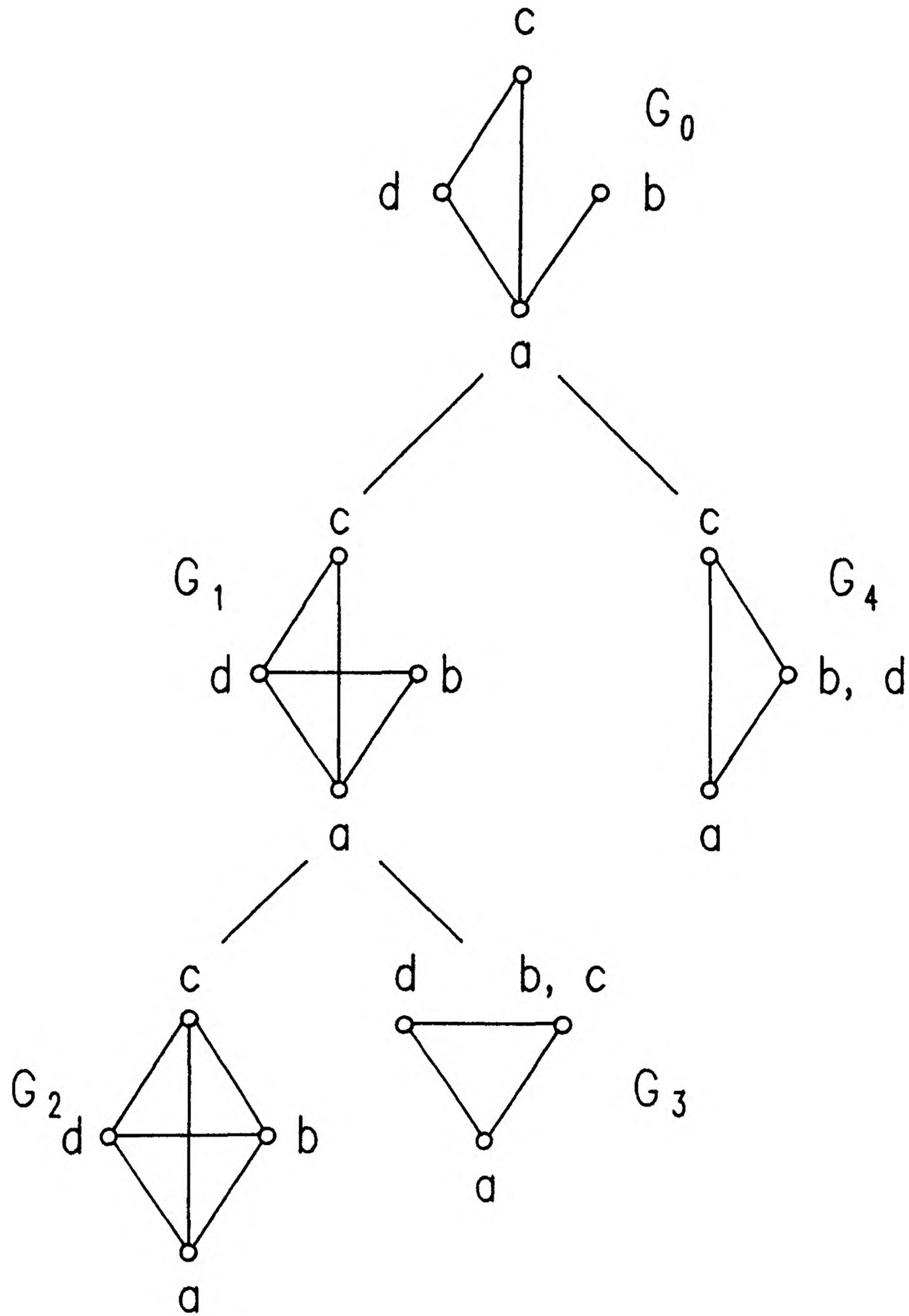


Figure 6. Example of Zykov-tree

Before going through Christofides's algorithm, we would like to define the maximal r-chromatic subset M_r and a recurrence relation.

Definition A *maximal r-chromatic* subset of $G=(V, E)$ is a maximal subset of V which can be colored with r colors but not with fewer.

Two facts can be inferred from the definition above: (1) any MIS of G is a maximal 1-chromatic subset of G; and (2) G is a $\chi(G)$-chromatic subset of itself.

Let $\aleph_k$ denote a family of maximal k-chromatic subsets of G, and M_k^i be the i-th element of $\aleph_k$. Then $\aleph_{k+1}$ is the union of T_i, where

$$T_i = \{M_k^i \cup S \mid S \text{ is a MIS of } G_{V-M_k^i}\}, \qquad \text{for } i = 1, \ldots, |\aleph_k|.$$

```
input G = (V, E)
output k is the chromatic number of G
begin
   k := 1;
   compute ℵ₁, a family of MIS's of G; [BK73]
   if order of ℵ₁ = 1  then STOP
   else loop
      T := φ;
      for each M in ℵk do
         for each MIS M' of G(V−M) do
            if M ∪ M' = V then
               begin k := k + 1; STOP end
            else begin
               T := T ∪ {M ∪ M'};
               maximize T such that every element in T is maximal;
            end
      k := k + 1;
      ℵk := T;
   end
end.
```

Figure 7. Christofides's Algorithm.

Example 3.3 Illustrate Christofides's algorithm by Figure 3.

$\aleph_1 = \{\{a\}, \{b, c, d\}, \{b, d, e\}\}.$

$T_1 = \{\{a\} \cup \{b, c, d\}, \{a\} \cup \{b, d, e\}\}.$

$T_2 = \{\{b, c, d\} \cup \{a\}, \{b, c, d\} \cup \{e\}\}.$

Since $\{b, c, d\} \cup \{a\}$ is a subset of $\{a\} \cup \{b, c, d\}$, we get rid of $\{b, c, d\} \cup \{a\}$.

$T_3 = \{\{b, d, e\} \cup \{a\}, \{b, d, e\} \cup \{c\}\}.$

Remove $\{b, d, e\} \cup \{a\}$ because it has already appeared in T_1. The same situation happens on $\{b, d, e\} \cup \{c\}$ since it is in T_2.

$\aleph_2 = \{\{a\} \cup \{b, c, d\}, \{a\} \cup \{b, d, e\}, \{b, c, d\} \cup \{e\}\}.$

$T_1 = \{\{a\} \cup \{b, c, d\} \cup \{e\}\}.$

$k = 3$, STOP.

Christofides's algorithm is a breadth-first search implementation of theorem 3.2; it wastes computation time and space because it traverses one step on all possible paths before going one more step further. One improvement of Christofides's algorithm is that the computation of MIS of a subgraph of G can be done by the fact that each MIS of a subgraph of G must be a subset of a MIS of G [*RF*73]. Thus, the MIS-finding [*BK*73] can be done only once. In 1974, Wang [*Wa*74] proposed a depth-first search implementation on the search tree which is much smaller than the search tree implied in Christofides's algorithm.

4. Implicit Enumeration Approach.

The implicit enumeration approach is another tree search method for solving the GCP; it is sometimes named the backtracking sequential method. There are two basic steps to this approach:

(1) Pre-ordering of vertices of G;

(2) Forward movement, the *coloring* procedure, which traverses the search tree vertex by vertex according to the pre-ordering sequence of vertices by assigning the smallest color of the feasible color set of the current uncolored vertex, and updates, if necessary, the upper bound q of the chromatic number.

The recursive procedure *coloring* constructs the frame of the search tree vertex by vertex until it reaches the following conditions: either every color of the feasible color set of the current vertex has been used, or no uncolored vertex is left. In the former case we backtrack to the previous vertex; in the latter case a better upper bound of the chromatic number of G is found. If we get a better upper bound q of $\chi(G)$, we update q and backtrack to the previous vertex. This approach terminates whenever we backtrack to the beginning vertex of the pre-ordering sequence of vertices of G.

One more question we have not mentioned is how to determine the feasible color set such that there is no redundant coloring. Let us take a look at the following theorem first.

Definition If a coloring of G can be derived from another coloring of G by permuting colors without changing the associated color classes. We say this coloring is *redundant*.

Example 3.4 Let G = (V, E), V = $\{v_1, v_2, v_3, v_4, v_5\}$. Define a coloring f of G as follow:

$f(v_1) = 0, \ f(v_2) = 0,$

$f(v_3) = 1, \ f(v_4) = 1,$

$f(v_5) = 2.$

If g is another coloring function of G such that

$g(v_1) = 1, \ g(v_2) = 1,$

$g(v_3) = 0, \ g(v_4) = 0,$

```
Input a graph G of order n;
Output
   q : integer; (* number of colors required *)

Procedure coloring (k: 1..n+1);
   var
      j: integer;
   begin
      if k = n+1 then update q
      else begin
         compute the feasible set of v_k ;
         if k = n then begin
            j ← 0;
            repeat
               j ← j + 1;
            until j in feasible(v_k) ;
            color v_k with j;
            coloring(k+1);
         end else begin
            j ← 1;
            repeat
               if j in feasible(v_k)  then begin
                  color v_k with color j;
                  coloring(k+1);
               end;
               j ← j + 1;
            until j = q;
         end
      end
   end; (* coloring *)

begin (* main *)
   let v_1, v_2, ... , v_n be a sequence of vertices of G according to a
      rearrangement of vertices;
   q ← upper bound of χ(G);
   coloring(1);
end.  (* main *)
```

Figure 8. Implicit Enumeration Approach.

$g(v_5) = 2$.

Then g is redundant because g can be derived from f by interchanging color 0 and 1. However, if we assigned v_2 with color 2 in the coloring g, g would be not redundant because the family of color classes of g, $\{\{v_3, v_4\}, \{v_1\}, \{v_2, v_5\}\}$, is different from that of f, $\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5\}\}$.

Theorem 3.3 Let $v_1, \ldots, v_{i-1}$ be a p-coloring. If no redundant coloring is to be generated, then the assigned color of v_i can not be greater than $p + 1$.

Proof. This theorem is proved by induction.

i = 1, there is only one color for v_i.

Assume that there is no redundant coloring for $v_1, \ldots, v_{i-1}$. For a p-coloring C of $v_1, \ldots, v_{i-1}$, if the extended coloring C′ of $v_1, \ldots, v_{i-1}, v_i$ is also p-coloring, C′ is not redundant since C is not redundant. If v_i must be assigned a new color, we know that v_i should be colored by color $p+1$. Otherwise $C'(v_i) > p+1$ is redundant because of the interchangeability between $C'(v_i)$ and $p+1$. Therefore, v_i can not be colored with color greater than $p+1$. Q.E.D.

Suppose $v_1, v_2, \ldots, v_{i-1}$ have already been colored with p colors. Then every feasible color j of v_i must satisfy the following conditions:

(1) $j \leq p+1$;

(2) j has not been assigned to any colored adjacent vertex of v_i ;

(3) $j \leq q-1$, where q is an upper bound of $\chi(G)$.

Brown [*Br*72] is the first person who used the implicit enumeration algorithm to solve the GCP. It is obvious that if we construct a clique C of G, then each vertex of C can be colored by only one possible color. Therefore, we can terminate Brown's algorithm when backtracking to the clique because the upper part of the search tree is linear. Brown's algorithm can be improved from the *forw_move* procedure, the *back_move* procedure, or both. The forw_move and the back_move are shown in Figure 9. The *forw_move* procedure can be improved by using either the dynamic reordering of the uncolored vertices [*Ko*79] or a look-ahead procedure [*Br*72]. On the other hand, Christofides [*Ch*75] advanced the *back_move* procedure and unintentionally ended up with a heuristic algorithm. In 1979, Brèlaz [*Br*79] made two errors in his Randall-Brown's modified algorithm and also arrived at a heuristic

algorithm. The correct version of the Brèlaz's algorithm was given by Peemöller [*Pe*83].

A very complete reference to the implicit enumeration approach of the GCP was given by Kubale and Jackowski [*KJ*85].

Example 3.4 An illustration of Brown's algorithm is given in Figure 3. For the preordering, the largest-first preordering will arrange the vertices in non-increasing degree.

With largest-first preordering:

a, c, e, b, d is the coloring sequence of vertices.

The first pass:

C(a) = 1, C(c) = 2, C(e) = 3, C(b) = 2, C(d) = 2.

upper bound q = 3.

Backtrack to vertex c. Since $F[c] = \phi$, the algorithm immediately backtracks again to vertex a, and **STOP.**

With arbitrary ordering: a, b, c, d, e.

the first pass:

C(a) = 1, C(b) = 2, C(c) = 2, C(d) = 2, C(e) = 3.

upper bound q = 3.

Backtrack to d. $F[d] = \phi$ after we remove the feasible color 3 (Since $q = 3$ now); we move back to c. Again $F[c] = \phi$ after we update the feasible color set, and $F[b] = \phi$ originally. We reach the top vertex a, and then **STOP.**

B. HEURISTIC ALGORITHMS

Before taking the coloring action, most graph coloring algorithms in existence determine both the vertex (or vertices) to be colored and the color to be utilized. That is, the execution order of choosing the next uncolored vertex and the next color can

```
Input a graph G of order n;
Output
   q : integer; (* number of colors required *)
   the coloring function;
Global Variable
   k : 1 .. n+1 (* index of the current processing vertex *)
Local Variable
   quit : boolean

Procedure forw_move
   begin
      compute the feasible set of v_k ;
      if feasible(v_k) = φ then
         return;
      else begin
         color v_k with the smallest feasible color;
         if k = n then begin
            k ← n + 1;
            return;
         end else forw_move;
      end
   end; (* forw_move *)

Procedure back_move
   begin
      if k = n + 1 then begin
         find the smallest index, r, such that v_r was colored with color q;
         k ← r − 1;
      end else k ← k − 1 ;
      return;
   end; (* back_move *)

begin (* main *)
   let v_1, v_2, ... , v_n be a sequence of vertices of G according to
      non-increasing degree;
   q ← upper bound of χ(G);
   k ← 1;
   quit ← false;
   repeat
      forw_move;
      if k = n+1 then
         update both q and the coloring function;
      back_move;
      if k = 1 then quit ← true ;
   until quit;
end. (* main *)
```

Figure 9. Brown's Algorithm.

change an algorithm from one to another. Consequently, this section will cover three methods of selection of alternatives: (1) vertex-color sequential algorithm, (2) color-vertex sequential algorithm, and (3) vertex-vertex pair scanning.

1. Vertex-color Sequential Algorithm.

Let $v_1, v_2, \ldots, v_n$ be an ordering of vertices of a graph G. The *vertex-color sequential* algorithm colors the first vertex v_1 with color 1, and then it makes use of the basic procedure recursively as follow:

> If $v_1, \ldots, v_{i-1}$ have been colored, then v_i is assigned the smallest possible color not occurring on adjacent vertices of v_i. v_i is colored with the new color if all existing colors do not fit for v_i.

The algorithm terminates whenever there is no uncolored vertex. The efficiency of the algorithm is mainly based on the ordering of vertices. It is evident that there is at least one optimal ordering of vertices. However, we have not found any algorithm which selects the optimal ordering from $n!$ possible orderings in polynomial time bound.

There are a number of algorithms in this category. We are going to review some well-known algorithms here. One of the first versions of the vertex-color sequential algorithms was proposed by Welsh and Powell [*WP67*], who arranged the vertices according to non-increasing degree. Such an algorithm is called the *largest-first sequential* algorithm (LFS). The total number of colors used by the LFS will not exceed $\max_{1 \le i \le n}\{\min\{i, \deg(v_i) + 1\}\}$.

In contrast to the LFS, another algorithm named *smallest-last sequential* algorithm (SLS) was proposed by Matula, Marble, and Isaacson [*MM72*]. The *smallest-last ordering* is found according to:

1. v_n is the vertex of the smallest degree of G;

2. v_i is the vertex having the smallest degree of the induced subgraph $G_{V-\{v_{i+1},\dots,v_n\}}$.

Lemma The number of colors used in the smallest-last sequential algorithm are bounded by $1+\max_{H_i}\{\min_{1\le j\le i}\{\deg_{H_i}(v_j)\}\}$, where H_i is a subgraph of G induced by $\{v_1, v_2, \dots, v_i\}$.

Proof. From the procedure of sequential algorithm, there is at least one feasible color between 1 and $1+\deg_{H_i}(v_i)$ for every vertex v_i. That is, $C(v_i)\le 1+\deg_{H_i}(v_i)$, where C is a coloring function of G. By the definition of SLS, $\deg_{H_i}(v_i) = \min_{1\le j\le i}\{\deg_{H_i}(v_j)\}$. So $C(v_i)\le 1+\min_{1\le j\le i}\{\deg_{H_i}(v_j)\}$. Therefore, $\chi(G)\le \max_{1\le i\le n}\{C(v_i)\}$ $\le \max_{1\le i\le n}\{1+\min_{1\le j\le i}\{\deg_{H_i}(v_j)\}\} = 1+\max_{1\le i\le n}\{\min_{1\le j\le i}\{\deg_{H_i}(v_j)\}\}$. Q.E.D.

Theorem 3.4 $\chi(G)\le 1+\max_{H\subseteq G}\{\min_{v\in V(H)}\{\deg_H(v)\}\}$.

Proof. It can be easily derived from the Lemma above because the family of $H_1, \dots, H_n = G$ is only a subset of the power set of G. Q.E.D.

The upper bound of theorem 3.4 is called *Szekeres-Wilf* bound [*SW*68]. It is evident that $\deg_{H_i}(v)\le \deg_G(v)$. Thus, we have a better upper bound of the chromatic number while using SLS.

Brèlaz [*Br*79] presented a dynamic way of ordering vertices, called *Dsatur* algorithm (DLF). The DLF ordering is determined as follow:

1. v_1 is the largest degree of G.

2. v_i is adjacent to the maximum number of distinct colors.

DLF successively colors the vertices in a DLF ordering with the smallest possible color, we obviously color a clique of v_1 first. Thus we obtain a lower bound, the order of a clique, of the chromatic number. The clique found by the DLF may not be maximal. We *may* obtain a maximal clique by taking advantage of the interchange

method proposed by Matula et al [*MM*72]. Note that the DLF is an exact algorithm for bipartite graphs.

The *interchange method* is a way of attempting to repack the existing colors by recoloring a subgraph of the partially colored graph whenever a new color is introduced. Suppose $v_1, \dots, v_{i-1}$ have already been colored in k colors. An *(a,b)-colored* subgraph of $V = \{v_1, \dots, v_{i-1}\}$ is the induced subgraph of vertices, $V' = \{v \in V \mid C(v) = a \text{ or } C(v) = b\}$. An *(a,b)-component* is a component of (*a*,*b*)-colored subgraph. Suppose that v_i has a feasible color *m*, $m \leq k$, then $C(v_i) = m$ and go to the next uncolored vertex. On the other hand, if the only feasible color for v_i is $k+1$, we consider every pair (*a*,*b*), $1 \leq a \leq b \leq k$. If there exist an (*a*,*b*)-colored subgraph such that in every (*a*,*b*)-component the vertices which are adjacent to v_i are of *at most one color*, then we recolor every adjacent vertex of v_i having the existing color *a* with color *b* and yield a feasible color *a* for v_i. Otherwise, v_i is colored with the new color $k+1$.

The combination of SLS and interchange method will color any planar graph in five or fewer colors [*MM*72].

2. Color-vertex Sequential Algorithm.

The *color-vertex sequential* algorithm is as follow:

1. k = 1;
2. Initially we place the first uncolored vertex into the color set C_k ;
3. All uncolored vertices are examined in order; any vertex which is not adjacent to any vertex of C_k is added to C_k ;
4. If there is no uncolored vertex, then **STOP** ; otherwise, k = k + 1, and goto step 2.

Peck and Williams [*PW*66] arranged the vertices by non-increasing order of degree, and performed the color-vertex sequential coloring. A few years later, Williams [*Wi*69] modified the Peck-Williams algorithm by pre-ordering the vertices; he made use of d^m instead of d, where d is the degree. d^m comes from the following recursive relation:

$$d_i^0 = \deg_G(v_i);$$

$$d_i^k = \sum_j a_{ij} d_j^{k-1}, \quad \text{where } [a_{ij}] \text{ is the adjacency matrix.}$$

William also mentioned that if $|V| = n$, then $m = \sqrt[3]{n}$ is generally sufficient.

The following algorithm is called *Approximately Maximum Independent Set* (AMIS) [*Jo*74] which is a heuristic algorithm for determining the MIS. Figure 10 shows the sketch of AMIS. In Figure 10, P is the prohibited set of vertices of each MIS finding; H is the subgraph of G induced by all uncolored vertices. Every member v of MIS is the vertex with the minimum degree in the current subgraph $H_{V(H)-P}$. Of course, P has to be updated after we select and color vertex v.

```
begin
   H := G;
   k := 0;
   while H is not null do begin
      k := k + 1;
      P := φ;
      while P ≠ V(H)  do begin
         find vertex v of minimum degree in H_V(H)-P ;
         C(v) := k ;
         P := P ∪ {v} ∪ adj(v) ;
      end ;
      H := subgraph of H induced by uncolored vertices;
   end ;
end ;
```

Figure 10. Approximately Maximum Independent Set

All the other heuristic algorithms in this section are capable of producing a worst case G having colors proportional to $\chi(G) * n$, where $n = |G|$. However, AMIS will color any graph G with n vertices in $O(\frac{n}{\log n})\chi(G)$ or fewer colors [*Jo*74].

Leighton [*Le*79] presented the *Recursive-Largest-First* algorithm (RLF), which makes use of the LFS strategy in the AMIS algorithm. Let U be the set of uncolored vertices which are not adjacent to any colored vertex, and W be the set of uncolored vertices which are adjacent to at least one color vertex. For each MIS, RLF chooses the vertex having maximal degree in H as the first element of MIS, and then selects another member v of MIS by the rule: (1) $|adj_H(v) \cap W|$ is maximal; and (2) $|adj_H(v) \cap U|$ is minimal if (1) is tied.

```
begin
   H := G;
   k := 0;
   while H is not null do begin
      k := k + 1;
      find v ∈ V(H) such that deg_H(v) is maximal;
      C(v) := k;
      construct U and W;
      while U ≠ φ do begin
         find v ∈ U such that |adj_H(v) ∩ W| is maximal;
         (tie is, if possible, broken by |adj_H(v) ∩ U| is minimal)
         C(v) := k;
         update U and W;
      end;
      H := subgraph of H induced by W;
   end
end;
```

Figure 11. Leighton's Algorithm

The RLF algorithm can color sparse graphs with the chromatic number near to $\frac{|V|^2}{|E|}$ in $O(|V|^2)$ time. Such graphs happen very often in practical application such as timetable scheduling.

3. Vertex-vertex pair scanning.

The *vertex-vertex pair scanning* will inspect all pairs of vertices. It is evident that this class of algorithms has a time-consuming step while selecting a candidate on the basis of analysis of all pairs of vertices.

Wood's algorithm [*Wo*68] arranges the order of pairs of nonadjacent vertices by the number of connection vertices, which are adjacent to both nonadjacent vertices. Figure 12 shows the frame of Wood's algorithm. Wood made use of the fact that an vertex of degree less than the number of colors can always be colored with one of the existing colors.

The heart of Wood's algorithm is quite sophisticated because there are many comparisons before the next move of the selecting pair of nonadjacent vertices. A method which simplified the next move of the selected pair of nonadjacent vertices by the operation, contraction, in Zykov's algorithm and dynamically chooses the pair with the largest number of connection vertices was proposed by Dutton and Brigham [*DB*81].

```
begin
   construct the queue L of pairs of nonadjacent vertices by
      non-increasing order of the number of vertices which are
      adjacent to both vertices of the indicated pair;
   p := 0; /* number of colors for the current partial coloring */
   repeat
      take pair (x,y) from the head of L;
      delete (x,y) from L;
      case 1: both x and y are colored
         go into next pair;
      case 2: exactly one vertex of x, y is colored. Assume x is uncolored,
         and y ∈ C_i, where C_i is the i-th color class.
         if (deg(x) ≥ p) and ( adj(x) ∩ C_i = φ) then
            C_i := C_i ∪ {x} ;
      case 3: neither x nor y is colored
         if (deg(x) ≥ p) or (deg(y) ≥ p) then
            if there exists the smallest index i ≤ p such that both x and y
               can be colored with i then
               C_i := C_i ∪ {x,y} /* otherwise go into next pair */
            else begin
               p := p + 1 ;
               C_p := {x,y} ;
            end;
   until (L is empty) or (no uncolored vertices left);
   if uncolored vertices left then
      do the sequential coloring for all uncolored vertices;
end ;
```

Figure 12. Wood's Algorithm.

```
begin
   while there is a pair of nonadjacent vertices do
      begin
         select the pair of nonadjacent vertices (x, y ) with the largest
            number of connection vertices;
         identify x and y;
      end;
   |existing vertices| is an upper bound of the chromatic number;
end;
```

Figure 13. Dutton-Brigham's Algorithm.

C. APPLICATIONS

This section indicates a number of applications that are most often encountered.

1. Loading Problem.

We are given a set of objects. Assuming that part of the objects can't be packed together because of some reason such as chemical contamination. The *loading problem* [*EC*71] [*Ch*75] is to find the minimum number of boxes to accommodate the objects. Let each object be a vertex of a graph, and two objects are joined by an edge if they can't be placed in the same box. Supposing that the capacity requirement of two objects is additive. That is, two objects of size O_1 and O_2 can by packed with a box of capacity $O_1 + O_2$. There are several cases which take account of both the size of the objects and the capacity of boxes.

Case 1. Object of same size and boxes of infinite capacity.
This is the standard coloring problem where each box corresponds to a color.

Case 2. Objects of same size O and boxes of same finite capacity B.
This is saying that no more than $\frac{B}{O}$ objects can be put together in the same box. That is, we add one more constraint that no more $\frac{B}{O}$ vertices can be assigned to the same color to the fundamental constraint that no adjacent vertices can be colored with the same color.

Case 3. Objects of different size and boxes of the same finite capacity.
Case 3 of the loading problem is similiar to the *knapsack problem* in which every two objects can be put together. Algorithms of this kind are highly similar to those of case
2

Case 4. Objects of different size and boxes of different capacity.

This is the most general case. All 4 cases are NP-complete and computationally equivalent.

2. Timetable Scheduling.

Given a set of jobs which are to be accomplished by a set of people with some hardware tools. Assume that the company has one of each kind of hardware tool, and every job has the same process time. The *timetable scheduling problem* [*Br*79] is to find a schedule such that all jobs can be executed with the minimum process time. Each job is denoted by a vertex of a graph; every two jobs are adjacent if they have to be performed by either the same person or the same hardware tool. Every period of the timetable is equivalent to a color of the graph coloring.

3. Resource Allocation.

We submit n jobs to a computer which owns m resources. Let us suppose that each job can be executed in a fixed time slot with a subset of the m available resources. Each job is represented by a vertex of a graph; the edge of two vertices is introduced if the associated jobs require a common resource which can't be allocated at the same time. The greatest resource utilization [*Ch*75] can be achieved by the optimal coloring of the vertices of the graph.

IV. VARIATIONS ON THE IMPLICIT ENUMERATION APPROACH

A. BACKGROUND

1. Terminology of Trees.

A *tree* is a connected acyclic graph. Vertices of a tree are called *nodes,* one of which is named the *root* of that tree. The edges of a tree are called *branches.* A *spanning tree* of a graph G is a tree of G having all vertices of G. After the root is removed, the remainder of the tree is partitioned into a family of disjoint sets, where each of these family is called a *subtree* of the root. Within each disjoint set, the node y which is connected to the root x is called a *son* of x. Meanwhile, x is called the *father* of y. A node with no son is called a *leaf.* Sons of the same father are said to be *siblings.* The number of sons of a node v is called the *degree* of v. The *degree* of a *tree* is the maximum degree of the nodes in the tree. The *level* of a node is recursively defined as follows: the root is initially at level 0; if a node is at level a, then its sons are at level (a + 1). The *depth* of a node v in a tree is the absolute difference between the level of v and the level of the root. The *height* of a node v in a tree is recursively defined as follow: the height of a leaf is initially 0; the height of v = 1 + max { height of v_i | for every son v_i of v }.

Example 4-1 In Figure 14, node a is a root of TS, node b as well as node c are sons of node a. Nodes d, g, h, i, and j are leaves of TS. Nodes g and h are the siblings of node i because they have the same father node e.

Depth(a) = 0;

depth(b) = 1; depth(c) = 1;

depth(d) = 2; depth(e) = 2; depth(f) = 2;

depth(g) = 3; depth(h) = 3; depth(i) = 3; depth(j) = 3.

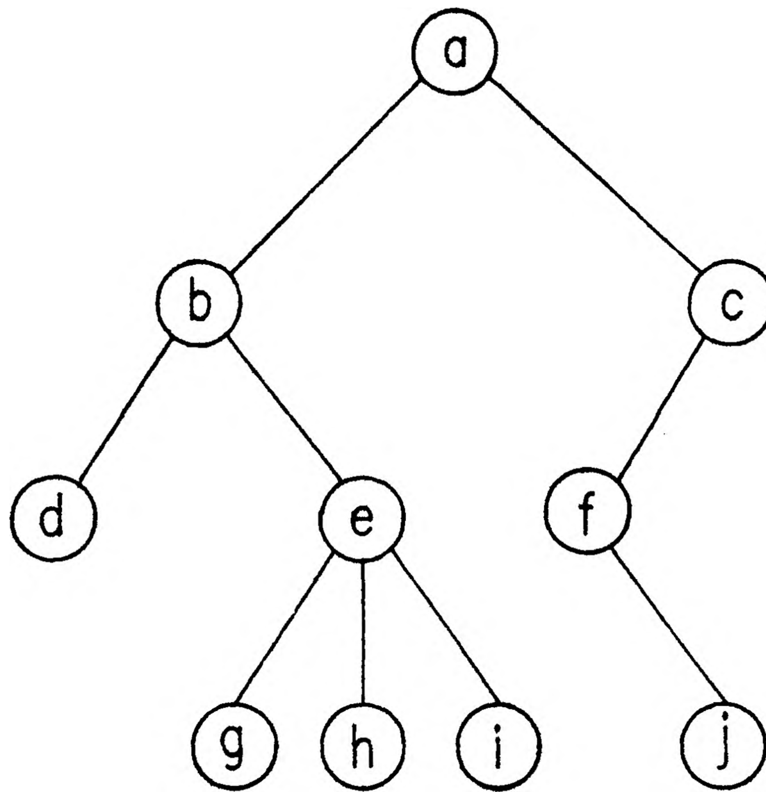


Figure 14. Example of tree TS

Height(a) = 3;

height(b) = 2; height(c) = 2;

height(e) = 1; height(f) = 1;

height(d) = 0; height(g) = 0; height(h) = 0; height(i) = 0;

height(j) = 0.

Degree(a) = 2; degree(b) = 2; degree(c) = 1; degree(d) = 0;

degree(e) = 3; degree(f) = 1; degree(g) = 0; degree(h) = 0;

degree(i) = 0; degree(j) = 0.

Degree of TS is 3.

2. Tree Search Techniques.

For constructing a search tree, we have to visit all the nodes of the search tree systematically. Depth-first search and breadth-first search [AH83][AH74][HS78] are two common ways to accomplish this work.

a. Depth-first Search (DFS).

For a connected graph G, we start at a vertex v_0 , mark it as having been visited, and then visit an unmarked vertex v_1, which is adjacent to v_0. Next we visit an unmarked vertex v_2, which is adjacent to v_1. We continue to penetrate the graph G until a vertex v_m , which has no unmarked adjacent vertex, is met. At this time, we backtrack from v_m to its previous adjacent vertex v_{m-1}, and then apply the same process to v_{m-1}. After backtracking to v_0 again, we terminate the algorithm.

```
Procedure DFS(w)
   begin
      mark(w);
      for each vertex v in adj(w) do
         if v has not been marked then
            DFS(v);
   end;
```

In constructing a search tree, we start at the root of the search tree, and then penetrate the tree via branches until a leaf is met. At that time we backtrack to the father of that leaf and do the same penetration work. Finally, we terminate the tree traversing when the root of the search tree is reached again.

Example 4-2 In Figure 14, we apply the DFS on TS by starting at the root a. Assume that we visit sons of a father from left to right. Then we can visit the nodes in the order a, b, d e, g, h, i, c, f, and j.

b. Breadth-first Search (BFS).

For a connected graph G, we start at a vertex v_0 , mark it as having been visited, and then visit all unmarked vertices which are adjacent to v_0 in an order such as $v_{01}, v_{02}, \dots, v_{0m_0}$. Next we visit all unmarked adjacent vertices of v_{01} such as $v_{011}, v_{012}, \dots, v_{01m_1}$, all unmarked adjacent vertices of v_{02} such as $v_{021}, v_{022}, \dots, v_{02m_2}$, ..., and all unmarked adjacent vertices of v_{0m} such as $v_{0m1}, v_{0m2}, \dots, v_{0mm_m}$. We continue this process until there is no unmarked vertex. In other words, BFS recursively explores a pending vertex (marked node not yet explored) v and all sons of v by initially starting from a vertex v_0 of a graph.

```
Procedure BFS(w)
   Var
      Q: queue of pending vertices;
   begin
      mark(w);
      empty Q;
      add w to Q;
      repeat
         get the first pending vertex x from Q;
         for each vertex v in adj(x) do
            if v has not been marked then begin
               mark(v);
               add v to Q;
            end;
      until Q is empty;
   end;
```

In constructing a search tree, we start at the root v_0 of the search tree and then visit all the sons of v_0 such as $v_{01}, v_{02}, \dots, v_{0m}$. Next we visit all the sons of v_{01}, all the sons of v_{02}, ..., and all the sons of v_{0m}. The procedure, BFS, is called level by level until there is no pending node.

Example 4-3 In Figure 14, BFS is applied on TS by starting from the root a. Assume we visit the sons of a father from left to right. Then we traverse the nodes of TS in the order a, b, c, d, e, f, g, h, i, and j.

So far, we describe DFS and BFS in a connected graph. How can we apply DFS and BFS to a graph? Taking a closer look at the algorithms of DFS and BFS, we discover that these algorithms terminate whenever a maximal connected subgraph of a graph is found. Thus, the DFS (or BFS) of a graph is carried out by repeatedly calling DFS (or BFS) from a new unmarked starting vertex.

B. BACKTRACKING

For many problems which we have encountered so far, there is a certain deterministic approach, which takes a certain amount of computational work, for obtaining a solution. For some combinatorial problems, however, there is no such approach. In this case, we may start on one attempt at a solution. If discovering that the solution cannot be achieved under the direction of the first attempt after an amount of work, we have to make an adjustment on the first attempt and start over with a second attempt. To solve this kind of problem, we must search through a *finite* set of possible solutions. Since we do not know the positive principle of searching direction which leads to a solution, this nondeterministic approach is usually very slow for a large-scale problem. For example, the 4-queens problem, placing 4 queens on a 4x4 chessboard such that no queens can attack another queen (i.e. no queens on the same row, column, or diagonal), there are 16 possible positions for 4 queens. The *brute-force approach* will evaluate $\binom{16}{4}$ possible configurations one by one until a solution is found. However, if we do a clever organization of the finite set of the possible solutions, most of the configurations will not be visited in searching. One way to achieve this job is to employ the backtracking technique.

Backtracking[*HS*78][*Hu*82] is a technique of organizing a search tree on a finite set of possible solutions such that at one time a subset of possible solutions can be eliminated from further consideration. To solve problems with backtracking, we have to solve the following problems: (1) How to systematically search the finite set of possible solutions? (2) How to set the bounding functions to eliminate a subset of possible solutions from further consideration? For the searching part of backtracking, one first represents the finite set of possible solution as a tree and then traverses the generated tree by DFS. Usually the finite set of possible solutions is expressed as a n-tuple vector $(x_1, x_2, \dots, x_n)$, where each x_i is chosen from a possible component set. For example, in the 4-queens problem, we represent each possible configuration as a 4-tuple vector (x_1, x_2, x_3, x_4), where component x_i indicates the position of the chess queen on the i-th row. The possible component set of x_i is $\{1, 2, 3, 4\}$. Before introducing the enumeration of $(x_1, x_2, \dots, x_n)$, we define the lexicographical ordering. A vector $(x_1, x_2, \dots, x_n)$ is *lexicographically smaller than* $(y_1, y_2, \dots, y_n)$ if and only if there exists i, $1 \leq i \leq n$, such that $x_i < y_i$ and $x_j = y_j$ for all $1 \leq j < i$. The search of backtracking is done by lexicographically enumerating the vectors starting from the lexicographically smallest vector.

The *bounding functions* can be the *explicit constraints* (except the constraints deciding the finite set of possible solutions), the *implicit constraints, or both.* Some problems may not have the implicit constraints; for example, the 4-queens problem, $1 \leq x_i \leq 4$ form the set of possible solutions, and those conditions that two queens are neither on the same column nor on the same diagonal are the explicit constraints. There are no implicit constraints in the 4-queens problem. However, in optimization problems such as minimum optimization, which is to find the global minimum among all possible solutions, the current lower bound of the possible solutions is an implicit constraint. The search tree built by a backtracking algorithm is called a *backtrack-tree*.

Leaves of the backtrack-tree are either solutions of the problem or dead nodes which are found by the bounding functions.

```
Input: a n-tuple vector (x1, x2, ... , xn) representation of a problem
Output: all possible solutions of the problem
Global Variable
   k: the current component of the n-tuple vector;
   CIBF: the current implicit bounding functions;

Procedure backtrack(k)
   begin
      compute the feasible set Fk of xk using bounding functions;
      for each x in Fk do
         if CIBF(x) = true then begin
            xk <- x;
            if k = n then begin
               save the path from the root to this leaf;
               update, if necessary, CIBF;
            end else backtrack(k+1);
         end;
   end;


begin (* main *)
   initialize CIBF;
   backtrack(1);
end.
```

Figure 15. General Backtracking Algorithm

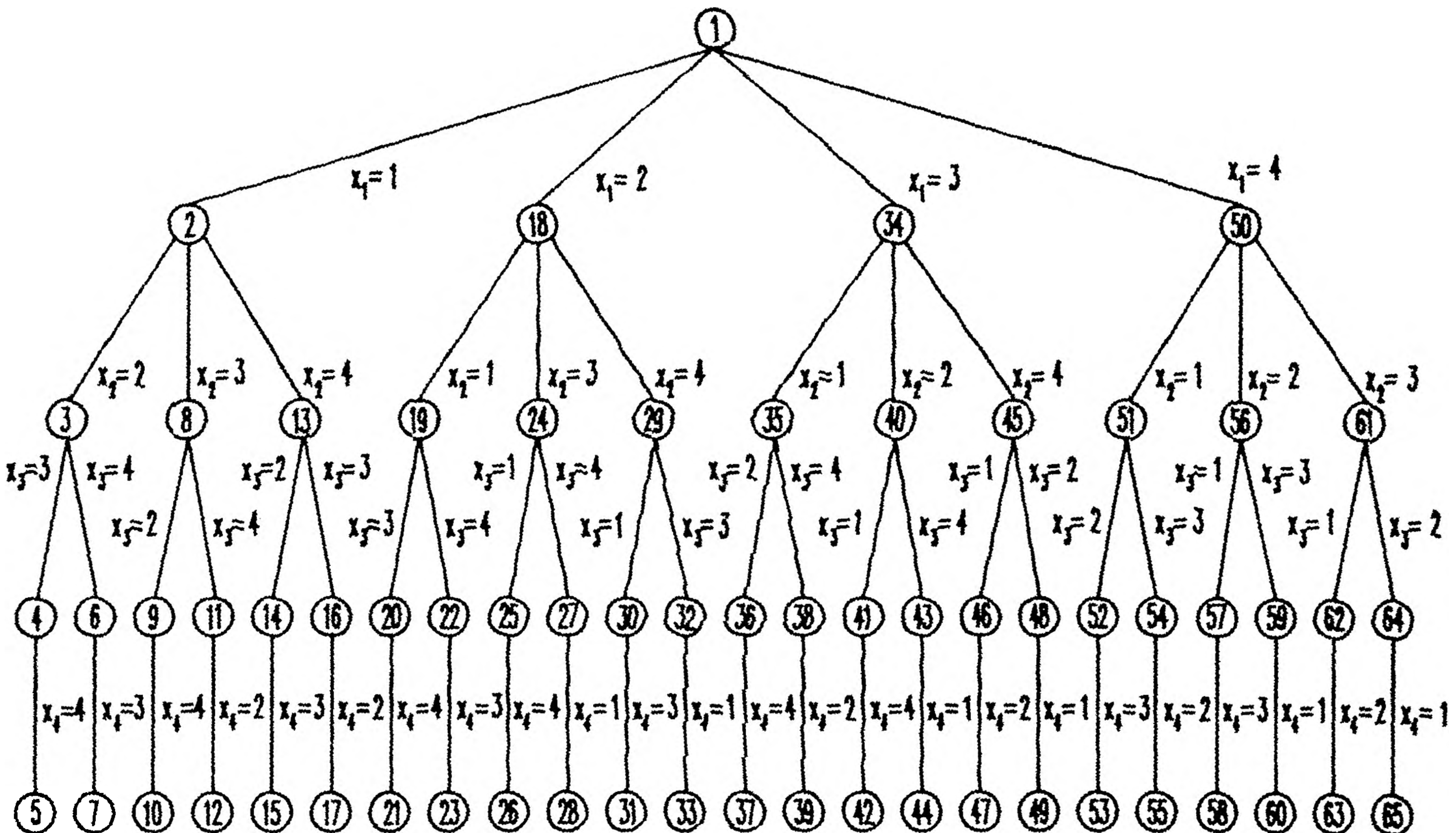


Figure 16. Tree representation of the 4-queens problem

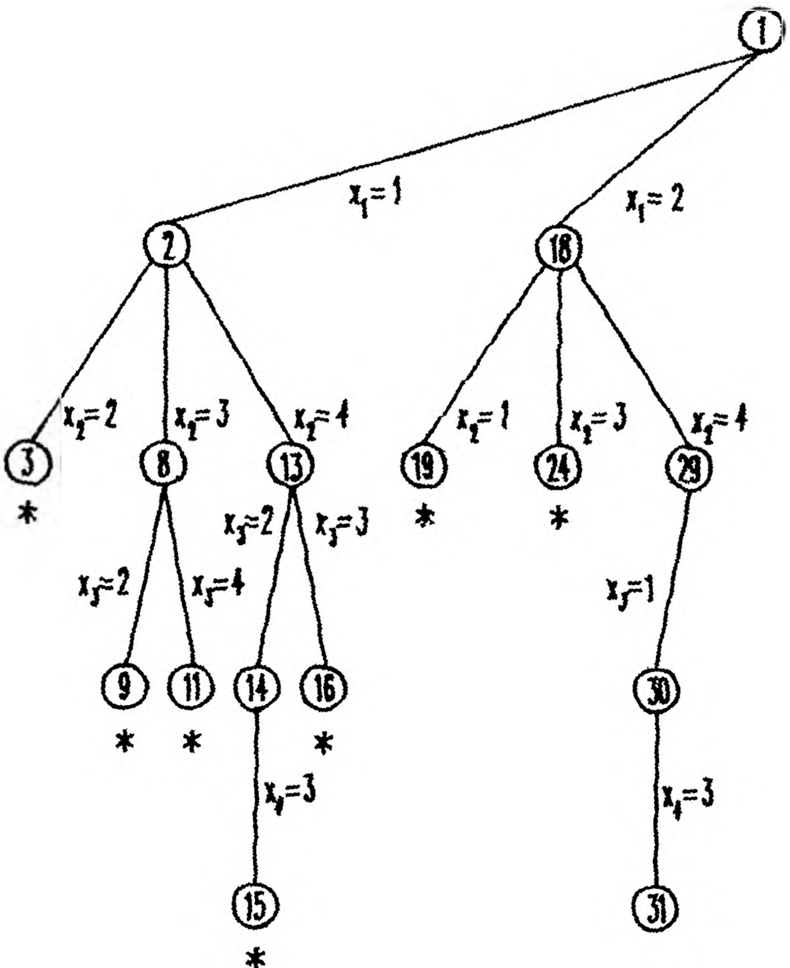


Figure 17. Backtrack-tree of 4-queens problem

Example 4-4 The tree representation of the feasible set (called the tree of 4-queens in this example), is shown in Figure 16, and the backtrack-tree of the 4-queens problem are shown in Figure 17. The nodes are labeled by the ordering of tree traversing in the tree of 4-queens. The branches are labeled by possible values of the x_i's assigning from the root to it; for example, the node 23 represents that $x_1 = 2$, $x_2 = 1$, $x_3 = 4$, and $x_4 = 3$. There are 24 (4!) possible configurations, which are described as leaves in the tree of 4-queens. In the backtrack-tree of 4-queens, the picture shows the steps that the backtracking technique goes through as it tries to find a solution. The star beside a node indicates that further consideration of the subtree rooted at that node can be disregarded. The backtrack-tree, which has 16 nodes, has already cut 15 nodes from the tree of 4-queens as the first solution is found.

C. BRANCH-AND-BOUND

The branch-and-bound method [*HS*78][*LW*66][*Mi*70] is another powerful alternative to do the exhaustive enumeration on a search tree. In contrast to the backtracking which is a DFS-like method, the *branch-and-bound* is a BFS-like strategy which generates all sons of the current node before visiting another node. The search tree of the branch-and-bound is called the *BNB-tree* Each node of the BNB-tree represents a class of possible solutions to the problem. All nodes but the solution nodes of the BNB-tree are called *pending nodes*, and the union of all pending nodes is the set of all possible solutions. The algorithm begins by assigning the formal representation of the original problem to the root of the BNB-tree. The job of branching is to replace a pending node v by all sons of v. That is, branching will divide a subproblem into a class of subsubproblems. The algorithm stops when it is not possible to do any further branching. The branch-and-bound requires a buffer for temporarily buffering the pending nodes. Basically there are two strategies to implement the buffer: One is FIFO (first in first out); another is LIFO (last in first out).

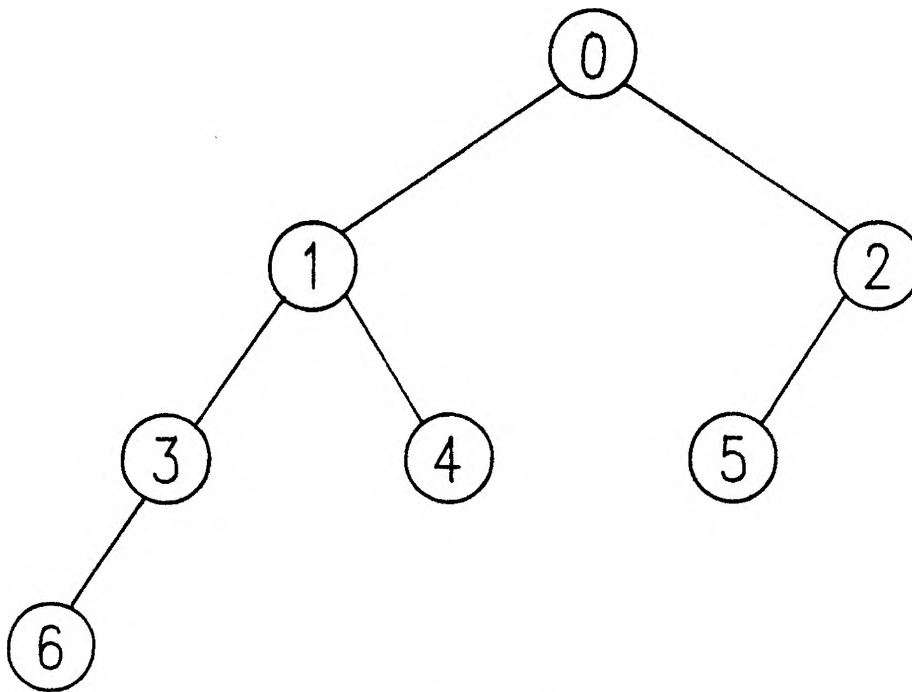
```
Input: root node of the BNB-tree
Output: solutions

Program branch-and-bound
   Var
      buffer: sequence of pending nodes of BNB-tree;
   begin
      empty buffer;
      add root node of BNB-tree to buffer;
      while buffer is not empty do begin
         get node x from buffer;
         for each son x' of x do begin
            if x' is a possible solution then begin
               save x' ;
               update the bounding functions;
            end;
            add x' to buffer;
         end;
      end;
   end.
```

Figure 18. General Branch-and-bound Algorithm

Example 4-5 Figure 19 shows FIFO BNB-tree and LIFO BNB-tree, where the labels of nodes represent the ordering of the tree traversing.

In the general branch-and-bound algorithm, the selection rule for the next pending node is in a *blind* sense. This kind of selection rules does not choose a node, which has a good chance to reach the solution quickly, according to the degree of *preference*. One way to speed up the branch-and-bound technique is to find a nice *preference function*, and then the selection of the next node from the buffer is totally based on the preference function. We will discuss an intelligent branch-and-bound algorithm on the GCP in the later section.



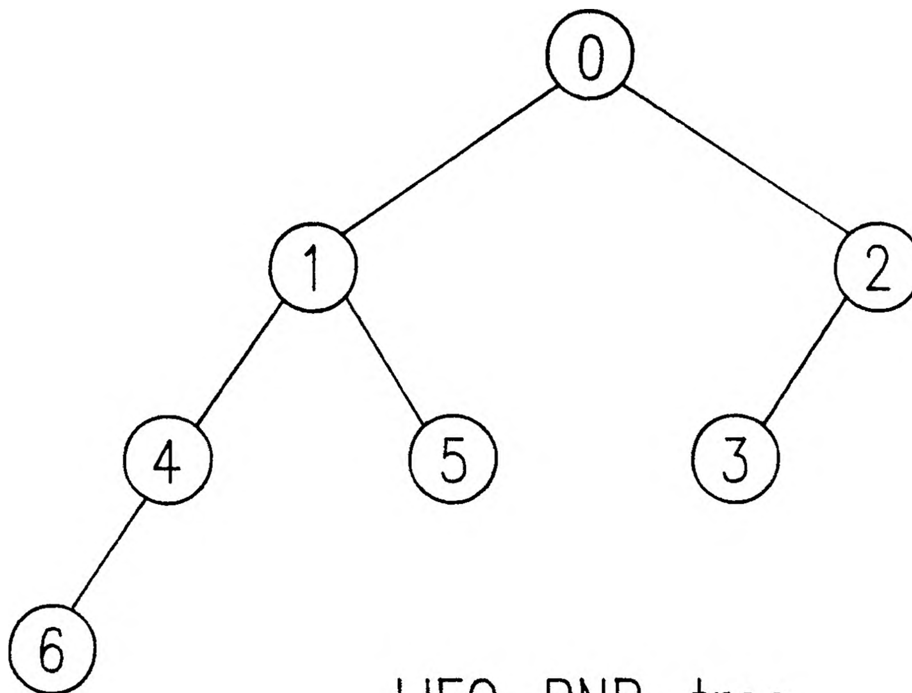


Figure 19. Examples of BNB-tree

```
Input:
    root node of the BNB-tree;
    preference function P;
Output:  solution nodes

Program intelligent_branch_and_bound
    Var
        buffer: sequence of pending nodes of BNB-tree;
    begin
        empty buffer;
        add root node of BNB-tree to buffer;
        while buffer is not empty do begin
            get node x with greatest degree of preference P(x) from buffer;
            for each son x' of x do begin
                if x' is a possible solution then begin
                    save x' ;
                    update the bounding functions;
                end;
                add x' to buffer;
            end;
        end;
    end.
```

Figure 20. Intelligent Branch-and-bound Algorithm

D. PROGRAMMING FRAME OF BACKTRACKING ON THE GCP

The graph-coloring system we designed is based on Brown's algorithm which was mentioned in chapter 3. Before describing the graph coloring system, we introduce several terms. In a graph, the vertices which have been labeled by a suitable color are called *colored vertices* In contrast to the colored vertices, we name the vertices which are going to be colored *uncolored vertices*. Therefore, the vertices of a graph are divided into two parts: colored vertices and uncolored vertices. A pool of the colored vertices of a graph is called the *core* of a graph. A pool of the uncolored vertices is called the *periphery* of a graph. For a vertex v of a graph, the number of neighbors of v in the core is called the *chrome-degree* of v; and the number of neighbors in

periphery is called the *white-degree* of v. Two key operations in the graph-coloring system are *newcolor* and *mergecolor*. In the graph-coloring system, we always put core ahead of periphery. The ordering of core is as follows: color1, color2, ..., color$|core|$. The jobs of both newcolor and mergecolor are listed in Figure 21.

```
Procedure newcolor(i, j)
   begin
      exchange vertices i and j;
      for each vertex v in (adj(i) ∪ adj(j)) do
         update both white-degree(v) and chrome-degree(v);
   end;


Procedure mergecolor(i, j)
   begin
      merge uncolored vertex i into colored vertex j;
      move the last vertex w of the periphery to the position of vertex i;
      for each vertex v in (adj(i) ∪ adj(j) ∪ adj(w)) do
         update both white-degree(v) and chrome-degree(v);
      shrink the periphery by discarding the last vertex;
   end;
```

Figure 21. Procedures newcolor and mergecolor

The process of the graph coloring system can conveniently be presented as a cyclic process. Within an unit step, both next-uncolored-vertex (nucv) and next-color-vertex (ncv) are chosen according to a nucv-selection function and a ncv-selection function, and then either *newcolor(nucv, ncv)* or *mergecolor(nucv, ncv)* is performed. The coloring process stops as soon as the periphery becomes empty. We will discuss several nucv-selection functions as well as ncv-selection functions in a later chapter. Note that nucv is executed first, and then a feasible color ncv for nucv is chosen in the graph-coloring system.

In the backtrack algorithm on the GCP shown below, the root of backtrack-tree is the original graph G. The nodes of backtrack-tree represent the subgraphs of G after performing either the *newcolor* operation or the *mergecolor* operation. Every branch denotes a pair of (nucv, ncv) corresponding to the father node. The bounding function q is initially set to the upper bound of $\chi(G)$. We update q whenever a better coloring path in backtrack-tree is found. Procedure *node_to_process* indicates the proper node of the coloring path in backtrack-tree. Along the coloring path in backtrack-tree, *node_to_process* advances one branch at a time and also backtracks branch by branch except the first backtracking movement of a path of complete coloring. Variable subgr[0 .. n] represents the nodes on a possible path of complete coloring (from the root to a leaf).

```
Input a graph G of order n
Output
   q: integer; (* χ(G) *)
Global Variable
   subgr[0 .. n]: nodes on the current path of coloration;
   k: the index of the current processing node of the path of coloration;
   nucv ∈ subgr[0 .. n]: the uncolored vertex which is to be colored next;
   ncv ∈ subgr[0 .. n]: best color for vertex nucv;
   quit: boolean;

Program backtrack_on_GCP
   begin
      k ← – 2; /* initialization */
      q ← upper bound of χ(G);
      quit ← false;
      repeat
         forward;
         backward;
      until quit;
   end.
```

Figure 22. Backtracking scheme on the GCP

```
Procedure node_to_process(m)
   begin
      if m = k + 1 then begin
         copy subgr[k] to subgr[k+1] ; (* move to subgr[k+1] *)
         k ← k + 1;
      end else if m < k then begin
         move back to subgr[m] ;
         k ← m;
      end else begin
         writeln('check m-value');
         halt;
      end;
   end;
```

Figure 23. Procedure node_to_process in backtracking and branch-and-bound

```
Procedure forward
   begin
      if k = -2 then begin (* place a graph into the root *)
         subgr[0] ← graph G;
         k ← 0;
      end else begin
         node_to_process(k+1);
         if ncv is a new color then
            newcolor(nucv, ncv)
         else mergecolor(nucv, ncv);
         if periphery is empty then begin
            update q;
            return;
         end;
      end;
      choose nucv according to nucv-selection function;
      choose ncv according to ncv-selection function;
      if ncv < q then forward;
   end;
```

Figure 24. Procedure forward in backtracking

```
Procedure backward
  begin
    if k = 0 then quit ← true
    else if k = n then begin
      find the smallest index r such that subgr[r].ncv = q ;
      node_to_process(r − 1);
    end else node_to_process(k − 1);
    update ncv;
    if there is no feasible color for nucv then
      backward;
  end;
```

Figure 25. Procedure backward in backtracking

Basically the backtracking algorithm on the GCP is an exact algorithm. It is usually difficult to color a graph of order over 50 using the backtracking algorithm. We will present various nucv-selection functions as well as ncv-selection functions in chapter 6.

E. PROGRAMMING FRAME OF BRANCH-AND-BOUND ON THE GCP

Basically our branch-and-bound implementation is a modification of the intelligent branch-and bound method. Instead of visiting all possible nodes which are in the same level of the search tree, we use DFS to find all pending nodes (those which can be partially colored with $m+1$ colors) of all m-partially colored nodes. Starting from the 0-partially colored node of a given graph, we search all 1-partially colored pending nodes by DFS. Then we find all 2-partially colored pending nodes from each of 1-partially colored pending nodes in a sequence in accordance with the degree of preference. We continue this process. The coloring process stops as soon as the leaf of a complete coloring appears.

In the branch-and-bound algorithm on the GCP shown below, we use the *sorted list outbuf*, which treats the degree of preference as the priority, to buffer all pending nodes which are partially colored with m colors and the *priority queue inbuf* to buffer all pending nodes which are partially colored with m+1 colors. Subgr [0 .. n] is the working space for running DFS on the root subgr [0]. Variable k is the index of the current node of the working space subgr[0 .. n]. We initially place the given graph G into the *outbuf*, and then recursively do the process cycle: For every node v with the highest priority in *outbuf*, finding all pending nodes, which call the *newcolor* procedure in the previous movement, of the subtree with the root v in DFS, and placing them into the priority queue *inbuf* according to the degree of preference. After finishing each process cycle, we copy all nodes of *inbuf* to *outbuf* in the non-increasing order of priority of nodes. Procedure *savestatus* inserts the pending node into *inbuf* with respect to the computed degree of preference. Procedure *getnextstate* is to get a new node with the highest priority from *outbuf*.

There are four problems to be solved in the branch-and-bound algorithm on the GCP: (1) How to find a *good* (heuristic) preference function? (2) How to handle a huge set of pending nodes whose size grows in an exponential way corresponding to the order of a given graph? (3) What kind of data structure is suitable for *inbuf* and *outbuf* in order to speed up the in-out actions of nodes of *inbuf* and *outbuf*? (4) How to find *good* (heuristic) *nucv-selection* as well as *ncv-selection* function? Since the GCP is $\mathbb{NP}$-complete, we suspect there is no *perfect* (exact) preference function. However, we will propose a preference function construction in chapter 7. In practice, we limit the number of pending nodes stored in the buffers. That is, the branch-and-bound algorithm on the GCP becomes a heuristic GCP algorithm. We choose the data structure, *heapsort*, to implement *inbuf* and *outbuf* because heapsort is an $O(n \log n)$ comparison sort. The detail description of heapsort will be in chapter 5. The

nucv-selection functions as well as the *ncv-selection* functions are the same functions mentioned in the backtracking algorithm on the GCP.

```
Input a graph G of order n
Output
   q: integer; (* χ(G) *)
Global Variable
   inbuf, outbuf: sequence of pending nodes which have the same core size;
   subgr[0 .. n]: working space;
   k: the index of the current processing node of the current working subtree;
   nucv ∈ subgr[0 .. n]: the uncolored vertex which is to be colored next;
   ncv ∈ subgr[0 .. n]: best color for vertex nucv;
   quit: boolean;

Program branch_and_bound_on_GCP
   begin
      k ← − 1;
      empty inbuf;
      empty outbuf;
      place graph G into outbuf;
      quit ← false;
      repeat
         forward;
         if not quit then backward;
      until quit;
   end.
```

Figure 26. Branch-and-bound scheme on the GCP

```
Procedure getnextstate
   begin
      if outbuf is empty then begin
         outbuf ← inbuf; (* maintain the same ordering *)
         empty inbuf;
      end;
      subgr[0] ← the first element of outbuf;
      delete the first element of outbuf;
      k ← 0;
   end;
```

Figure 27. Procedure getnextstate in branch-and-bound

```
Procedure savestatus
   begin
      compute the degree of preference of subgr[k] ;
      place subgr[k] as well as its degree of preference into the suitable
         position of inbuf w.r.t. the degree of preference;
   end;
```

Figure 28. Procedure savestatus in branch-and-bound

```
Procedure forward
   begin
      if k = - 1 then getnextstate
      end else begin
         node_to_process(k+1);
         if ncv is a new color then begin
            newcolor(nucv, ncv)
            savestatus;
            return
         end else mergecolor(nucv, ncv);
         if periphery is empty then begin
            quit ← true;
            return;
         end;
      end;
      choose nucv according to nucv-selection function;
      choose ncv according to ncv-selection function;
      forward;
   end;
```

Figure 29. Procedure forward in branch-and-bound

```
Procedure backward
   begin
      if k = 0 then k ← − 1
      else begin
         node_to_process(k − 1);
         update ncv;
      if there is no feasible color for nucv then
         backward;
   end;
```

Figure 30. Procedure backward in branch-and-bound

V. IMPLEMENTATION NOTES

In order to compare the time efficiency of various algorithms on the same experimental ground, all algorithms were coded in *Turbo Pascal 4.0* from Borland International and run on an 16 MHz IBM PS2-80.

A. GRAPHS REPRESENTATIONS

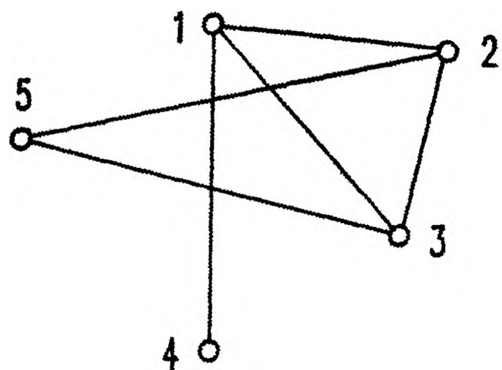
There are two common ways to represent a graph. One is the *adjacency matrix;* another is the *adjacency list.* An adjacency matrix for the graph G = (V, E) of order n is an n x n matrix $[a_{ij}]_{nxn}$ such that

$$a_{ij} = 1 \quad \text{if } <v_i, v_j> \in E\ ;$$
$$= 0 \quad \text{otherwise.}$$

Because G is a simple undirected graph, $a_{ij} = a_{ji}$, and $a_{ii} = 0$. In the adjacency list representation of a graph, each vertex has an associated list of its adjacent vertices.

Example 5-1 Figure 31 shows the adjacency matrix and the adjacency list of a graph G.

The adjacency matrix representation is convenient for the graph algorithms which frequently check whether certain edges exist because the time for deciding the existence of an edge is fixed and independent of $|V|$ and $|E|$. The initialization of the adjacency matrix requires $O(|V|^2)$ time even if a graph has edges with the property, $|E| \ll |V|^2$. In our experiment, we select the adjacency matrix representation because of (1) the fixed time for deciding the existence of an edge (this operation occurs very often), and (2) the space efficiency (the rows of an adjacency matrix are represented by bit vectors which are implemented by the set type in Turbo Pascal).



adjacency matrix of G:

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	1
3	1	1	0	0	1
4	1	0	0	0	0
5	0	1	1	0	0

adjacency list of G:

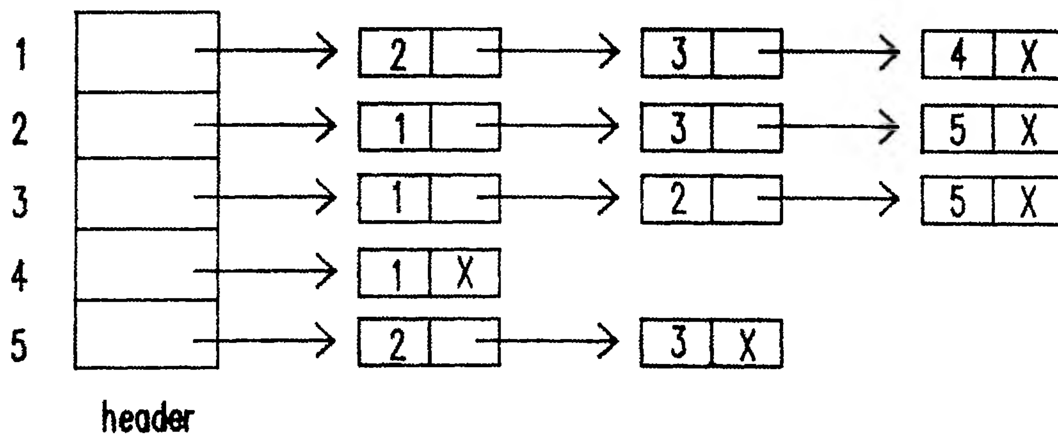


Figure 31. Representations of a graph

In addition to the adjacency matrix, additional information is required to represent the current status of the coloring of nodes in the search tree. The basic data type for a node in the search tree is shown as follows:

```
type
    vertex      = 0 ..|V| - 1 ;
    vrtxset     = set of vertex;
    vrtxmat     = array [vertex] of vrtxset;
    vrtxarr     = array [vertex] of bytes;
    graphtype   = record
        adj                 : vrtxmat;
        chrome-degree       : vrtxarr;
        white-degree        : vrtxarr;
        feasible            : array [ -1 ..|V| ] of integer;
        feastop             : integer;
        rsmpt               : integer;
        nucv                : integer;
        ncv                 : integer;
        fclr                : integer;
        clr                 : integer;
        vrtx                : integer;
    end;
```

Initially we assume that there is a one-to-one mapping between the set V and the integer sequence $0 .. |V| - 1$. Eleven fields are assigned to the graphtype.

adj	is an adjacency matrix in set form.
chrome-degree	is the number of adjacent colored vertices.
white-degree	is the number of adjacent uncolored vertices.
feasible	is a sorted list containing all feasible colors which have not been assigned to ncv yet.
feastop	is the index of the head of the sorted list feasible.
rsmpt	is the lowest index of a node which performs the procedure *newcolor* to the current highest color.
nucv	is an uncolored vertex of the highest degree of preference.
ncv	is a feasible color, which has the highest degree of preference, for nucv.
fclr	is the first vertex of core.
clr	is the last vertex of core.

vrtx is the last vertex of periphery.

It requires $O(|V|^2)$ time to formalize a graph to the root of the search tree.

B. MEMORY SWAPPING

As we declared in the backtracking and branch-and-bound programming frame, subgr[i], $0 \leq i \leq n$, is in graphtype shown in the previous section. The memory space for a graphtype grows in $O(|V|^2)$. Consequently, the declaration of subgr[0 .. n] requires $O(|V|^3)$ space reserved from the main memory. However, the working space is limited. So the memory swapping is required for running the graphs of huge size. In the obvious sense, the memory swapping will slow down the problem solving time.

In the memory swapping, we keep *(1 + physlimit)* nodes in the main memory. The content of k always locates between 0 and physlimit in a circular list form; that is, k = k mod (physlimit + 1). The *inmemory*[0 .. physlimit] are used to bookkeep whether subgr[0 .. physlimit] are in the main memory.

inmemory[i] = 1 if subgr[i] is in the main memory;

= 0 otherwise.

For the forward movement, if inmemory[next-node] = false, where next-node = ((k + 1) mod (phylimit + 1)), then we process subgr[next-node] in the main memory. Otherwise we push subgr[next-node] to *stackfl*, which hold the overflow nodes in LIFO, and process subgr[next-node]. For the backward movement, if inmemory[prior-node] = true, where prior-node = ((k − 1) mod (physlimit + 1)), , we process subgr[prior-node] in the main memory. Otherwise we pop subgr[prior-node] from *stackfl* , set inmemory[prior-node] = true, and process subgr[prior-node]. Figures 32-34 show how the memory swapping is placed into the backtracking algorithm on the GCP.

```
Global Constant
    physlimit
Global Variable
    inmemory: array [0 .. physlimit] of boolean;
    subgr: array [0 .. physlimit] of graphtype;
    stackfl: file of graphtype;
    k: 0 .. physlimit;


Initialization (* main *)
    rewrite(stackfl);
    inmemory[0] ← true;
    for i := 1 to physlimit do
        inmemory[i] ← fasle;
```

Figure 32. Memory swapping in the backtracking scheme

```
Procedure node_to_process(m)
   var
      fp: integer; (* file pointer of stackfl *)
   begin
      if m = k + 1 then begin
         if k = physlimit then m ← 0 ;
         if inmemory [m] then
            write(stackfl, a[m])
         else inmemory[m] ← true ;
         a[m] ← a[k];
         k ← m;
      end else if m = k - 1 then begin
            inmemory[k] ← false;
            if k = 0 then k ← physlimit  else k ← m ;
            if not inmemory[k] then begin
               fp ← filepos(stackfl);
               fp ← fp - 1;
               seek(stackfl, fp);
               read(stackfl, a[k]) ;
               seek(stackfl, fp);
               inmemory[k] ← true;
            end;
         end else begin
            writeln('check m-value');
            halt;
         end;
   end;
```

Figure 33. Procedure node_to_process of memory swapping

C. HEAPSORT

In the frame of the branch-and-bound algorithm on the GCP mentioned in chapter 4, we employ *heapsort* to implement the priority queue *inbuf* and the sorted list *outbuf*. First we build a *heap* in *inbuf*, and then transform the nodes in the heap to the sorted list *outbuf*.

```
Procedure backward
   var
      fp: integer;
   begin
      fp ← filepos(stackfl);
      if (k = 0) and (fp = 0) then quit ← true
      else if periphery is empty then begin
         repeat
            inmemory[k] ← false;
            if k = 0 then k ← physlimit
            else k ← k – 1 ;
            if not inmemory[k] then begin
               fp ← fp – 1;
               seek(stackfl, fp);
               read(stackfl, a[k]) ;
               seek(stackfl, fp);
               inmemory[k] ← true;
            end;
         until a[k].ncv = q ;
         node_to_process(k – 1);
         update ncv;
         if there is no feasible color for ncv then
            backward;
      end
   end;
```

Figure 34. Procedure backward of memory swapping

Before describing the heapsort in the branch-and-bound algorithm, we would like to introduce the general concept of heapsort. A tree is called a *binary tree* if each node has *at most two* sons. The binary tree of depth d which has exactly $2^{d+1} - 1$ nodes is called a *full* binary tree. In order to describe a full binary tree by a sequential representation, we start from the root on level 0, then go to those on level 1, and so on. Nodes on each level are numbered from left to right. For example, Figure 35 is a full binary tree. The *complete tree* of order n can be denoted by the sequential representation of a full binary tree from 1 to n. It is seen that the leaves of a complete tree occur on at most two adjacent levels. A *heap* is a complete binary tree with the property, the value of each node is greater than or equal to that of its sons. A heap

having n nodes is denoted by an array H[1 .. n] in which H[1] is the root and the sons of H[i] are at H[2i] and H[2i+1]. There are two basic operations in heapsort. One is *heapup* which forms a bigger heap by inserting one additional node into an already existing heap. Another is *heapdown* which forms a smaller heap by taking the root away from an already existing heap. In Figure 36, procedure *heapup(n)* compares the input data which is in H[n] (assuming H[1] ... H[n − 1] have already formed a heap) with its father, grandfather, greatgrandfather, etc. until it is less than or equal to one of these values. In Figure 37, procedure *heapdown(n)* compares the input data which is in the root H[1] of H having a heap from H[2] to H[n] with its sons, grandsons, greatgrandsons, etc, until it is greater than or equal to one of these values. While comparing the target item in the position of H[i] with its sons, H[2i] and H[2i+1], we move the greater value of H[2i] and H[2i+1] up if either H[2i] or H[2i+1] is greater than the target item.

The *heap* is first built by repeatly calling the procedure *heapup* whenever a new element is added. Second it exchanges the largest element with the last element of heap (i.e. exchange(H[1], H[n])), and then applies procedure *heapdown(n − 1)*. We continue this cycle of process until the size of the heap becomes 1. The worst case time complexity of heapsort is $O(n \log n)$.

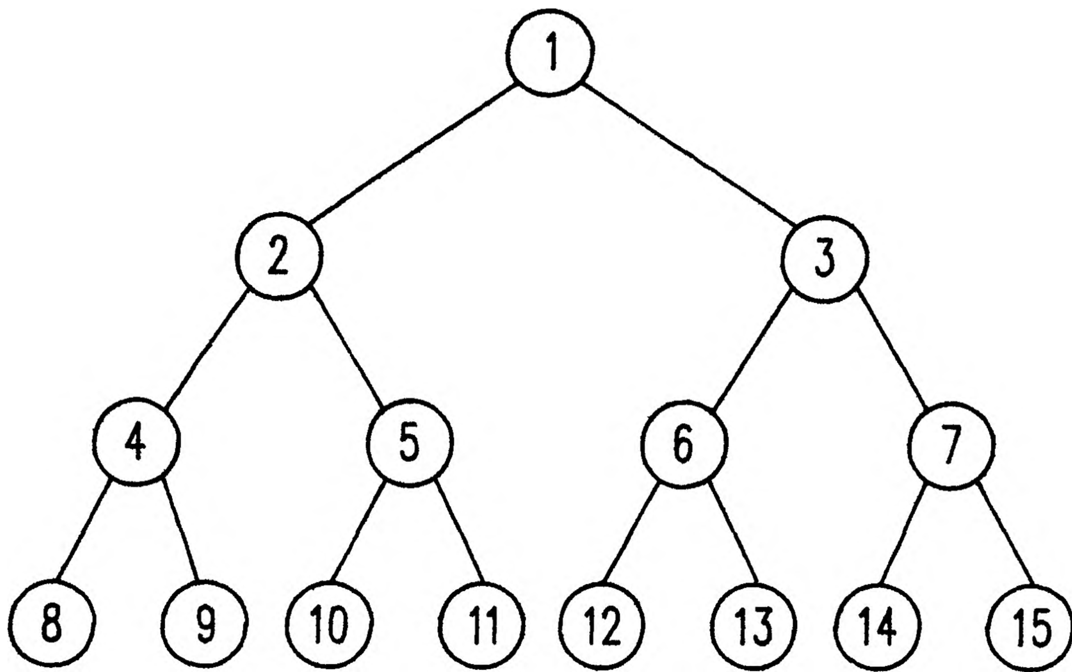


Figure 35. Example of full tree

```
Procedure heapup(n)
   begin
      target ← H[n];
      i ← n;
      ii ← i div 2 ;
      while (ii > 0) and (target > H[ii]) do
         begin
            H[i] ← H[ii]; (* move the father node down *)
            i ← ii;
            ii ← i div 2;
         end;
      H[i] ← target;
   end;
```

Figure 36. Procedure heapup

```
Procedure heapdown(n)
  begin
    target ← H[1];
    i ← 1;
    ii ← i * 2;
    if H[ii+1] > H[ii] then ii ← ii + 1 ;
    while ( ii ≤ n) and (target < H[ii]) do
      begin
        H[i] ← H[ii]; (* move the son node up *)
        i ← ii;
        ii ← i * 2;
        if H[ii+1] > H[ii] then ii ← ii + 1 ;
      end;
    H[i] ← target;
  end;
```

Figure 37. Procedure heapdown

```
(* sorting H[1], H[2], ... , H[N] to an non-decreasing order *)
Procedure heapsort(N)
  begin
    for i := 1 to N do (* build the heap *)
      heapup(i);
    for i := N downto 2 do
      begin
        exchange (H[1] , H[i]);
        heapdown(i - 1);
      end;
  end;
```

Figure 38. Procedure heapsort

Example 5-2 Figures 39-42 shows how to sort the date (7, 12, 2, 15, 4, 8, 10) into a non-decreasing order by heapsort algorithm.

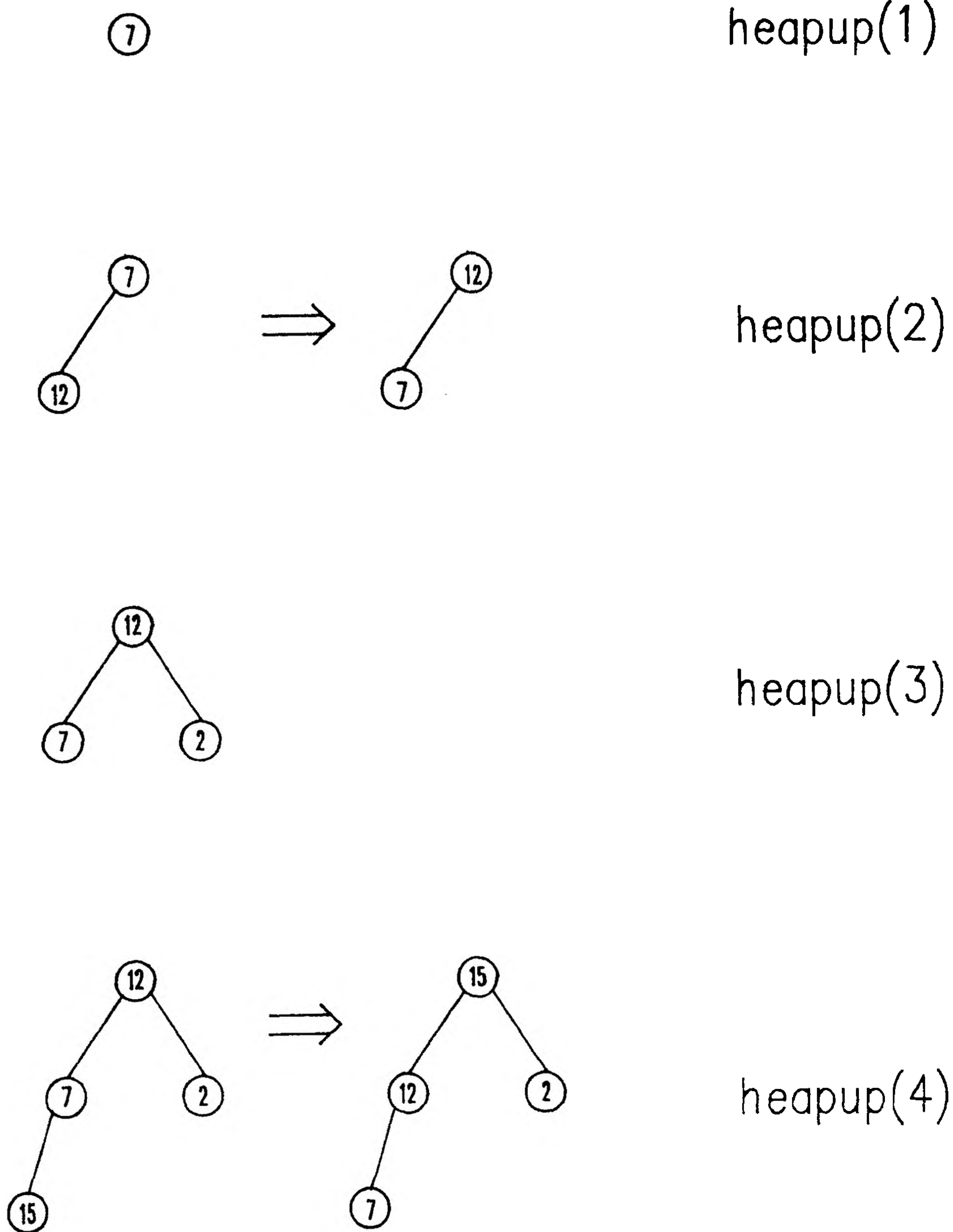
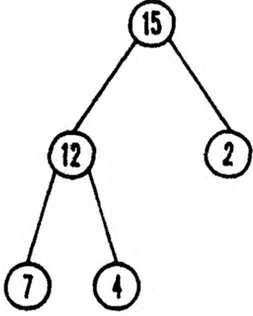
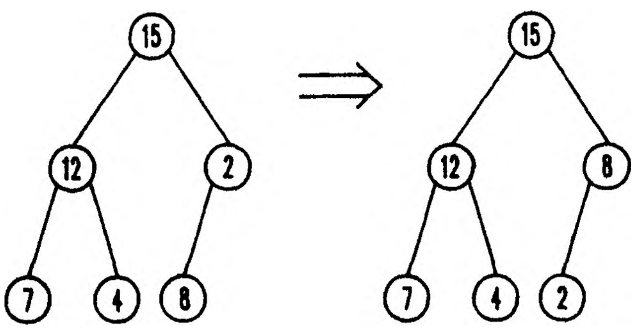


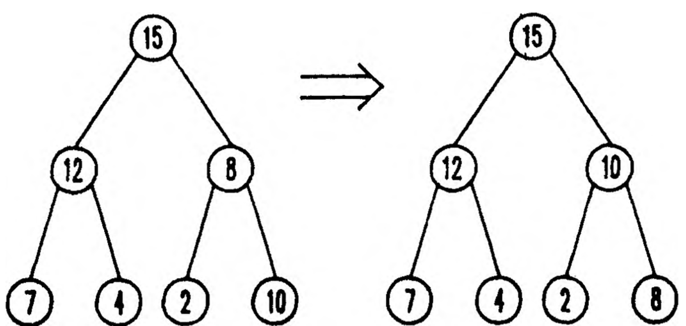
Figure 39. Part 1 of the example 5-2



heapup(5)

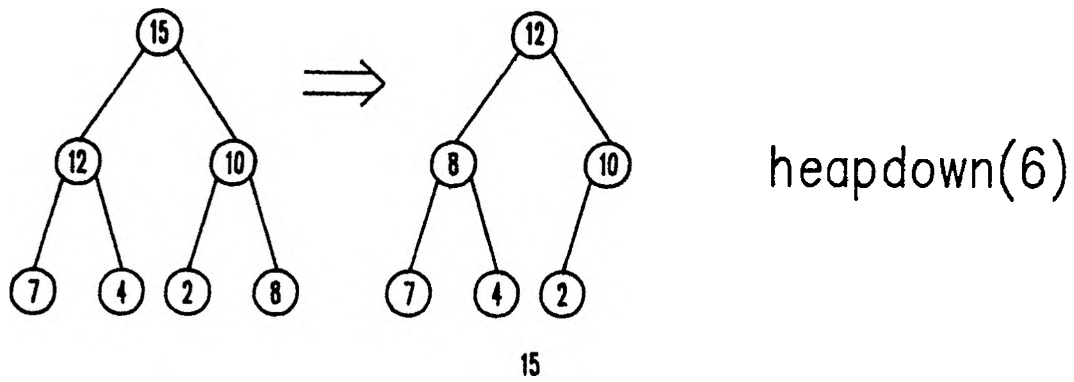


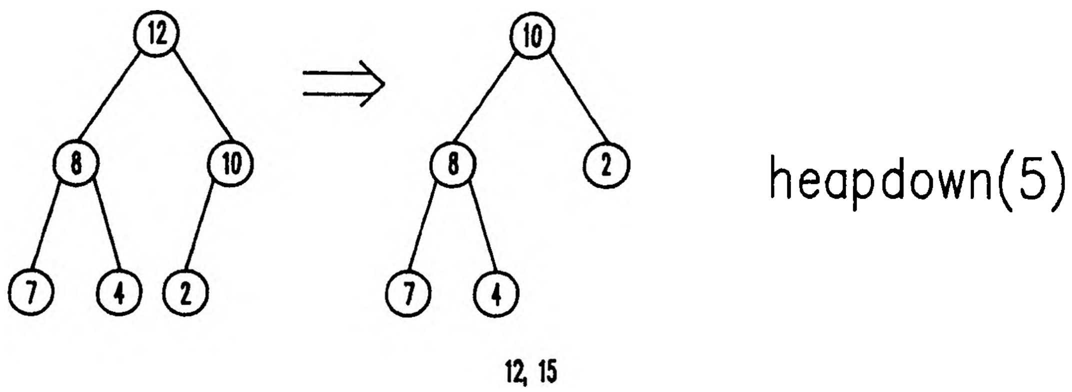
heapup(6)



heapup(7)

Figure 40. Part 2 of the example 5-2





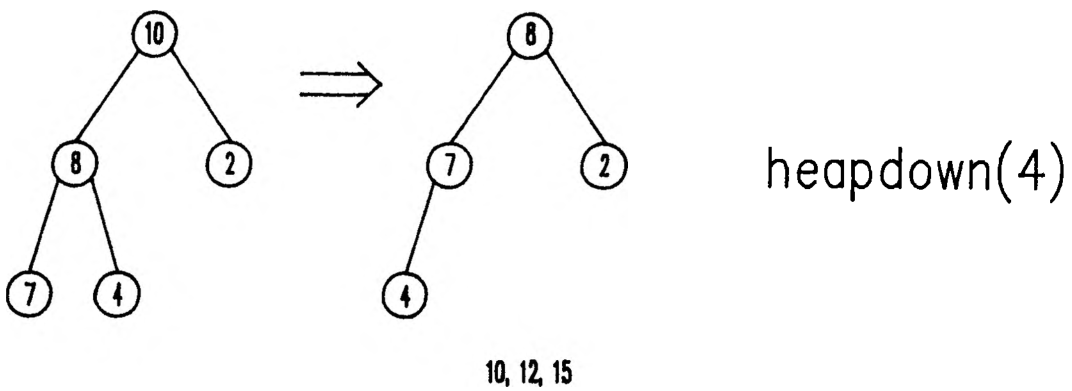
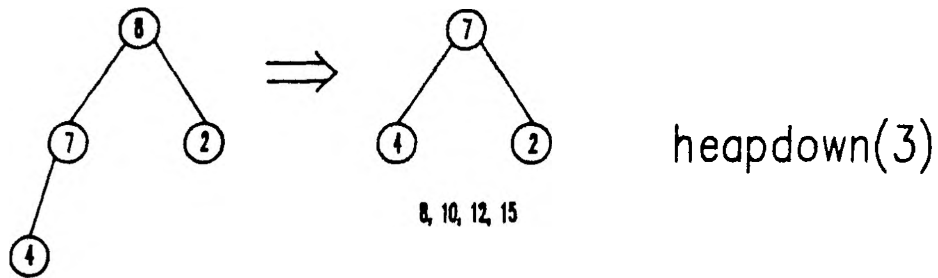
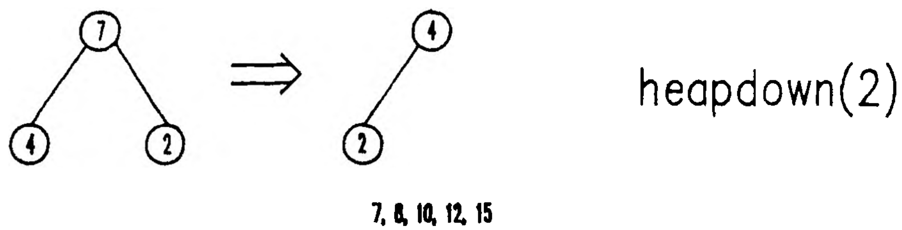
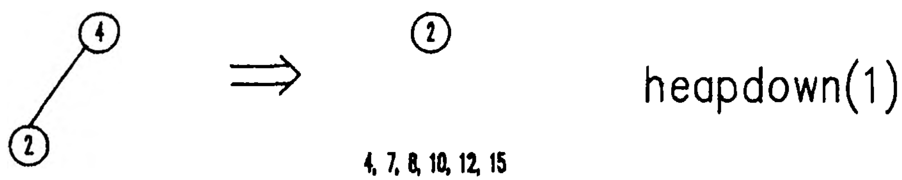


Figure 41. Part 3 of the example 5-2







Thus, the sorting order is 2, 4, 7, 8, 10, 12, 15

Figure 42. Part 4 of the example 5-2

In the branch-and-bound algorithm on the GCP, we use procedure ***heapup*** to build the heap *inbuf* and procedure ***heapdown*** (if necessary) to rebuild the new heap, which replaces the root of the heap by the added data, whenever the limit size of the heap is exceeded. Upon *outbuf* being empty, we repeatly do the transformation cycle to transform the heap *inbuf* into the sorted list *outbuf* in non-decreasing order. The transformation cycle is as follows: place the root of heap into the tail of *outbuf*, replace the root by the last element of *inbuf*, and call procedure ***heapdown*** to rebuild the heap on new *inbuf*, which removed the last element.

D. RANDOM NUMBER GENERATOR

A *multiplicative congruential generator* generates a sequence of integers within a specific interval by the generating functions of the form $f(x) = (a * x) \bmod m$, where a and m are two fixed integer parameters. a is called the *multiplier* of f(x). m is called the *modulus* of f(x). The sequence $x_1, x_2, x_3, \dots$ is generated via the iterative equation,

$$x_{i+1} = f(x_i) = (a * x_i) \bmod m \qquad \text{for } i = 1, 2, \dots .$$

x_1 is called the *seed* chosen from 1 .. $m-1$ before calling the iterative equation. Because of the deterministic characteristic of the iterative equation, the seed will guarantee to generate the *exact* same sequence for each run. Therefore, instead of storing a cycle of sequence, we only need to keep the seed in addition to the iterative equation for a sequence of reproducing work. It is seen that every element of the sequence $x_1, x_2, x_3, \dots$ is within the interval 0 .. $m-1$. In order to insure a period = $m-1$, a prime m is require. Consequently, the period of the sequence $x_1, x_2, x_3, \dots$ becomes 1 .. $m-1$. Usually a large prime modulus $m = 2^{31}-1$ is used to make the cycle of the sequence $x_1, x_2, x_3, \dots$ larger [*Le*51][*PM*88]. For a fixed modulus, in this case $m = 2^{31}-1$, a good multiplier requires the following three properties:

(1) $f(x) = (a * x) \bmod m$ is a full period generating function. That is, the generated sequence $x_1, x_2, \dots, x_{m-1}$ is a permutation of 1 .. $m-1$;

(2) The full period sequence $x_1, x_2, \ldots, x_{m-1}$ is pseudo random [*FM*86] ;

(3) $f(x) = (a * x) \bmod m$ can be implemented with 32-bit arithmetic.

The multiplier $a = 7^5 = 16807$ is suggested by Park and Miller [*PM*88]. The generator $f(x) = 16807x \bmod (2^{31} - 1)$ has a full period, and can be implemented on system having either 32-bit integer type or 32-bit (or larger) mantissa real type. The randomness of the generated sequence of $f(x) = 16807x \bmod (2^{31} - 1)$ is acceptable [*PM*88] although it is not within the optimal range set by Fishman and Moore [*FM*86].

The basic idea of 32-bit implementation of $f(x) = 16807 * x \bmod (2^{31} - 1)$ is to avoid the potential overflow associated with the term $16807 * x$. Our implementation is based on Schrage's method [*BF*87]. The theoretical details of Schrage's method are presented as follows:

The generator $f(x) = (a * x) \bmod m$

First, decomposing m such that

$$m = a * q + r \quad \text{where } q = m \text{ div } a,\ r = m \bmod a,\ r < q.$$

So

$$f(x) = (a * x) \bmod m$$
$$= a * x - m((a * x) \text{ div } m).$$

Subtracting and adding $m(x \text{ div } q)$

$$f(x) = a * x - m(x \text{ div } q) + m(x \text{ div } q) - m((a * x) \text{ div } m)$$
$$= a * x - (a * q + r)(x \text{ div } q) + m((x \text{ div } q) - ((a * x) \text{ div } m)).$$

Let $E(x) = (x \text{ div } q) - ((a * x) \text{ div } m)$

$$f(x) = a * x - (a * q + r)(x \text{ div } q) + m(E(x))$$
$$= a * x - (a * q)(x \text{ div } q) - r(x \text{ div } q) + m(E(x))$$
$$= a(x - q(x \text{ div } q)) - r(x \text{ div } q) + m(E(x))$$
$$= a(x \bmod q) - r(x \text{ div } q) + m(E(x)).$$

Let $R(x) = x \bmod q$, and $Q(x) = x \text{ div } q$.

$$f(x) = a(R(x)) - r(Q(x)) + m(E(x)).$$

Claim that for x in 1 .. m − 1 the followings are true:

(1) $0 \le a(R(x)) \le m - 1$, and $0 \le r(Q(x)) \le m - 1$;

(2) $|a(R(x)) - r(Q(x))| \le m - 1$;

(3) $E(x) \in \{0, 1\}$.

In (1), $0 \le a(R(x)) = a(x \bmod q) \le a * q < m$, since $r > 0$;

$0 \le r(Q(x)) = r(x \text{ div } q) < r(m \text{ div } q) = r * a < q * a < m$.

In (2), adding $0 \le a(R(x)) \le m - 1$ and $-(m-1) \le -r(Q(x)) \le 0$.

In (3), $\frac{x}{q} = (x \text{ div } q) + r_1$, where $0 < r_1 < 1$;

$\frac{a * x}{m} = ((a * x) \text{ div } m) + r_2$, where $0 < r_2 < 1$.

Consider

$$\frac{x}{q} - \frac{a * x}{m}$$
$$= \frac{x}{q} - \frac{a * x}{a * q + r}$$

$$= \frac{a * q * x + r * x - a * q * x}{q(a * q + r)}$$

$$= \frac{r * x}{q(a * q + r)} = \frac{r * x}{q * m}.$$

Since $0 < r < q$, and $0 < x < m$.

So $r * x < q * m$.

Thus $0 < \frac{x}{q} - \frac{a * x}{m} < 1$.

Substituting $\frac{x}{q}$ by $(x \text{ div } q) + r_1$, and

$\frac{a * x}{m}$ by $((a * x) \text{ div } m) + r_2$,

$0 < (x \text{ div } q) + r_1 - ((a * x) \text{ div } m) - r_2 < 1$;

$0 < (x \text{ div } q) - ((a * x) \text{ div } m) + (r_1 - r_2) < 1$.

Since $-1 < -(r_1 - r_2) < 1$,

$-1 < (x \text{ div } q) - ((a * x) \text{ div } m) < 2$;

$-1 < E(x) < 2$.

Since E(x) is an integer.

So $E(x) \in \{0, 1\}$.

In the equation $f(x) = a(R(x)) - r(Q(x)) + m(E(x))$, item (1) and item (2) prevent the intermediate results from the potential overflow on 32-bit arithmetic. Item (3) says that the computation on E(x) is not necessary.

$$f(x) = a(R(x)) - r(Q(x)) \quad \text{if } a(R(x)) - r(Q(x)) > 0 \;;$$
$$= a(R(x)) - r(Q(x)) + m \quad \text{otherwise.}$$

Figure 43 shows the Turbo Pascal implementation of $f(x) = (16807 * x) \bmod (2^{31} - 1)$ based on Schrage's method.

```
Global var:
   seed:  integer;
function random: real;   (* 0 < random < 1 *)
   const
      a = 16807;
      m = 2147483647;  (* 2^31 - 1 *)
      q = 127773;  (* m div a *)
      r = 2836;    (* m mod a *)
   var
      Qx, Rx, temp:  integer;

   begin
      Qx := seed div q;
      Rx := seed mod q;
      temp := a * Rx - r * Qx;
      if temp > 0 then
         seed := temp
      else seed := seed + m;
      random := seed / m;
   end;
```

Figure 43. A multiplicative congruential generator

VI. SELECTION FUNCTIONS

As we know from chapter 4 within the frame of either backtracking or branch-and-bound, the next-uncolored-vertex selection (nucv-selection) function and the next-color-vertex selection (ncv-selection) function are the crucial parts of these algorithms. In this chapter, a number of nucv-selection functions are introduced, and then some ncv-selection functions corresponding to a given nucv-selection function are presented. Our algorithms use the nucv-selection function first. The naming scheme of various nucv-selection ncv-selection functions is as follows:

(1) In the first character of a name, 'i' represents the class of nucv-selection functions, and 'j' represents the class of ncv-selection functions;

(2) In the last character of a name, 'a' indicates that a look-ahead procedure is used.

Experimental results are shown in order to determine which algorithms have high running speed, good solution quality, or preferable application area (small/large scale graphs as well as dense/sparse graphs).

A. TERMS

Definition: In a partially colored graph G = (V, E), an uncolored vertex v is called a *prevention vertex* of an uncolored vertex i on a colored vertex j if $(i, j) \notin E$, $(v, i) \in E$, and $(v, j) \notin E$, and the set of the prevention vertices of i on j is denoted by *p-set(i, j)*. Meanwhile, j is called a *feasible color* of i.

Definition: In a partially colored graph G = (V, E), an uncolored vertex v is called a *peer vertex* of an uncolored vertex i if $(i, v) \in E$, and the chrome-degree of v is equal to the chrome-degree of i. The set of the peer vertices of i is denoted by *peer-set(i)*.

Definition: In a partial coloring graph G = (V, E), an uncolored vertex v is called a *connection vertex* of an uncolored vertex i on a color vertex j if $(i, j) \notin E$, $(v, i) \in E$, and $(v, j) \in E$, and the set of connection vertices of i on j is denoted by *c-set(i, j)*.

Definition: In a partial coloring graph G = (V, E), a feasible color j of an uncolored vertex i is called a *block color* of i if j is the only feasible color for an adjacent uncolored vertex of i. An uncolored vertex v is called a *block vertex* if there is only one feasible color j for v, and j is a block color of v.

B. NEXT-UNCOLORED-VERTEX SELECTION FUNCTIONS

The nucv-selection function will choose the uncolored vertex *nucv* of the highest degree of preference from the periphery.

ikorman [*Ko*79] --- selects the nucv with the smallest feasible color set size. Ties are broken by choosing the vertex of greater degree. This selection strategy helps to reduce the size of the search tree. This movement takes O(n) time; n is the order of a graph.

ipkorman --- selects the nucv with the smallest feasible color set size. If there is more than one such vertex, it selects the one for which the cardinality of its peer-set is maximal. Ties are broken by choosing the one i whose score, $S(i) = (\sum_{av} \text{weight(chrome-degree of } av\text{)}$, over all av in the adjacent set of i), is maximal. For a fixed base b, the weight function w is defined as follow:

$$w(k) = b^k \qquad \text{if } b > 0\,;$$
$$= (k+1)^{-b} \qquad \text{otherwise.}$$

Ideally, the weight function is to select the nucv which is adjacent to a set of neighbors having particularly high chrome-degree. Note that S(i) is the number of adjacent neighbors of i if b = 0 or 1. This movement takes $O(n^2)$ time.

ipactual --- selects the nucv with the smallest feasible color set size. Ties are broken by choosing the one i whose score,

$$S(i) = (\sum_j(\sum_{pv}\text{weight(chrome-degree of pv), over all pv in p-set(i, j)), over all j in}$$
$$\text{the feasible color set of i),}$$

is maximal. This movement takes $O(n^3)$ time.

iprevent1a (with look-ahead) --- selects the nucv with the smallest feasible color set size. If there is more than one such vertex, it selects the one i for which the cardinality of (peer-set(i) $\cap$ p-set(i, j)) , for every feasible color j of i, is maximal. Ties are broken by choosing the one whose sum of the number of prevention vertices over every feasible color is maximal. The look-ahead procedure is done by eliminating block vertices from further consideration. This movement takes $O(n^3)$ time.

iprevent2a (with look-ahead) --- selects the nucv with the smallest feasible color set size. If there is more than one such vertex, it selects the one i for which the cardinality of (peer-set(i) $\cap$ p-set(i, j)) , for every feasible color j of i, is maximal. Ties are broken by choosing the one i whose score,

$$S(i) = (\sum_i(\sum_{pv}\text{weight(chrome-degree of pv), over all pv in p-set(i, j)), over all j in}$$
$$\text{the feasible color set of i),}$$

is maximal. This movement takes $O(n^3)$ time.

iprevent3a (with look-ahead) --- selects the nucv with the smallest feasible color set size. If there is more than one such vertex, it selects the one i for which the cardinality of (peer-set(i) $\cap$ p-set(i, j)) , for every feasible color j of i, is maximal. Ties are broken by choosing the one i whose score,

$$S(i) = (\sum_j(\sum_{pv}\text{((chrome-degree of pv) * (white-degree of pv)), over all pv in}$$
$$\text{p-set(i, j)), over all j in the feasible color set of i),}$$

is maximal. This movement takes $O(n^3)$ time.

iprevent4a (with look-ahead) --- selects the nucv with the smallest feasible color set size. If there is more than one such vertex, it selects the one i whose score,

$S(i) = (\sum_j (\sum_{pv}((\text{chrome-degree of pv}) * (\text{white-degree of pv})), \text{ over all pv in}$
$\text{p-set}(i, j)), \text{ over all j in the feasible color set of i}),$

is maximal. Ties are broken by choosing the one i for which the cardinality of (peer-set(i) $\cap$ p-set(i, j)) , for every feasible color j of i, is maximal. This movement takes $O(n^3)$ time.

iconnecta (with look-ahead) --- selects the nucv with the smallest feasible color set size. If there is more than one such vertex, it selects the one i for which the cardinality of (peer-set(i) $\cap$ p-set(i, j)) , for every feasible color j of i, is maximal. Ties are broken by choosing the one i whose score,

$S(i) = (\sum_j (\sum_{cv}((\text{chrome-degree of cv}) + (\text{white-degree of cv})), \text{ over all cv in}$
$\text{c-set}(i, j)), \text{ over all j in the feasible color set of i}),$

is maximal. This movement takes $O(n^3)$ time.

C. NEXT-COLORED-VERTEX SELECTION FUNCTIONS

The ncv-selection function will find the feasible color set for a vertex chosen by a nucv-selection function, and then order colors in the feasible color set according to a certain rule of preference. Note that in this section, we always let the new color be the last choice.

jkorman [*Ko*79]--- sorts the feasible color set of nucv according to the index of the color, and assigns the smallest color to ncv. This movement takes O(n) time.

jkormana (with look-ahead) --- is a look-ahead version of *jkorman*. The look-ahead procedure is to prevent the block color of the nucv from being in the feasible color set. This movement takes $O(n^2)$ time.

jsucadja (with look-ahead) --- sorts the feasible color set according to the following rules:

(a) $\min_j$ {white-degree of j}, where j is a feasible color.

If colors are left unsorted by (a), then apply rule (b):

(b) $\min_j \{j\}$.

This movement takes $O(n^2)$ time.

jpactual --- sorts the feasible color set according to the following rules:

(a) $\min_j$ {($\sum_{pv}$ weight(chrome-degree of pv), over all pv in p-set(nucv, j))},

where j is a feasible color.

If colors are left unsorted by (a), then apply rule (b):

(b) $\min_j$ {white-degree of j}.

Finally, if (a) and (b) do not distinguish between feasible colors, use (c):

(c) $\min_j \{j\}$.

This movement takes $O(n^2)$ time.

jpactuala (with look-ahead) --- is a look-ahead version of the *jpactual.* This movement takes $O(n^2)$ time.

jprevent1a (with look-ahead)--- sorts the feasible color set according to the following rules:

(a) $\min_j$ {|peer-set(nucv) ∩ p-set(nucv, j)|}, where j is a feasible color.

If colors are left unsorted by (a), then apply rule (b):

(b) $\min_j$ {|p-set(nucv, j)|}.

Finally, if (a) and (b) do not distinguish between colors, use (c)

(c) $\min_j \{j\}$.

This movement takes $O(n^2)$ time.

jprevent2a (with look-ahead)--- sorts the feasible color set according to the following rules:

(a) $\min_j\{|\text{peer-set(nucv)} \cap \text{p-set(nucv, j)}|\}$, where j is a feasible color.

If colors are left unsorted by (a), then apply rule (b):

(b) $\min_j\{(\sum_{pv}\text{weight(chrome-degree of pv)}$, over all pv in p-set(nucv, j))},
where j is a feasible color.

Finally, if (a) and (b) do not distinguish between feasible colors, use (c):

(c) $\min_j\{j\}$.

This movement takes $O(n^2)$ time.

jprevent3a (with look-ahead)--- sorts the feasible color set according to the following rules:

(a) $\min_j\{|\text{peer-set(nucv)} \cap \text{p-set(nucv, j)}|\}$, where j is a feasible color.

If colors are left unsorted by (a), then apply rule (b):

(b) $\min_j\{(\sum_{pv}((\text{chrome-degree of pv}) * (\text{white-degree of pv}))$, over all pv in p-set(nucv, j))}, where j is a feasible color.

Finally, if (a) and (b) do not distinguish between feasible colors, use (c):

(c) $\min_j\{j\}$.

This movement takes $O(n^2)$ time.

jprevent4a (with look-ahead)--- sorts the feasible color set according to the following rules:

(a) $\min_j\{(\sum_{pv}((\text{chrome-degree of pv}) * (\text{white-degree of pv}))$, over all pv in p-set(nucv, j))}, where j is a feasible color.

If colors are left unsorted by (a), then apply rule (b):

(b) $\min_j\{|\text{peer-set(nucv)} \cap \text{p-set(nucv, j)}|\}$.

Finally, if (a) and (b) do not distinguish between feasible colors, use (c):

(c) $\min_j\{j\}$.

This movement takes $O(n^2)$ time.

jconnecta (with look-ahead)--- sorts the feasible color set according to the following rules:

(a) $\min_j\{|\text{peer-set(nucv)} \cap \text{p-set(nucv, j)}|\}$, where j is a feasible color.

If colors are left unsorted by (a), then apply rule (b):

(b) $\min_j\{(\sum_{cv}((\text{chrome-degree of cv}) + (\text{white-degree of cv})), \text{over all cv in}$ $\text{c-set(nucv, j))}\}$, where j is a feasible color.

Finally, if (a) and (b) do not distinguish between feasible colors, use (c):

(c) $\min_j\{j\}$.

This movement takes $O(n^2)$ time.

D. WEIGHTED SCALE ON THE SCORE OF THE NEW COLOR

In the last section, the new color is always the last choice. However, this kind of arrangement may require more backward and forward moves for some graph. In this section we attempted to beat this problem in the average sense by employing a weighted scale on the score of the new color.

In Tables XVII-XXIV, the score of the new color is equal to the score, which is computed according to the formula in the rule (a) of algorithm *jpactual* or algorithm *jprevent4a* , with the factor of the given weight. The results in Tables XVII-XX support the claim that the method putting weighted scale on the score of the new color fails to significantly improve the performance of coloring in the global sense.

E. SWAPPING BETWEEN THE CORE AND THE PERIPHERY

With regard to the computational time, we find that the algorithm Korman is simple but fast because of its linear time complexity for each forward movement. The

algorithm Korman can be improved by swapping between the core and the periphery. Let us assume that c_0, c_1 ... , c_k form the core, and v_{k+1}, v_{k+2} ... , v_m form the periphery. If there exists a vertex v_i, among v_{k+1} ... , v_m, which is connected to all colors in the core but c_j, the pair (c_j, v_i) is said to be 1-1 *swappable*. In this case, we move v_i to the core and c_j to the periphery. If there exist two adjacent vertices v_{i1}, v_{i2} in the periphery such that each of them adjacents to all colors in the core but c_j, the triple (c_j, v_{i1}, v_{i2}) is said to be 2-1 *swappable*. In this case, we move v_{i1}, v_{i2} to the core and c_j to the periphery. It is evident that the 2-1 swapping will introduce a new color to the core. Similarly, if there exist three mutually connected vertices v_{i1}, v_{i2}, and v_{i3} in the periphery such that each of them adjacent to all colors in the core but c_{j1} and c_{j2}, the 5-tuple (c_{j1}, c_{j2}, v_{i1}, v_{i2}, v_{i3}) is said to be 3-2 *swappable*. In this case, we move v_{i1}, v_{i2}, v_{i3}, the the core and c_{j1}, c_{j2} to the periphery. The 3-2 swapping also adds a new color to the core. The swapping method iterates the cycle: finding a swappable candidate and performing (if necessary) the swapping process until there is no swappable candidate. The vertex sequential coloring algorithm with swapping is as follows: (1) if there is an vertex v which is adjacent to all existing color , color it with new color; (2) else search for 2-1 swappable, do swapping if found; (3) else use the vertex sequential coloring algorithm. the swapping method is done before a new forward movement. The algorithms in this section differ only in the swapping method from the corresponding algorithms mentioned in section B.

ikorqk2 --- is the *ikorman* algorithm with the swapping method which searchs for a 2-1 swappable candidate as early as possible and does the swapping process. This movement takes $O(n^3)$.

ikorpw2 --- is the *ikorman* algorithm with the swapping method which searchs for all 2-1 swappable triples (c_j, v_{i1}, v_{i2}). If there is more than one candidate, it takes the

2-1 swappable candidate whose value (white-degree(v_{i1}) + white-degree(v_{i2})) is maximal. This movement takes $O(n^3)$.

ikorw2 --- is the *ikorman* algorithm with the swapping method which first searchs for all 2-1 swappable triples (c_j, v_{i1}, v_{i2}). If there is a candidate, it takes the 2-1 swappable candidate whose value (white-degree(v_{i1}) + white-degree(v_{i2}) − white-degree(c_j)) is maximal. This movement takes $O(n^3)$.

ikorw2p --- is similar to *ikorw2* except that all 2-1 swappable triples (c_j, v_{i1}, v_{i2}) in the first part of the swapping method satisfy the condition (white-degree(v_{i1}) + white-degree(v_{i2}) − white-degree(c_j)) > 0.

ikormaxw2 --- is the *ikorman* algorithm with the following swapping method. The swapping method searchs for all 2-1 swappable triples (c_j, v_{i1}, v_{i2}). If there is a candidate, it takes the 2-1 swappable candidate whose value max{white-degree(v_{i1}), white-degree(v_{i2})} is maximal. This movement takes $O(n^3)$ time.

ikorw23 --- is the *ikorman* algorithm with the swapping method. The swapping method is as follows:

step 1: search for all 2-1 swappable triples. If there is no candidate, go to step 3.

step 2: take the candidate (c_j, v_{i1}, v_{i2}) whose value (white-degree(v_{i1}) + white-degree(v_{i2}) − white-degree(c_j)) is maximal, and perform the swapping process. Goto step 1.

step 3: search for all 3-2 swappable 5-tuples. If there is no candidate, then go to step 5.

step 4: take the candidate (c_{j1}, c_{j2}, v_{i1}, v_{i2}, v_{i3}) whose value (white-degree(v_{i1}) + white-degree(v_{i2}) + white-degree(v_{i3}) − white-degree(c_{j1}) − white-degree(c_{j2})) is maximal, and perform the swapping process. Goto step 1.

step 5: begin the *ikorman* selection.

This movement takes $O(n^4)$ time.

ikorqk23 --- is similar to *ikorw23* except that instead of searching for all possible swapping triples in order to select the best one, it picks the swapping candidate as soon as it appears.

ikorw2e (scale) --- is analogous to *ikorw2* except that all 2-1 swappable condidates (c_j, v_{i1}, v_{i2}) must satisfy the following condition: wscore > threshold, where wscore = white-degree(v_{i1}) + white-degree(v_{i2}) − white-degree(c_j), and threshold = lower_bound(wscore) + (upper_bound(wscore) − lower_bound(wscore)) * scale. Note that *ikorw2e* with scale = 0 is equivalent to *ikorw2.* This movement takes $O(n^3)$ time. Default scale = 0.0.

ikorw21e (scale1, scale2) --- is the *ikorman* algorithm with the following swapping method:

step 1: search for all 2-1 swappable triples (c_j, v_{i1}, v_{i2}) satisfying the condition: wscore2 > threshold2, where wscore2 = white-degree(v_{i1}) + white-degree(v_{i2}) − white-degree(c_j), and threshold2 = lower_bound(wscore2) + (upper_bound(wscore2) − lower_bound(wscore2)) * scale2. If there is no candidate, go to step 3.

step 2: take the candidate with maximal score on wscore2, and perform the swapping process. Goto step 1.

step 3: search for all 1-1 swappable pairs (c_j, v_i) satisfying the condition: wscore1 > threshold1, where wscore1 = white-degree(v_i), − white-degree(c_j), and threshold1 = lower_bound(wscore1) + (upper_bound(wscore1) − lower_bound(wscore1)) * scale1. If there is no candidate, then go to step 5.

step 4: take the candidate with maximal score on wscore1, and perform the swapping process. Goto step 1.

step 5: begin the *ikorman* selection.

Default scale1 = 0.5, and scale2 = 0.0.

ikorw12e (scale1, scale2) --- instead of searching for 2-1 swappable candidate first in the algorithm *ikorw21e*, the algorithm *ikorw12e* search for 1-1 swappable candidate first and then 2-1 swappable candidate.

ikorw21ec (scale1, scale2) --- Algorithm *ikorw21e* always does 2-1 swapping first and then 1-1 swapping (if there is no 2-1 swappable candidate). Another algorithm *ikorw12e* does the swapping method by giving 1-1 swapping a higher priority (than 2-1 swapping). In the global sense, *ikorw21ec* searchs for all 1-1 swappable pairs and 2-1 swappable triples. Meanwhile, the score of each candidate is evaluated (see *ikorw21e*). The swapping process takes the candidate which has the maximal score among 1-1 and 2-1 swappable candidates.

ikormaxw21e (scale) --- is similar to *ikorw21e* except replacing step 1, 2 in *ikorw21e* by the *ikormaxw2* algorithm. Default scale = 0.5.

ipactqk2 --- is the *ipactual* algorithm with the swapping method which searchs for a 2-1 swappable candidate as early as possible and perform the swapping process. This movement takes $O(n^3)$.

ipactmaxw2 --- is the *ipactual* algorithm with the swapping method. The swapping method searchs for all 2-1 swappable triples (c_j, v_{i1}, v_{i2}). If there is a candidate, it takes the 2-1 swappable candidate with maximal score on max{white-degree(v_{i1}), white-degree(v_{i2})}. This movement takes $O(n^3)$ time.

F. HEURISTIC ALGORITHMS

Recall that the ncv-selection function keeps all feasible colors of nucv in a certain order according to the user-defined preference. The heuristic algorithm can by done by pruning some feasible colors of lower preference. From the experimental results, in about 95% of all random graphs, using our algorithms, the number of branches of every node in the backtrack-tree is either 1 or 2. We introduce two straightforwards but effective pruning techniques.

1. Limit.

Assume that L, $1 \leq L \leq 2$, branches of every node are expected to be left after pruning, and b denotes the number of branches of a node. The *limit* pruning technique is as follow: If $b = 1$, then it does nothing; otherwise the second branch of lower preference is chosen with probability L -1.

2. Epsilon.

The *epsilon* prunning technique is to set a threshold for the user-defined preference. Any branch with degree of preference less than threshold is eliminated. Assume that the full range between the lower bound of potential degree of preference and the upper bound of potential degree of preference is treated as one unit. The *threshold* is defined to be the lower bound plus the product of *eps*, where $0 \leq eps \leq 1$, and the absolute difference between the upper bound and lower bound .

G. COMPUTATIONAL RESULTS

The various coloring algorithms under the backtracking scheme (refer to chapter 4) were applied to an identical sequence of random graphs, and the results were tabulated in order to compare their speed. The random graphs were generated utilizing the pseudo random number generator according to two parameters, the order of a random graph and the edgeload. The detailed description of the pseudo random number generator has been discussed in chapter 5. The edgeload of a random graph is the ratio of the number of actual edges to the number of potential edges. For each constant order of a random graph, four different classes of random graphs were produced with edgeloads: 0.3, 0.5, 0.7, and 0.9. For each order and edgeload, 100 random graphs were generated. Every figure in the tables is the mean of 100 graphs. The "exact color" column is the chromatic number, the "first color" column is the number of colors required for the first complete coloring of the backtrack tree, "moves" column is the number of forward moves, "S.D." column is the standard deviation of forward moves, and "var cof." column is the variance coefficient which is the ratio of standard deviation to mean of forward moves.

Tables I-IV show that Pkorman algorithms with weight other than 0 and 1 run faster than Pkorman with weight 0 or 1 except the situation, vertices = 40 and edgeload = 0.7. Tables V-VIII display that weight other than 0 and 1 in the Pactual algorithm performs better in speed than weight = 0 or 1 even in the situation, edgeload = 0.7. From Tables I-VIII, we know that finding the *best* weight other than 0 and 1 according to the computational time is not possible. However, we observe that *weight = 2* has a relatively good performance (especially for edgeload = 0.7).

There is no apparent relation between "first colors" and "moves". An algorithm with fewer "first colors" may take more forward moves. Tables IX-XVI show that a fixed nucv-selection function with various ncv-selection functions produce algorithms

with "moves" which are within 0-7% of the mean for the corresponding set of forward moves. On the other hand, a fixed ncv-selection function with various nucv-selection functions generate widely different numbers of forward moves; for example, *(ikorman, jkorman)* and *(ipactual, jkorman)*.

Tables XXXVII-XLVI contain the computational results of a number of exact coloring algorithms. The *variance coefficient* is the ratio of standard deviation to mean. When observing the following algorithms: *(ikorman, jkorman)*, *(ikorman, jkormana)*, *(ipactual, jpactual)* and *(ipactual, jpactuala)*, we find that adding the look-ahead procedure to the backtracking sequential algorithm yields significant (9% - 15%) improvement on the forward movement except the case, edgeload = 0.9. The linear Korman selection is nearly the fastest algorithm (Pactual is shown to be faster than Korman in Table XLIV). The Connecta algorithm is the slowest one. Most of the other algorithms produce a significant cut on the forward movement. However, the additional computation time for the search tree pruning is also significant. For example, for the algorithm *(ipkorman, jkorman)*, the additional computation time required for the prior work of tree pruning is more than the time saved by tree pruning over algorithm *ikorman*. The algorithm *(ikorman, jsucadja)* generates better "first colors" than other algorithms.

Tables XVII-XX show that raising the preference of the new color increases both the forward moves and the colors required for the first complete coloring except for edgeload = 0.5, the Pactual algorithm with w = 0.6 has slightly fewer (about 1.5%) forward moves. Similar situations occur both for edgeload = 0.5 and w = 0.4 in Table XXII and for edgeload = 0.7 and w = 0.3 or 0.4 in Table XXIII.

Tables XXV-XXVIII show that *ikorw21e* and *ikorw21ec* are slightly better than *ikorw12e* , and the performance of *ikorw21e* is nearly similar to that of *ikorw21ec*.

Tables XXIX-XXXII show that (1) the case, scale2 = 0.0, (full 2-1 swapping) takes fewer forward moves than the case, scale2 = 0.0 and scale1 = 0.5 (full 2-1 swapping plus full 1-1 swapping); (2) the case, scale2 = 1.0 and scale1 = 0.5 (full 1-1 swapping), takes fewer forward moves than the case, scale2 = 1.0 (without 2-1 swapping); (3) the full 2-1 swapping takes fewer forward moves than the partial 2-1 swapping (scale2 $\neq$ 0.0); it seems that the wider the range of 2-1 swapping is, the less the forward moves are; (4) the full 1-1 swapping only behaves better than the Korman algorithm in "moves" and "first color"; and (5) the full 2-1 swapping without 1-1 swapping takes fewer forward moves than other algorithms.

Tables XXXIII-XXXVI show that (1) *ikorw2* has slightly better performance than either *ikorpw2* or *ikorw2p* ; (2) adding 3-2 swapping process to either *ikorqk2* or *ikorw2* creates a significant improvement on the "moves" (about 19% for edgeload = 0.3, about 27% for edgeload = 0.5, about 21% for edgeload = 0.7, and about 10% for edgeload = 0.9). Both *ikorqk23* and *ikorw23* , however, pay a significant overhead in time (about 50% for edgeload = 0.3, about 108% for edgeload = 0.5, about 260% for edgeload = 0.7, and about 450% for edgeload = 0.9); (3) the Korman algorithm with 2-1 swapping or its variations take fewer forward moves than the Korman algorithm and needs fewer "first color" except the case, edgeload = 0.9; (4) the Pactual algorithm with 2-1 swapping has better performance in "moves" and "first color" than the Pactual algorithm; and (5) the algorithms, *(ipactqk2, jpactual)* and *(ipactmaxw2, jpactual)*, need fewer "first color" than other algorithms.

Tables XXXVII-XL, XLI-XLII, XLIII-XLVI, and XLVII-XLVIII show that (1) both *ikorw2* and *ikormaxw2* makes a substantial improvement on the forward moves (about 25% better than Korman) and the computational time (about 10% better than Korman for edgeload = 0.5 and 0.7); and (2) *(ipactmaxw2, jpactual)* behaves

exceptionally better (about 50%) than Korman on the forward moves and colors a graph using fewer colors than Korman.

In Figures 44-59, Korman algorithm represents *(ikorman, jkorman)* algorithm, Korw2 algorithm represents *(ikorw2, jkorman)* algorithm, Pactual algorithm represents *(ipactual, jpactual)* algorithm, and Pactmaxw2 algorithm stands for *(ipactmaxw2, jpactual)* algorithm. Every continuous curve in Figures 44-60 is developed by (1) computing the data points associating with $|V|$ = 28, 32, 36, 40, 44, 48, 52, and 56; and (2) using the cubic spline function to fit a curve to the data points. Figures 44-47 show that (1) Chrome represents the exact colors; (2) Pactmaxw2 algorithm with non-backtracking makes use of fewer heuristic colors than Korman, Korw2, and Pactual for 100 identical random graphs; and (3) every curve is nearly linear. Figures 52-55 show that (1) Pactmaxw2 takes fewer forward moves than Pactual, Pactual takes fewer forward moves than Korw2, and Korw2 takes fewer forward moves than Korman; (2) the conjecture, bigger graphs requires more forward moves to get the chromatic number, is not always true; for example, $|V|$ = 52 and 56 for edgeload = 0.3 and $|V|$ = 40 and 44 for edgeload = 0.3; (3) all curves for edgeload = 0.5, 0.7, and 0.9 are exponential, and they go up sharply after passing $|V|$ = 44; and (4) for edgeload = 0.5, 0.7 and 0.9, the gap between Pactmaxw2 and Korman becomes bigger as the number of vertices becomes larger. Figures 56-59 have the same data points as Figures 52-55, plotted on a *logarithmic* scale on the *forward moves* axis. Figures 48-51 show that (1) Korw2 is faster than Korman; (2) Pactmaxw2 is faster than Pactual; and (3) for edgeload = 0.3, 0.5, 0.7, and 0.9, all curves are exponential.

In Tables LI-LII display that (1) "mean" column is the mean of differences of running time between Korw2 and Korman; (2) "S.D." column is the standard deviation of the differences of running time; (3) "better" column is the percentage of graphs in which Korw2 is faster Korman; (4) "worse" column is the percentage of graphs in

which Korw2 is slower than Korman; and (5) Korw2 is faster than Korman on the average. Tables LIII-LIV show that (1) Pactual is slower than Korman for most of graphs except that n = 56 on edgeload = 0.5 and 0.7, and n = 52 on edgeload = 0.7. Tables LV-LVI show that (1) for edgeload = 0.3, Pactmaxw2 is slower than Korman for most of graphs except n = 52 and 56; (2) for edgeload = 0.5 and 0.7, Pactmaxw2 is faster than Korman for most of graphs except n = 28, 32, and 36; and (3) for edgeload = 0.9, Pactmaxw2 is slower than Korman for most of graphs. Tables LVII-LVIII show that Pactmaxw2 is faster than Pactual on the average and for most of graphs.

In Tables LIX-LXX, the equation, c-ap = q, means that q of the 100 graphs can be colored with (c + chromatic number) colors by the *limit* pruning technique. The *jkorlm, jpaclm,* and *jpv4lm* are the modified version of *jkorman, jpactual,* and *jprevent4a* respectively. In Tables LXVII-LXX, *iprv4a* is a shorthand of *iprevent4a.* The "heur. color" column is the mean of the heuristic colors of 100 random graphs. Tables LIX and LXII show that for edgeload = 0.3 and edgeload = 0.9, the 0-ap decreases slowly as the *eps* becomes smaller. On the other hand, for the *limit* pruning technique, the 0-ap may not decrease as the *lim* becomes smaller, such as *lim* = 1.6 and 1.7 and edgeload = 0.9 in Table LXVI, because of using the probability for picking the second branch of every node of the backtrack tree.

H. DISCUSSION

The weight function shown in *pkorman* is helpful for deciding the next uncolored vertex in the average sense. A nucv-selection function with various ncv-selection functions generate the number of forward moves which are within 7% of the mean of its corresponding set. However, a ncv-selection with various nucv-selection functions have widely different performances. We suspect that the reason is that, the

nucv-selection function is called first. The look-ahead procedure either in the nucv-selection function or the ncv-selection function has a significant improvement on the forward moves and the running time.

From the worst case analysis, the Korman algorithm is O(n), the Pkorman algorithm is $O(n^2)$, and Pactual, Prevent1a, Prevent2a, Prevent3a, Prevent4a, and Connecta are $O(n^3)$. However, from experimental results, the Pkorman algorithm is inferior to algorithms, in $O(n^3)$ time complexity, except Connecta in the average sense. The variations of the Korman algorithm, Pactual, Prevent1a, Prevent2a, Prevent3a, and Prevent4a, prune the backtrack tree effectively. However, the computational time for choosing a good forward movement is also significant.

Although the 3-2 swapping method makes fewer forward moves, its overhead for searching for a 3-2 swapping candidate is significant (especially for dense graphs). The Korman algorithm with either 2-1 swapping or 2-1 swapping plus 1-1 swapping is superior to the Korman algorithm on the forward moves and the running time. The Pactual algorithm with 2-1 swapping is superior to the Korman algorithm on the forward moves and the running time. The Pactual algorithm with 2-1 swapping generates the smallest "first color" among Korman's algorithm and the algorithms which have been developed. That is, it is a good heuristic algorithm of the vertex-color sequential without backtracking type.

In Figure 52, all curves go down between $|V| = 52$ and 56. We suspect that it is only a local action. From the global sense, the curves are still exponential.

In our algorithms, the *top two* feasible colors, of higher degree of preference, of a nucv chosen by using the nucv-selection function almost always (within 95% up to 52 vertices) yield an optimal coloring. Two heuristic algorithms, *limit* and *epsilon* , based on the above fact are presented.

Table I. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Pkorman algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ikorman	jkorman	-	5.96	183.86	6.40	2
ipkorman	jkorman	0		153.77	6.21	3
ipkorman	jkorman	1		153.77	6.21	3
ipkorman	jkorman	2		147.43	6.23	3
ipkorman	jkorman	3		148.97	6.24	3
ipkorman	jkorman	4		143.62	6.25	3
ipkorman	jkorman	5		144.77	6.24	3
ipkorman	jkorman	6		143.79	6.25	3
ipkorman	jkorman	7		143.98	6.25	3
ipkorman	jkorman	8		143.94	6.25	3
ipkorman	jkorman	9		143.90	6.25	3
ipkorman	jkorman	-1		153.71	6.22	3
ipkorman	jkorman	-2		143.58	6.24	3
ipkorman	jkorman	-3		143.59	6.24	3
ipkorman	jkorman	-4		144.24	6.26	3
ipkorman	jkorman	-5		144.18	6.26	3
ipkorman	jkorman	-6		143.83	6.25	3
ipkorman	jkorman	-7		143.88	6.25	3
ipkorman	jkorman	-8		143.95	6.25	3
ipkorman	jkorman	-9		143.93	6.25	3

Table II. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Pkorman algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ikorman	jkorman	-	8.23	1108.45	9.44	15
ipkorman	jkorman	0		1209.46	9.36	24
ipkorman	jkorman	1		1209.46	9.36	24
ipkorman	jkorman	2		1076.45	9.42	21
ipkorman	jkorman	3		1026.46	9.40	20
ipkorman	jkorman	4		1012.69	9.35	20
ipkorman	jkorman	5		1007.52	9.34	20
ipkorman	jkorman	6		1007.25	9.34	20
ipkorman	jkorman	7		1007.21	9.33	20
ipkorman	jkorman	8		1008.18	9.34	20
ipkorman	jkorman	9		1008.41	9.34	20
ipkorman	jkorman	-1		1163.14	9.46	23
ipkorman	jkorman	-2		1094.72	9.44	22
ipkorman	jkorman	-3		1036.52	9.42	20
ipkorman	jkorman	-4		1016.94	9.37	20
ipkorman	jkorman	-5		1018.65	9.38	20
ipkorman	jkorman	-6		1019.91	9.39	20
ipkorman	jkorman	-7		1006.79	9.35	20
ipkorman	jkorman	-8		1003.70	9.33	20
ipkorman	jkorman	-9		1006.82	9.33	20

Table III. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Pkorman algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ikorman	jkorman	-	11.88	833.55	13.14	11
ipkorman	jkorman	0		979.34	13.11	20
ipkorman	jkorman	1		979.34	13.11	20
ipkorman	jkorman	2		880.44	13.22	18
ipkorman	jkorman	3		1009.44	13.22	20
ipkorman	jkorman	4		981.67	13.28	19
ipkorman	jkorman	5		961.39	13.23	19
ipkorman	jkorman	6		967.88	13.24	19
ipkorman	jkorman	7		970.87	13.23	19
ipkorman	jkorman	8		971.03	13.23	19
ipkorman	jkorman	9		972.87	13.24	19
ipkorman	jkorman	-1		923.28	13.21	19
ipkorman	jkorman	-2		884.06	13.20	18
ipkorman	jkorman	-3		963.17	13.21	19
ipkorman	jkorman	-4		981.06	13.22	20
ipkorman	jkorman	-5		899.73	13.20	18
ipkorman	jkorman	-6		965.37	13.21	19
ipkorman	jkorman	-7		984.18	13.21	20
ipkorman	jkorman	-8		1031.85	13.22	20
ipkorman	jkorman	-9		963.04	13.23	19

Table IV. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Pkorman algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ikorman	jkorman	-	19.20	78.83	19.53	1
ipkorman	jkorman	0		64.03	19.59	3
ipkorman	jkorman	1		64.03	19.59	3
ipkorman	jkorman	2		62.88	19.62	3
ipkorman	jkorman	3		62.46	19.61	3
ipkorman	jkorman	4		63.45	19.66	3
ipkorman	jkorman	5		63.54	19.67	3
ipkorman	jkorman	6		63.83	19.65	3
ipkorman	jkorman	7		61.76	19.61	3
ipkorman	jkorman	8		64.01	19.67	3
ipkorman	jkorman	9		62.99	19.66	3
ipkorman	jkorman	-1		63.05	19.58	3
ipkorman	jkorman	-2		62.72	19.60	3
ipkorman	jkorman	-3		63.34	19.59	3
ipkorman	jkorman	-4		62.05	19.59	3
ipkorman	jkorman	-5		62.10	19.56	3
ipkorman	jkorman	-6		62.83	19.61	3
ipkorman	jkorman	-7		63.32	19.62	3
ipkorman	jkorman	-8		62.40	19.62	3
ipkorman	jkorman	-9		62.96	19.60	3

Table V. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Pactual algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ipactual	jpactual	0	5.96	161.40	6.33	4
ipactual	jpactual	1		161.40	6.33	4
ipactual	jpactual	2		129.22	6.25	3
ipactual	jpactual	3		121.38	6.22	3
ipactual	jpactual	4		118.35	6.16	3
ipactual	jpactual	5		119.41	6.14	3
ipactual	jpactual	6		122.67	6.14	3
ipactual	jpactual	7		123.52	6.14	3
ipactual	jpactual	8		123.68	6.11	3
ipactual	jpactual	9		124.26	6.12	3
ipactual	jpactual	-1		140.24	6.24	4
ipactual	jpactual	-2		124.53	6.23	3
ipactual	jpactual	-3		120.08	6.22	3
ipactual	jpactual	-4		120.40	6.18	3
ipactual	jpactual	-5		120.88	6.18	3
ipactual	jpactual	-6		124.03	6.17	3
ipactual	jpactual	-7		123.73	6.12	3
ipactual	jpactual	-8		124.88	6.13	3
ipactual	jpactual	-9		125.11	6.12	3

Table VI. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Pactual algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ipactual	jpactual	0	8.23	851.32	9.52	20
ipactual	jpactual	1		851.32	9.52	20
ipactual	jpactual	2		702.96	9.35	16
ipactual	jpactual	3		639.00	9.34	15
ipactual	jpactual	4		650.68	9.33	15
ipactual	jpactual	5		740.20	9.30	17
ipactual	jpactual	6		735.63	9.29	17
ipactual	jpactual	7		741.08	9.29	17
ipactual	jpactual	8		745.38	9.27	17
ipactual	jpactual	9		745.66	9.26	17
ipactual	jpactual	-1		768.78	9.38	18
ipactual	jpactual	-2		780.65	9.39	18
ipactual	jpactual	-3		729.55	9.33	17
ipactual	jpactual	-4		765.62	9.32	18
ipactual	jpactual	-5		696.12	9.36	16
ipactual	jpactual	-6		698.95	9.38	16
ipactual	jpactual	-7		722.55	9.39	17
ipactual	jpactual	-8		738.51	9.34	17
ipactual	jpactual	-9		735.30	9.31	17

Table VII. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Pactual algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ipactual	jpactual	0	11.88	690.55	13.29	17
ipactual	jpactual	1		690.55	13.29	17
ipactual	jpactual	2		479.94	13.08	12
ipactual	jpactual	3		533.84	13.16	13
ipactual	jpactual	4		620.75	13.20	14
ipactual	jpactual	5		610.39	13.24	14
ipactual	jpactual	6		619.17	13.26	14
ipactual	jpactual	7		613.84	13.25	14
ipactual	jpactual	8		613.99	13.23	14
ipactual	jpactual	9		614.17	13.23	14
ipactual	jpactual	-1		669.09	13.24	16
ipactual	jpactual	-2		644.43	13.19	16
ipactual	jpactual	-3		580.39	13.14	14
ipactual	jpactual	-4		538.26	13.15	13
ipactual	jpactual	-5		531.43	13.07	13
ipactual	jpactual	-6		530.50	13.07	13
ipactual	jpactual	-7		567.34	13.09	14
ipactual	jpactual	-8		629.25	13.16	15
ipactual	jpactual	-9		590.23	13.22	14

Table VIII. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Pactual algorithm with different weights on the weight function

nucv	ncv	weight	exact color	moves	first color	time (sec)
ipactual	jpactual	0	19.20	73.42	19.54	3
ipactual	jpactual	1		73.42	19.54	3
ipactual	jpactual	2		60.31	19.50	2
ipactual	jpactual	3		56.21	19.50	2
ipactual	jpactual	4		58.12	19.50	2
ipactual	jpactual	5		58.17	19.49	2
ipactual	jpactual	6		58.51	19.52	2
ipactual	jpactual	7		58.37	19.52	2
ipactual	jpactual	8		58.18	19.50	2
ipactual	jpactual	9		58.71	19.51	2
ipactual	jpactual	-1		59.90	19.46	3
ipactual	jpactual	-2		60.84	19.47	3
ipactual	jpactual	-3		60.12	19.47	3
ipactual	jpactual	-4		60.81	19.51	3
ipactual	jpactual	-5		61.13	19.53	3
ipactual	jpactual	-6		60.03	19.48	3
ipactual	jpactual	-7		60.32	19.54	3
ipactual	jpactual	-8		60.45	19.52	3
ipactual	jpactual	-9		61.34	19.52	3

Table IX. VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2 (backtracking scheme). ikorman selection function with different kinds of ncv-selection functions

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.3	5.96	183.86	157.38	0.86	6.40	2
ikorman	jpactual			181.55	156.50	0.86	6.32	2
ikorman	jkormana			166.33	137.89	0.83	6.40	2
ikorman	jsucadja			167.55	140.41	0.84	6.48	2
ikorman	jpactuala			164.21	137.12	0.84	6.32	2
ikorman	jprevent1a			164.71	138.19	0.84	6.32	2
ikorman	jprevent2a			164.37	138.56	0.84	6.32	2
ikorman	jprevent3a			164.10	138.34	0.84	6.29	2
ikorman	jprevent4a			163.16	137.71	0.84	6.28	2
ikorman	jconnecta			164.38	138.57	0.84	6.31	3

*** S.D. column is the standard deviation of the forward moves.

Table X. VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2 (backtracking scheme). ikorman selection function with different kinds of ncv-selection functions

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.5	8.23	1108.45	1355.63	1.22	9.44	15
ikorman	jpactual			1046.49	1324.86	1.27	9.36	15
ikorman	jkormana			957.15	1157.19	1.21	9.44	14
ikorman	jsucadja			961.71	1148.28	1.19	9.59	15
ikorman	jpactuala			904.46	1133.29	1.25	9.36	14
ikorman	jprevent1a			925.96	1150.04	1.24	9.38	14
ikorman	jprevent2a			919.41	1143.75	1.24	9.38	14
ikorman	jprevent3a			921.82	1150.43	1.25	9.37	14
ikorman	jprevent4a			926.41	1161.72	1.25	9.41	14
ikorman	jconnecta			960.32	1157.19	1.21	9.42	16

*** S.D. column is the standard deviation of the forward moves.

Table XI. VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2 (backtracking scheme). ikorman selection function with different kinds of ncv-selection functions

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.7	11.88	833.55	736.86	0.88	13.14	11
ikorman	jpactual			751.28	669.39	0.89	13.03	11
ikorman	jkormana			714.82	620.46	0.87	13.14	11
ikorman	jsucadja			724.93	603.87	0.83	13.37	11
ikorman	jpactuala			647.18	566.28	0.88	13.03	10
ikorman	jprevent1a			690.41	614.46	0.89	13.13	11
ikorman	jprevent2a			687.09	610.14	0.89	13.11	11
ikorman	jprevent3a			686.94	609.32	0.89	13.12	11
ikorman	jprevent4a			690.55	616.66	0.89	13.09	11
ikorman	jconnecta			709.08	622.57	0.88	13.19	12

*** S.D. column is the standard deviation of the forward moves.

Table XII. VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2 (backtracking scheme). ikorman selection function with different kinds of ncv-selection functions

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.9	19.20	78.83	67.79	0.86	19.53	1
ikorman	jpactual			73.99	50.39	0.68	19.49	1
ikorman	jkormana			72.84	46.91	0.64	19.53	1
ikorman	jsucadja			74.31	45.40	0.61	19.64	1
ikorman	jpactuala			69.98	43.46	0.62	19.49	1
ikorman	jprevent1a			71.77	45.72	0.64	19.50	1
ikorman	jprevent2a			71.86	45.70	0.64	19.50	1
ikorman	jprevent3a			71.74	45.70	0.64	19.50	1
ikorman	jprevent4a			71.75	45.70	0.64	19.49	1
ikorman	jconnecta			72.92	45.79	0.63	19.56	1

*** S.D. column is the standard deviation of the forward moves.

Table XIII. VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2 (backtracking scheme). ipactual selection function with different kinds of ncv-selection function

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ipactual	jkorman	0.3	5.96	130.05	102.48	0.79	6.27	3
ipactual	jpactual			129.22	101.18	0.78	6.25	3
ipactual	jkormana			118.43	89.65	0.76	6.27	3
ipactual	jsucadja			119.37	89.77	0.75	6.33	3
ipactual	jpactuala			117.68	88.38	0.75	6.25	3
ipactual	jprevent1a			117.87	88.87	0.75	6.25	3
ipactual	jprevent2a			117.74	89.01	0.76	6.24	3
ipactual	jprevent3a			117.27	88.66	0.76	6.23	3
ipactual	jprevent4a			117.16	87.40	0.75	6.25	3
ipactual	jconnecta			116.45	88.62	0.76	6.20	3

*** S.D. column is the standard deviation of the forward moves.

Table XIV. VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2 (backtracking scheme). ipactual selection function with different kinds of ncv-selection function

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ipactual	jkorman	0.5	8.23	716.25	1009.91	1.41	9.43	16
ipactual	jpactual			702.96	1012.97	1.44	9.35	16
ipactual	jkormana			619.31	858.36	1.39	9.43	16
ipactual	jsucadja			600.25	847.55	1.41	9.59	15
ipactual	jpactuala			607.75	860.57	1.42	9.35	16
ipactual	jprevent1a			601.10	867.39	1.44	9.32	15
ipactual	jprevent2a			612.43	866.59	1.42	9.32	16
ipactual	jprevent3a			593.17	861.28	1.45	9.31	15
ipactual	jprevent4a			590.71	855.35	1.45	9.35	15
ipactual	jconnecta			602.85	863.88	1.43	9.36	16

*** S.D. column is the standard deviation of the forward moves.

Table XV. VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2 (backtracking scheme). ipactual selection function with different kinds of ncv-selection function

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ipactual	jkorman	0.7	11.88	502.67	603.20	1.20	13.09	12
ipactual	jpactual			479.94	571.13	1.19	13.08	12
ipactual	jkormana			430.99	499.09	1.16	13.09	11
ipactual	jsucadja			409.01	445.82	1.09	13.16	11
ipactual	jpactuala			411.81	472.76	1.15	13.08	11
ipactual	jprevent1a			411.23	488.54	1.19	13.07	11
ipactual	jprevent2a			411.48	488.43	1.19	13.07	11
ipactual	jprevent3a			411.96	488.58	1.19	13.08	11
ipactual	jprevent4a			420.41	499.87	1.19	13.08	11
ipactual	jconnecta			416.75	453.84	1.09	13.10	11

*** S.D. column is the standard deviation of the forward moves.

Table XVI. VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2 (backtracking scheme). ipactual selection function with different kinds of ncv-selection function

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ipactual	jkorman	0.9	19.20	61.11	38.25	0.63	19.52	2
ipactual	jpactual			60.31	38.54	0.64	19.50	2
ipactual	jkormana			58.56	33.73	0.58	19.52	2
ipactual	jsucadja			58.40	27.10	0.46	19.66	2
ipactual	jpactuala			57.85	34.02	0.59	19.50	2
ipactual	jprevent1a			58.13	34.24	0.59	19.50	2
ipactual	jprevent2a			58.13	34.24	0.59	19.50	2
ipactual	jprevent3a			58.28	34.15	0.59	19.51	2
ipactual	jprevent4a			58.38	33.63	0.58	19.51	2
ipactual	jconnecta			57.94	33.55	0.58	19.52	3

*** S.D. column is the standard deviation of the forward moves.

Table XVII. VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2 (backtracking scheme). Pactual algorithm with different weights on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
ipactual	jpactual	0.3	1.0	5.96	129.22	6.25	3
ipactual	jpactual		0.9		146.93	7.62	4
ipactual	jpactual		0.8		154.63	7.92	4
ipactual	jpactual		0.7		173.59	8.57	5
ipactual	jpactual		0.6		216.70	10.09	6
ipactual	jpactual		0.5		266.22	11.96	8
ipactual	jpactual		0.4		369.05	16.46	13
ipactual	jpactual		0.3		481.35	21.75	20
ipactual	jpactual		0.2		569.62	26.37	26
ipactual	jpactual		0.1		614.23	28.69	29

*** Weight column is the weight of new color.

Table XVIII. VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2 (backtracking scheme). Pactual algorithm with different weights on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
ipactual	jpactual	0.5	1.0	8.23	702.96	9.35	18
ipactual	jpactual		0.9		709.88	9.94	18
ipactual	jpactual		0.8		720.14	9.98	18
ipactual	jpactual		0.7		723.36	10.22	18
ipactual	jpactual		0.6		692.23	10.70	17
ipactual	jpactual		0.5		720.82	11.53	18
ipactual	jpactual		0.4		769.12	14.10	20
ipactual	jpactual		0.3		770.46	17.87	21
ipactual	jpactual		0.2		883.05	23.29	27
ipactual	jpactual		0.1		970.60	28.43	32

*** Weight column is the weight of new color.

Table XIX. VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2 (backtracking scheme). Pactual algorithm with different weights on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
ipactual	jpactual	0.7	1.0	11.88	479.94	13.08	12
ipactual	jpactual		0.9		481.33	13.24	12
ipactual	jpactual		0.8		488.99	13.24	13
ipactual	jpactual		0.7		488.46	13.30	13
ipactual	jpactual		0.6		507.22	13.47	13
ipactual	jpactual		0.5		488.89	13.60	13
ipactual	jpactual		0.4		518.58	14.70	13
ipactual	jpactual		0.3		540.24	16.45	14
ipactual	jpactual		0.2		605.18	20.17	16
ipactual	jpactual		0.1		677.58	25.66	19

*** Weight column is the weight of new color.

Table XX. VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2 (backtracking scheme). Pactual algorithm with different weights on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
ipactual	jpactual	0.9	1.0	19.20	60.31	19.50	3
ipactual	jpactual		0.9		61.09	19.64	3
ipactual	jpactual		0.8		61.44	19.65	3
ipactual	jpactual		0.7		61.41	19.68	3
ipactual	jpactual		0.6		61.92	19.69	3
ipactual	jpactual		0.5		62.00	19.67	3
ipactual	jpactual		0.4		63.68	19.94	3
ipactual	jpactual		0.3		65.48	20.31	3
ipactual	jpactual		0.2		78.46	21.35	3
ipactual	jpactual		0.1		113.08	23.80	4

*** Weight column is the weight of new color.

Table XXI. VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2 (backtracking scheme). Prevent4a algorithm with different weights on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
iprevent4a	jprevent4a	0.3	1.0	5.96	111.01	6.26	3
iprevent4a	jprevent4a		0.9		130.50	7.58	4
iprevent4a	jprevent4a		0.8		144.58	8.12	4
iprevent4a	jprevent4a		0.7		166.02	9.02	5
iprevent4a	jprevent4a		0.6		218.58	11.03	7
iprevent4a	jprevent4a		0.5		284.87	13.66	10
iprevent4a	jprevent4a		0.4		399.04	18.86	16
iprevent4a	jprevent4a		0.3		523.99	24.42	24
iprevent4a	jprevent4a		0.2		585.56	28.21	29
iprevent4a	jprevent4a		0.1		613.53	29.35	31

*** Weight column is the weight of new color.

Table XXII. VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2 (backtracking scheme). Prevent4a algorithm with different weights on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
iprevent4a	jprevent4a	0.5	1.0	8.23	631.19	9.35	17
iprevent4a	jprevent4a		0.9		632.28	9.83	17
iprevent4a	jprevent4a		0.8		634.82	9.95	17
iprevent4a	jprevent4a		0.7		647.29	10.43	18
iprevent4a	jprevent4a		0.6		637.85	11.11	17
iprevent4a	jprevent4a		0.5		633.77	12.61	17
iprevent4a	jprevent4a		0.4		615.32	16.37	18
iprevent4a	jprevent4a		0.3		787.18	21.65	25
iprevent4a	jprevent4a		0.2		856.56	27.87	31
iprevent4a	jprevent4a		0.1		1088.01	31.19	39

*** Weight column is the weight of new color.

Table XXIII. VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2 (backtracking scheme). Prevent4a algorithm with different weight on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
iprevent4a	jprevent4a	0.7	1.0	11.88	532.47	13.31	15
iprevent4a	jprevent4a		0.9		539.40	13.45	15
iprevent4a	jprevent4a		0.8		533.69	13.44	15
iprevent4a	jprevent4a		0.7		553.00	13.65	15
iprevent4a	jprevent4a		0.6		563.75	13.94	15
iprevent4a	jprevent4a		0.5		566.49	14.40	16
iprevent4a	jprevent4a		0.4		520.16	15.49	14
iprevent4a	jprevent4a		0.3		485.32	18.08	14
iprevent4a	jprevent4a		0.2		639.00	23.24	19
iprevent4a	jprevent4a		0.1		769.65	29.56	24

*** Weight column is the weight of new color.

Table XXIV. VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2 (backtracking scheme). Prevent4a algorithm with different weight on the new color

nucv	ncv	edge load	weight new clr	exact color	moves	first color	time (sec)
iprevent4a	jprevent4a	0.9	1.0	19.20	59.34	19.52	3
iprevent4a	jprevent4a		0.9		60.49	19.61	3
iprevent4a	jprevent4a		0.8		60.73	19.59	3
iprevent4a	jprevent4a		0.7		61.12	19.62	3
iprevent4a	jprevent4a		0.6		61.35	19.65	3
iprevent4a	jprevent4a		0.5		61.49	19.68	3
iprevent4a	jprevent4a		0.4		64.63	20.00	3
iprevent4a	jprevent4a		0.3		73.49	20.53	3
iprevent4a	jprevent4a		0.2		85.35	21.92	4
iprevent4a	jprevent4a		0.1		143.30	25.57	5

*** Weight column is the weight of new color.

Table XXV. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Korw21e, Korw21ec, and Korw12e with the parameter scale2 set to 0.0 and 0.5 and parameter scale1 set to 0.5, 0.7, and 0.9

nucv	ncv	scale 2	scale 1	exact color	moves	first color	time (sec)
ikorw21e	jkorman	0.0	0.5	5.96	144.66	6.41	2
ikorw21e	jkorman	0.0	0.7		143.04	6.35	2
ikorw21e	jkorman	0.0	0.9		143.01	6.35	2
ikorw21e	jkorman	0.5	0.5		181.97	6.39	3
ikorw21e	jkorman	0.5	0.7		183.51	6.41	3
ikorw21e	jkorman	0.5	0.9		183.51	6.41	3
ikorw21ec	jkorman	0.0	0.5		144.84	6.41	2
ikorw21ec	jkorman	0.0	0.7		143.04	6.35	2
ikorw21ec	jkorman	0.0	0.9		143.01	6.35	2
ikorw21ec	jkorman	0.5	0.5		181.97	6.39	3
ikorw21ec	jkorman	0.5	0.7		183.51	6.41	3
ikorw21ec	jkorman	0.5	0.9		183.51	6.41	3
ikorw12e	jkorman	0.0	0.5		149.97	6.39	2
ikorw12e	jkorman	0.0	0.7		143.06	6.35	2
ikorw12e	jkorman	0.0	0.9		143.01	6.35	2
ikorw12e	jkorman	0.5	0.5		182.04	6.39	2
ikorw12e	jkorman	0.5	0.7		183.53	6.41	3
ikorw12e	jkorman	0.5	0.9		183.51	6.41	3

Table XXVI. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Korw21e, Korw21ec, and Korw12e with the parameter scale2 set to 0.0 and 0.5 and parameter scale1 set to 0.5, 0.7, and 0.9

nucv	ncv	scale 2	scale 1	exact color	moves	first color	time (sec)
ikorw21e	jkorman	0.0	0.5	8.23	853.51	9.36	13
ikorw21e	jkorman	0.0	0.7		817.45	9.37	13
ikorw21e	jkorman	0.0	0.9		816.79	9.37	13
ikorw21e	jkorman	0.5	0.5		1026.26	9.36	16
ikorw21e	jkorman	0.5	0.7		995.00	9.40	16
ikorw21e	jkorman	0.5	0.9		993.98	9.40	16
ikorw21ec	jkorman	0.0	0.5		853.06	9.36	13
ikorw21ec	jkorman	0.0	0.7		817.46	9.37	13
ikorw21ec	jkorman	0.0	0.9		816.79	9.37	13
ikorw21ec	jkorman	0.5	0.5		1026.26	9.36	16
ikorw21ec	jkorman	0.5	0.7		995.00	9.40	16
ikorw21ec	jkorman	0.5	0.9		993.98	9.40	16
ikorw12e	jkorman	0.0	0.5		926.76	9.36	13
ikorw12e	jkorman	0.0	0.7		824.32	9.38	13
ikorw12e	jkorman	0.0	0.9		816.84	9.37	13
ikorw12e	jkorman	0.5	0.5		1053.01	9.36	15
ikorw12e	jkorman	0.5	0.7		1000.76	9.40	16
ikorw12e	jkorman	0.5	0.9		994.00	9.40	16

Table XXVII. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Korw21e, Korw21ec, and Korw12e with the parameter scale2 set to 0.0 and 0.5 and parameter scale1 set to 0.5, 0.7, and 0.9

nucv	ncv	scale 2	scale 1	exact color	moves	first color	time (sec)
ikorw21e	jkorman	0.0	0.5	11.88	633.70	12.98	10
ikorw21e	jkorman	0.0	0.7		621.53	13.08	10
ikorw21e	jkorman	0.0	0.9		623.22	13.11	10
ikorw21e	jkorman	0.5	0.5		707.15	13.00	11
ikorw21e	jkorman	0.5	0.7		668.12	13.03	11
ikorw21e	jkorman	0.5	0.9		665.84	13.04	11
ikorw21ec	jkorman	0.0	0.5		633.67	12.98	10
ikorw21ec	jkorman	0.0	0.7		621.44	13.08	10
ikorw21ec	jkorman	0.0	0.9		623.22	13.11	10
ikorw21ec	jkorman	0.5	0.5		707.15	13.00	11
ikorw21ec	jkorman	0.5	0.7		668.12	13.03	11
ikorw21ec	jkorman	0.5	0.9		665.84	13.04	11
ikorw12e	jkorman	0.0	0.5		704.41	13.00	10
ikorw12e	jkorman	0.0	0.7		638.76	13.08	11
ikorw12e	jkorman	0.0	0.9		623.53	13.11	11
ikorw12e	jkorman	0.5	0.5		775.56	12.94	11
ikorw12e	jkorman	0.5	0.7		686.16	13.03	11
ikorw12e	jkorman	0.5	0.9		666.17	13.04	11

Table XXVIII. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Korw21e, Korw21ec, and Korw12e with the parameter scale2 set to 0.0 and 0.5 and parameter scale1 set to 0.5, 0.7, and 0.9

nucv	ncv	scale 2	scale 1	exact color	moves	first color	time (sec)
ikorw21e	jkorman	0.0	0.5	19.20	61.87	19.60	1
ikorw21e	jkorman	0.0	0.7		59.29	19.56	1
ikorw21e	jkorman	0.0	0.9		58.52	19.56	1
ikorw21e	jkorman	0.5	0.5		62.28	19.60	1
ikorw21e	jkorman	0.5	0.7		59.36	19.55	1
ikorw21e	jkorman	0.5	0.9		58.54	19.55	1
ikorw21ec	jkorman	0.0	0.5		61.87	19.60	1
ikorw21ec	jkorman	0.0	0.7		59.29	19.56	1
ikorw21ec	jkorman	0.0	0.9		58.52	19.56	1
ikorw21ec	jkorman	0.5	0.5		62.28	19.60	1
ikorw21ec	jkorman	0.5	0.7		59.36	19.55	1
ikorw21ec	jkorman	0.5	0.9		58.54	19.55	1
ikorw12e	jkorman	0.0	0.5		64.40	19.61	1
ikorw12e	jkorman	0.0	0.7		60.24	19.56	1
ikorw12e	jkorman	0.0	0.9		58.77	19.56	1
ikorw12e	jkorman	0.5	0.5		64.79	19.61	1
ikorw12e	jkorman	0.5	0.7		60.30	19.55	1
ikorw12e	jkorman	0.5	0.9		58.79	19.55	1

Table XXIX. VERTICES = 40, EDGELOAD = 0.3 AND 0.5 (backtracking scheme). Korw2e and Korw21e with the parameter scale2 set to 0.0, 0.5, and 1.0 and parameter scale1 set to 0.5

nucv	ncv	scale 2	scale 1	edge load	exact color	moves	var cof.	first color	time (sec)
ikorw2e	jkorman	0.0		0.3	5.96	143.01	0.822	6.35	2
ikorw21e	jkorman	0.0	0.5			144.66	0.766	6.41	2
ikorw2e	jkorman	0.5				183.51	0.852	6.41	2
ikorw21e	jkorman	0.5	0.5			181.97	0.837	6.39	3
ikorw2e	jkorman	1.0				183.86	0.856	6.40	2
ikorw21e	jkorman	1.0	0.5			182.04	0.839	6.39	3
ikorw2e	jkorman	0.0		0.5	8.23	816.79	1.378	9.37	12
ikorw21e	jkorman	0.0	0.5			853.51	1.412	9.36	13
ikorw2e	jkorman	0.5				993.98	1.233	9.40	15
ikorw21e	jkorman	0.5	0.5			1026.26	1.255	9.36	16
ikorw2e	jkorman	1.0				1108.45	1.223	9.44	16
ikorw21e	jkorman	1.0	0.5			1065.36	1.264	9.36	16

Table XXX. VERTICES = 40, EDGELOAD = 0.7 AND 0.9 (backtracking scheme). Korw2e and Korw21e with the parameter scale2 set to 0.0, 0.5, and 1.0 and parameter scale1 set to 0.5

nucv	ncv	scale 2	scale 1	edge load	exact color	moves	var cof.	first color	time (sec)
ikorw2e	jkorman	0.0		0.5	11.88	623.14	1.107	13.11	10
ikorw21e	jkorman	0.0	0.5			633.70	1.077	12.98	10
ikorw2e	jkorman	0.5				665.77	1.149	13.04	10
ikorw21e	jkorman	0.5	0.5			707.15	1.113	13.00	11
ikorw2e	jkorman	1.0				833.55	0.884	13.14	13
ikorw21e	jkorman	1.0	0.5			716.81	0.754	13.03	11
ikorw2e	jkorman	0.0		0.5	19.20	58.47	0.523	19.56	1
ikorw21e	jkorman	0.0	0.5			61.87	0.518	19.60	1
ikorw2e	jkorman	0.5				58.47	0.528	19.55	1
ikorw21e	jkorman	0.5	0.5			62.28	0.533	19.60	1
ikorw2e	jkorman	1.0				78.83	0.707	19.53	1
ikorw21e	jkorman	1.0	0.5			77.30	0.634	19.48	1

Table XXXI. VERTICES = 44, EDGELOAD = 0.3 AND 0.5 (backtracking scheme). Korw2e and Korw21e with the parameter scale2 set to 0.0, 0.5, and 1.0 and parameter scale1 set to 0.5

nucv	ncv	scale 2	scale 1	edge load	exact color	moves	var cof.	first color	time (sec)
ikorw2e	jkorman	0.0		0.3	6.00	139.26	0.982	6.75	2
ikorw21e	jkorman	0.0	0.5			141.95	0.998	6.72	2
ikorw2e	jkorman	0.5				181.66	1.121	6.75	2
ikorw21e	jkorman	0.5	0.5			158.60	1.067	6.72	2
ikorw2e	jkorman	1.0				181.73	1.122	6.74	2
ikorw21e	jkorman	1.0	0.5			158.67	1.067	6.72	2
ikorw2e	jkorman	0.0		0.5	8.92	975.65	0.742	10.02	14
ikorw21e	jkorman	0.0	0.5			986.64	0.748	9.91	14
ikorw2e	jkorman	0.5				1240.22	0.822	10.00	18
ikorw21e	jkorman	0.5	0.5			1277.36	0.787	10.01	19
ikorw2e	jkorman	1.0				1408.74	0.845	10.07	20
ikorw21e	jkorman	1.0	0.5			1331.66	0.796	10.01	20

Table XXXII. VERTICES = 44, EDGELOAD = 0.7 AND 0.9 (backtracking scheme). Korw2e and Korw21e with the parameter scale2 set to 0.0, 0.5, and 1.0 and parameter scale1 set to 0.5

nucv	ncv	scale 2	scale 1	edge load	exact color	moves	var cof.	first color	time (sec)
ikorw2e	jkorman	0.0		0.5	11.88	623.14	1.107	13.11	10
ikorw21e	jkorman	0.0	0.5			633.70	1.077	12.98	10
ikorw2e	jkorman	0.5				665.77	1.149	13.04	10
ikorw21e	jkorman	0.5	0.5			707.15	1.113	13.00	11
ikorw2e	jkorman	1.0				833.55	0.884	13.14	13
ikorw21e	jkorman	1.0	0.5			716.81	0.754	13.03	11
ikorw2e	jkorman	0.0		0.5	19.20	58.47	0.523	19.56	1
ikorw21e	jkorman	0.0	0.5			61.87	0.518	19.60	1
ikorw2e	jkorman	0.5				58.47	0.528	19.55	1
ikorw21e	jkorman	0.5	0.5			62.28	0.533	19.60	1
ikorw2e	jkorman	1.0				78.83	0.707	19.53	1
ikorw21e	jkorman	1.0	0.5			77.30	0.634	19.48	1

Table XXXIII. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). variations of Korman with swapping and Pactual with swapping Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	5.96	183.86	157.38	0.86	6.40	2
ikorqk2	jkorman		143.31	116.37	0.81	6.36	2
ikorpw2	jkorman		145.09	120.28	0.83	6.36	2
ikorw2	jkorman		143.01	117.55	0.82	6.35	2
ikorw2p	jkorman		152.14	128.25	0.84	6.36	2
ikorw21e	jkorman		144.66	110.81	0.77	6.41	2
ikormaxw2	jkorman		145.19	120.94	0.83	6.36	2
ikormaxw21e	jkorman		147.96	114.23	0.77	6.42	2
ikorqk23	jkorman		116.93	87.70	0.75	6.36	3
ikorw23	jkorman		115.41	91.40	0.79	6.34	3
ipactqk2	jkorman		115.25	87.36	0.76	6.22	3
ipactqk2	jpactual		111.03	87.71	0.79	6.15	3
ipactmaxw2	jkorman		114.38	86.81	0.76	6.24	3
ipactmaxw2	jpactual		112.80	87.31	0.77	6.18	3
ipactual	jpactuala		117.68	88.38	0.75	6.25	3

*** S.D. column is the standard deviation of the forward moves.

Table XXXIV. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). variations of Korman with swapping and Pactual with swapping Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	8.23	1108.45	1355.63	1.22	9.44	15
ikorqk2	jkorman		836.46	1137.59	1.36	9.45	12
ikorpw2	jkorman		812.58	1118.92	1.38	9.42	12
ikorw2	jkorman		816.79	1125.54	1.38	9.37	12
ikorw2p	jkorman		828.04	1146.01	1.38	9.35	12
ikorw21e	jkorman		853.51	1205.16	1.41	9.36	13
ikormaxw2	jkorman		755.45	929.96	1.23	9.42	11
ikormaxw21e	jkorman		785.61	974.16	1.24	9.37	12
ikorqk23	jkorman		612.04	870.32	1.42	9.39	23
ikorw23	jkorman		593.44	854.55	1.44	9.41	27
ipactqk2	jkorman		595.80	824.59	1.38	9.36	14
ipactqk2	jpactual		598.23	830.34	1.39	9.19	15
ipactmaxw2	jkorman		577.88	816.54	1.41	9.38	13
ipactmaxw2	jpactual		580.28	824.00	1.42	9.21	14
ipactual	jpactuala		607.75	860.57	1.42	9.35	16

*** S.D. column is the standard deviation of the forward moves.

Table XXXV. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). variations of Korman with swapping and Pactual with swapping Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	11.88	833.55	736.86	0.88	13.14	11
ikorqk2	jkorman		579.78	613.99	1.06	13.08	9
ikorpw2	jkorman		666.11	750.04	1.13	13.11	10
ikorw2	jkorman		623.14	689.82	1.11	13.11	10
ikorw2p	jkorman		623.52	690.86	1.11	13.11	10
ikorw21e	jkorman		633.70	682.49	1.08	12.98	10
ikormaxw2	jkorman		621.53	679.95	1.09	13.09	10
ikormaxw21e	jkorman		611.86	622.87	1.02	12.98	9
ikorqk23	jkorman		470.74	496.16	1.05	12.95	31
ikorw23	jkorman		479.32	554.09	1.16	12.97	38
ipactqk2	jkorman		397.00	452.58	1.14	12.99	10
ipactqk2	jpactual		381.62	441.15	1.16	12.89	10
ipactmaxw2	jkorman		381.16	434.14	1.14	13.01	9
ipactmaxw2	jpactual		366.49	424.40	1.16	12.89	9
ipactual	jpactuala		411.81	472.76	1.15	13.08	12

*** S.D. column is the standard deviation of the forward moves.

Table XXXVI. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). variations of Korman with swapping and Pactual with swapping Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	19.20	78.83	67.79	0.86	19.53	1
ikorqk2	jkorman		57.79	29.88	0.52	19.55	1
ikorpw2	jkorman		58.96	30.66	0.52	19.56	1
ikorw2	jkorman		58.47	30.58	0.52	19.56	1
ikorw2p	jkorman		58.47	30.58	0.52	19.56	1
ikorw21e	jkorman		61.87	32.05	0.52	19.60	1
ikormaxw2	jkorman		58.10	30.68	0.53	19.54	1
ikormaxw21e	jkorman		61.47	32.15	0.52	19.58	1
ikorqk23	jkorman		52.96	23.57	0.45	19.55	5
ikorw23	jkorman		51.90	21.49	0.41	19.55	6
ipactqk2	jkorman		54.59	29.81	0.55	19.46	2
ipactqk2	jpactual		53.65	28.86	0.54	19.44	2
ipactmaxw2	jkorman		54.59	29.81	0.55	19.46	2
ipactmaxw2	jpactual		53.65	28.86	0.54	19.44	2
ipactual	jpactuala		57.85	34.02	0.59	19.50	2

*** S.D. column is the standard deviation of the forward moves.

Table XXXVII. VERTICES = 40, EDGELOAD = 0.3, WEIGHT = 2 (backtracking scheme). variations of Korman without swapping

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	5.96	183.86	157.38	0.86	6.40	2
ikorman	jkormana		166.33	137.89	0.83	6.40	2
ikorman	jsucadja		167.55	140.41	0.84	6.48	2
ipkorman	jkorman		147.43	128.56	0.87	6.23	3
ipactual	jpactual		129.22	101.18	0.78	6.25	3
ipactual	jpactuala		117.68	88.38	0.75	6.25	3
iprevent1a	jprevent1a		121.79	91.71	0.75	6.16	3
iprevent2a	jprevent2a		114.15	82.64	0.72	6.14	3
iprevent3a	jprevent3a		116.60	89.20	0.77	6.20	3
iprevent4a	jprevent4a		111.01	76.71	0.69	6.26	3
iconnecta	jconnecta		150.41	120.18	0.80	6.28	4

*** S.D. column is the standard deviation of the forward moves.

Table XXXVIII. VERTICES = 40, EDGELOAD = 0.5, WEIGHT = 2 (backtracking scheme). variations of Korman without swapping

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	8.23	1108.45	1355.63	1.22	9.44	15
ikorman	jkormana		957.15	1157.19	1.21	9.44	14
ikorman	jsucadja		961.71	1148.28	1.19	9.59	15
ipkorman	jkorman		1076.45	1815.97	1.69	9.42	21
ipactual	jpactual		702.96	1012.97	1.44	9.35	16
ipactual	jpactuala		607.75	860.57	1.42	9.35	16
iprevent1a	jprevent1a		765.27	1435.65	1.88	9.33	17
iprevent2a	jprevent2a		656.13	1076.71	1.64	9.36	15
iprevent3a	jprevent3a		700.80	1140.20	1.63	9.36	16
iprevent4a	jprevent4a		631.19	883.03	1.40	9.35	15
iconnecta	jconnecta		1051.76	1880.55	1.79	9.42	29

*** S.D. column is the standard deviation of the forward moves.

Table XXXIX. VERTICES = 40, EDGELOAD = 0.7, WEIGHT = 2 (backtracking scheme). variations of Korman without swapping

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	11.88	833.55	736.86	0.88	13.14	11
ikorman	jkormana		714.82	620.46	0.87	13.14	11
ikorman	jsucadja		724.93	603.87	0.83	13.37	11
ipkorman	jkorman		880.44	967.60	1.10	13.22	18
ipactual	jpactual		479.94	571.13	1.19	13.08	12
ipactual	jpactuala		411.81	472.76	1.15	13.08	11
iprevent1a	jprevent1a		545.57	598.49	1.10	13.09	12
iprevent2a	jprevent2a		528.60	565.07	1.07	13.16	13
iprevent3a	jprevent3a		529.34	601.33	1.14	13.05	13
iprevent4a	jprevent4a		532.47	530.34	1.00	13.31	13
iconnecta	jconnecta		847.48	1125.45	1.33	13.12	23

*** S.D. column is the standard deviation of the forward moves.

Table XL. VERTICES = 40, EDGELOAD = 0.9, WEIGHT = 2 (backtracking scheme). variations of Korman without swapping

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	19.20	78.83	67.79	0.86	19.53	1
ikorman	jkormana		72.84	46.91	0.64	19.53	1
ikorman	jsucadja		74.31	45.40	0.61	19.64	1
ipkorman	jkorman		62.88	103.88	1.65	19.62	3
ipactual	jpactual		60.31	38.54	0.64	19.50	2
ipactual	jpactuala		57.85	34.02	0.59	19.50	2
iprevent1a	jprevent1a		58.08	33.22	0.57	19.55	2
iprevent2a	jprevent2a		56.18	31.29	0.56	19.51	3
iprevent3a	jprevent3a		56.04	32.90	0.59	19.48	3
iprevent4a	jprevent4a		59.34	27.83	0.47	19.52	3
iconnecta	jconnecta		79.64	121.05	1.52	19.59	3

*** S.D. column is the standard deviation of the forward moves.

Table XLI. VERTICES = 44, EDGELOAD = 0.3 AND 0.5, WEIGHT = 2 (backtracking scheme). variations of Korman with swapping and Pactual with swapping

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.3	6.00	181.73	203.90	1.12	6.74	2
ikorw2	jkorman			139.26	136.75	0.98	6.75	2
ikorw21e	jkorman			141.95	141.67	1.00	6.72	2
ikormaxw2	jkorman			140.25	138.43	0.99	6.74	2
ikormaxw21e	jkorman			145.42	141.93	0.98	6.72	2
ipactmaxw2	jpactual			104.76	69.25	0.66	6.57	2
ikorman	jkorman	0.5	8.92	1408.74	1190.39	0.85	10.07	17
ikorw2	jkorman			975.65	723.93	0.74	10.02	14
ikorw21e	jkorman			986.64	738.01	0.75	9.91	14
ikormaxw2	jkorman			981.87	727.57	0.74	9.97	15
ikormaxw21e	jkorman			1009.88	740.24	0.73	9.88	15
ipactmaxw2	jpactual			656.32	539.50	0.82	9.69	16

*** S.D. column is the standard deviation of the forward moves.

Table XLII. VERTICES = 44, EDGELOAD = 0.7 AND 0.9, WEIGHT = 2 (backtracking scheme). variations of Korman with swapping and Pactual with swapping

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.7	12.72	2133.82	2528.58	1.19	14.18	27
ikorw2	jkorman			1565.48	1682.89	1.08	14.25	23
ikorw21c	jkorman			1496.65	1731.62	1.16	14.07	23
ikormaxw2	jkorman			1605.68	1679.54	1.05	14.29	25
ikormaxw21e	jkorman			1535.77	1738.49	1.13	14.09	23
ipactmaxw2	jpactual			1018.76	1407.93	1.38	13.99	26
ikorman	jkorman	0.9	20.52	172.54	224.47	1.30	21.03	2
ikorw2	jkorman			143.41	273.63	1.91	21.02	2
ikorw21e	jkorman			135.12	227.68	1.69	20.91	2
ikormaxw2	jkorman			128.55	204.27	1.59	21.02	2
ikormaxw21e	jkorman			121.85	173.15	1.42	20.90	2
ipactmaxw2	jpactual			99.34	135.00	1.36	20.85	3

*** S.D. column is the standard deviation of the forward moves.

Table XLIII. VERTICES = 48, EDGELOAD = 0.3, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	6.03	888.34	1430.23	1.61	7.20	10
ikorman	jkormana		792.46	1256.84	1.59	7.20	11
ikorman	jsucadja		816.78	1184.33	1.45	7.22	12
ipkorman	jkorman		760.85	1292.68	1.70	7.01	14
ipactual	jpactual		599.01	909.30	1.52	7.02	14
ipactual	jpactuala		532.76	785.82	1.48	7.02	14
iprevent1a	jprevent1a		611.68	1071.66	1.75	7.01	14
iprevent2a	jprevent2a		606.67	1000.40	1.65	7.01	14
iprevent3a	jprevent3a		567.97	942.83	1.66	7.03	14
iprevent4a	jprevent4a		572.21	790.79	1.38	7.06	14
ikorqk2	jkorman		793.20	1460.28	1.84	7.17	10
ikorw2	jkorman		735.36	1261.88	1.72	7.15	9
ikorw21e	jkorman		759.51	1269.90	1.67	7.12	10
ikormaxw2	jkorman		769.72	1290.82	1.68	7.16	10
ikormaxw21e	jkorman		791.74	1292.12	1.63	7.10	11
ipactqk2	jpactual		517.88	775.78	1.50	6.99	13
ipactmaxw2	jpactual		500.69	747.03	1.49	6.98	12

*** S.D. column is the standard deviation of the forward moves.

Table XLIV. VERTICES = 48, EDGELOAD = 0.5, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual
Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	9.09	6297.14	7606.95	1.21	10.52	81
ikorman	jkormana		5413.61	6517.99	1.20	10.52	83
ikorman	jsucadja		5459.40	6655.01	1.22	10.72	88
ipkorman	jkorman		6160.09	9671.34	1.57	10.67	123
ipactual	jpactual		3150.71	3708.39	1.18	10.51	78
ipactual	jpactuala		2715.87	3185.72	1.17	10.51	75
iprevent1a	jprevent1a		3884.14	4711.46	1.21	10.62	92
iprevent2a	jprevent2a		3701.90	4534.83	1.23	10.60	92
iprevent3a	jprevent3a		3604.15	4847.58	1.35	10.52	89
iprevent4a	jprevent4a		2757.97	3251.65	1.18	10.61	72
ikorqk2	jkorman		5474.47	7910.61	1.45	10.70	81
ikorw2	jkorman		4738.64	6046.50	1.28	10.61	70
ikorw21e	jkorman		4720.14	6060.66	1.28	10.55	73
ikormaxw2	jkorman		4432.30	5473.89	1.24	10.58	66
ikormaxw21e	jkorman		4422.06	5403.76	1.22	10.55	69
ipactqk2	jpactual		2859.02	4151.30	1.45	10.33	77
ipactmaxw2	jpactual		2586.49	2974.46	1.15	10.34	68

*** S.D. column is the standard deviation of the forward moves.

Table XLV. VERTICES = 48, EDGELOAD = 0.7, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	13.36	9399.03	14248.93	1.52	15.11	126
ikorman	jkormana		7995.03	12064.50	1.51	15.11	125
ikorman	jsucadja		7964.36	11850.97	1.49	15.30	132
ipkorman	jkorman		8845.18	13435.83	1.52	15.11	182
ipactual	jpactual		5030.90	7938.76	1.58	15.03	126
ipactual	jpactuala		4274.90	6685.94	1.56	15.03	119
iprevent1a	jprevent1a		5499.27	9354.26	1.70	15.19	131
iprevent2a	jprevent2a		4708.43	6827.22	1.45	15.12	117
iprevent3a	jprevent3a		5114.55	8945.35	1.75	15.04	128
iprevent4a	jprevent4a		4849.15	8287.20	1.71	15.24	128
ikorqk2	jkorman		6527.48	10502.72	1.61	15.17	103
ikorw2	jkorman		6990.07	15000.69	2.15	15.13	112
ikorw21e	jkorman		7084.65	15579.15	2.20	15.04	116
ikormaxw2	jkorman		6642.25	10873.36	1.64	15.14	106
ikormaxw21e	jkorman		6605.28	11248.79	1.70	15.05	109
ipactqk2	jpactual		3993.22	7311.59	1.83	14.83	111
ipactmaxw2	jpactual		4023.01	7337.97	1.82	14.80	111

*** S.D. column is the standard deviation of the forward moves.

Table XLVI. VERTICES = 48, EDGELOAD = 0.9, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	21.85	428.55	1021.66	2.38	22.61	5
ikorman	jkormana		375.73	877.33	2.34	22.61	5
ikorman	jsucadja		385.89	878.29	2.28	22.76	6
ipkorman	jkorman		687.64	3109.51	4.52	22.64	13
ipactual	jpactual		271.59	747.42	2.75	22.51	8
ipactual	jpactuala		240.66	637.99	2.65	22.51	8
iprevent1a	jprevent1a		279.22	758.36	2.72	22.60	8
iprevent2a	jprevent2a		437.66	1896.82	4.33	22.58	11
iprevent3a	jprevent3a		342.05	1492.36	4.36	22.48	9
iprevent4a	jprevent4a		253.45	389.30	1.54	22.68	9
ikorqk2	jkorman		313.63	684.65	2.18	22.58	5
ikorw2	jkorman		309.98	686.30	2.21	22.59	5
ikorw21e	jkorman		286.24	545.29	1.91	22.54	5
ikormaxw2	jkorman		303.76	683.46	2.25	22.59	5
ikormaxw21e	jkorman		277.28	535.43	1.93	22.54	5
ipactqk2	jpactual		211.88	522.71	2.47	22.39	7
ipactmaxw2	jpactual		211.57	522.37	2.47	22.39	7

*** S.D. column is the standard deviation of the forward moves.

Table XLVII. VERTICES = 52, EDGELOAD = 0.3 AND 0.5, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual
Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.3	6.78	3196.80	3631.56	1.14	7.43	52
ikorw2	jkorman			2581.08	2981.22	1.16	7.50	45
ikormaxw2	jkorman			2535.05	2849.40	1.12	7.46	44
ikormaxw21e	jkorman			2593.06	2935.34	1.13	7.49	46
ipactmaxw2	jpactual			1582.63	1834.27	1.16	7.27	48
ikorman	jkorman	0.5	9.93	15096.79	15685.56	1.04	11.29	260
ikorw2	jkorman			11365.67	12408.66	1.09	11.17	216
ikormaxw2	jkorman			11458.22	12420.71	1.08	11.14	218
ikormaxw21e	jkorman			11855.98	13053.43	1.10	11.14	226
ipactmaxw2	jpactual			7580.97	8278.42	1.09	11.14	238

*** S.D. column is the standard deviation of the forward moves.

Table XLVIII. VERTICES = 52, EDGELOAD = 0.7 AND 0.9, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual
Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.7	14.07	30856.41	35268.88	1.14	16.09	536
ikorw2	jkorman			19604.14	21411.50	1.10	16.12	392
ikormaxw2	jkorman			20455.95	23769.81	1.16	16.10	407
ikormaxw21e	jkorman			19131.89	22269.52	1.16	15.86	371
ipactmaxw2	jpactual			9030.25	10673.76	1.18	15.82	291
ikorman	jkorman	0.9	23.10	691.42	1028.83	1.49	24.20	11
ikorw2	jkorman			498.50	1158.39	2.32	24.14	9
ikormaxw2	jkorman			517.23	1164.28	2.25	24.11	9
ikormaxw21e	jkorman			458.22	727.65	1.59	23.96	9
ipactmaxw2	jpactual			316.38	398.96	1.26	23.89	11

*** S.D. column is the standard deviation of the forward moves.

Table XLIX. VERTICES = 56, EDGELOAD = 0.3 AND 0.5, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.3	7.00	2283.26	2282.69	1.00	7.92	41
ikorw2	jkorman			1736.23	1719.78	0.99	7.84	34
ipactual	jpactual			1426.96	1420.42	1.00	7.77	50
ipactmaxw2	jpactual			1187.76	1114.47	0.94	7.70	41
ikorman	jkorman	0.5	10.03	40819.83	61973.11	1.52	11.98	728
ikorw2	jkorman			32495.33	58278.02	1.79	11.86	644
ipactual	jpactual			16859.05	23336.84	1.38	11.77	548
ipactmaxw2	jpactual			14917.26	20858.73	1.40	11.69	491

*** S.D. column is the standard deviation of the forward moves.

Table L. VERTICES = 56, EDGELOAD = 0.7 AND 0.9, WEIGHT = 2 (backtracking scheme). variations of Korman and Pactual Default: scale1 = 0.5, scale2 = 0.0.

nucv	ncv	edge load	exact color	moves	S.D.	var cof.	first color	time (sec)
ikorman	jkorman	0.7	14.86	100592.26	138854.67	1.38	16.96	1864
ikorw2	jkorman			67298.88	76181.61	1.13	16.88	1448
ipactual	jpactual			50482.88	67243.74	1.33	17.05	1663
ipactmaxw2	jpactual			35781.40	38021.55	1.06	16.82	1251
ikorman	jkorman	0.9	24.47	3194.90	9295.03	2.91	25.70	54
ikorw2	jkorman			2209.50	6524.20	2.95	25.72	51
ipactual	jpactual			1528.46	4353.31	2.85	25.59	45
ipactmaxw2	jpactual			1145.27	2976.75	2.60	25.49	37

*** S.D. column is the standard deviation of the forward moves.

Table LI. COMPARISON KORW2 WITH KORMAN IN RUNNING TIME (backtracking scheme).

edgeload = 0.3

	n = 28	n = 32	n = 36	n = 40	n = 44	n = 48	n = 52	n = 56
mean	-0.01	0.03	-0.02	*-0.37	*-0.31	-1.03	*-7.24	*-7.44
S.D.	0.10	0.46	0.97	0.84	1.56	10.52	18.45	10.14
better	1%	5%	22%	34%	21%	51%	84%	88%
worse	0%	5%	13%	4%	8%	29%	9%	11%

edgeload = 0.5

	n = 28	n = 32	n = 36	n = 40	n = 44	n = 48	n = 52	n = 56
mean	*-0.12	-0.11	*-0.97	*-2.69	*-3.67	*-11.55	*-44.29	-83.70
S.D.	0.38	0.72	2.82	10.87	6.81	49.88	55.82	737.34
better	14%	22%	45%	65%	75%	71%	92%	72%
worse	2%	13%	8%	17%	17%	26%	7%	27%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Korw2 is better (worse) than Korman.

3. '*' mean is significant at 0.05 level.

Table LII. COMPARISON KORW2 WITH KORMAN IN RUNNING TIME (backtracking scheme).

edgeload = 0.7

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*-0.13	*-0.25	*-0.82	-1.69	*-3.79	-15.22	*-143.84	*-415.76
S.D.	0.46	0.81	2.32	9.50	15.66	111.41	326.92	1547.02
better	14%	24%	53%	65%	61%	63%	74%	75%
worse	3%	6%	18%	23%	30%	34%	24%	24%

edgeload = 0.9

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	0.01	-0.04	*-0.11	-0.02	0.01	*-0.76	*-2.05	*-9.85
S.D.	0.10	0.24	0.47	0.64	2.22	3.67	9.40	33.97
better	0%	5%	15%	17%	25%	36%	67%	67%
worse	1%	1%	5%	17%	14%	14%	13%	17%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Korw2 is better (worse) than Korman.

3. '*' mean is significant at 0.05 level.

Table LIII. COMPARISON PACTUAL WITH KORMAN IN RUNNING TIME (backtracking scheme).

edgeload = 0.3

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.63	*0.90	*0.84	*1.03	*1.05	*3.94	4.19	*8.51
S.D.	0.49	0.56	1.82	1.57	1.86	17.72	37.79	20.20
better	0%	1%	9%	8%	9%	29%	34%	24%
worse	63%	84%	59%	66%	77%	65%	60%	72%

edgeload = 0.5

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.66	*0.70	0.36	1.79	*2.71	-3.05	13.23	*-179.61
S.D.	1.08	1.40	3.35	16.99	11.66	83.13	159.17	829.88
better	2%	8%	23%	37%	34%	39%	47%	60%
worse	59%	55%	57%	55%	61%	59%	52%	40%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Pactual is better (worse) than Korman.

3. '*' mean is significant at 0.05 level.

Table LIV. COMPARISON PACTUAL WITH KORMAN IN RUNNING TIME (backtracking scheme).

edgeload = 0.7

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.67	*0.63	*1.14	0.30	3.19	-0.26	*-190.23	-201.08
S.D.	0.53	1.13	3.69	12.50	25.14	88.18	561.78	2140.45
better	1%	6%	20%	48%	37%	45%	62%	68%
worse	66%	60%	57%	43%	58%	55%	37%	32%

edgeload = 0.9

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.99	*0.94	*1.67	*1.54	*1.96	*2.30	*2.87	-8.87
S.D.	0.10	0.24	0.59	0.73	1.41	4.99	14.44	46.16
better	0%	0%	0%	0%	3%	15%	17%	42%
worse	99%	94%	94%	93%	89%	81%	79%	55%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Pactual is better (worse) than Korman.

3. '*' mean is significant at 0.05 level.

Table LV. COMPARISON PACTMAXW2 WITH KORMAN IN RUNNING TIME (backtracking scheme).

edgeload = 0.3

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.41	*0.80	*0.50	*0.59	*0.55	1.82	-4.18	-0.60
S.D.	0.49	0.49	1.54	1.34	2.06	16.53	36.58	18.62
better	0%	1%	13%	13%	15%	34%	51%	51%
worse	41%	78%	50%	53%	67%	56%	44%	43%

edgeload = 0.5

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.45	*0.48	-0.27	-0.63	-1.10	-13.09	-22.09	*-236.86
S.D.	0.86	1.31	3.26	14.99	10.00	80.34	112.87	897.88
better	3%	9%	29%	46%	49%	51%	61%	62%
worse	42%	49%	46%	43%	43%	48%	39%	37%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Pactmaxw2 is better (worse) than Korman.

3. '*' mean is significant at 0.05 level.

Table LVI. COMPARISON PACTMAXW2 WITH KORMAN IN RUNNING TIME (backtracking scheme).

edgeload = 0.7

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.59	*0.37	0.16	-1.78	-1.27	-15.57	*-244.69	*-613.09
S.D.	0.53	1.06	3.19	9.82	26.64	84.86	546.24	2245.21
better	1%	8%	30%	56%	51%	58%	70%	72%
worse	59%	43%	40%	35%	46%	41%	29%	27%

edgeload = 0.9

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*0.99	*0.94	*1.40	*1.37	*1.38	*1.46	0.26	*-16.69
S.D.	0.10	0.24	0.77	0.68	1.44	4.35	10.76	72.43
better	0%	0%	2%	0%	6%	17%	27%	45%
worse	99%	94%	87%	90%	85%	79%	70%	49%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Pactmaxw2 is better (worse) than Korman.

3. '*' mean is significant at 0.05 level.

Table LVII. COMPARISON PACTMAXW2 WITH PACTUAL IN RUNNING TIME (backtracking scheme).

edgeload = 0.3

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*-0.22	*-0.10	*-0.34	*-0.44	*-0.50	*-2.12	*-8.37	*-9.11
S.D.	0.44	0.41	0.71	0.74	1.33	6.29	17.29	13.50
better	23%	9%	31%	44%	37%	55%	91%	92%
worse	1%	1%	4%	4%	6%	15%	4%	7%

edgeload = 0.5

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*-0.21	*-0.22	*-0.63	*-2.42	*-3.81	*-10.04	*-35.32	-57.25
S.D.	0.48	0.46	1.33	5.77	4.88	33.53	89.48	356.59
better	19%	20%	52%	66%	82%	83%	87%	70%
worse	0%	0%	11%	13%	10%	14%	13%	30%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Pactmaxw2 is better (worse) than Pactual.

3. '*' mean is significant at 0.05 level.

Table LVIII. COMPARISON PACTMAXW2 WITH PACTUAL IN RUNNING TIME (backtracking scheme).

edgeload = 0.7

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	*-0.08	*-0.26	*-0.98	*-2.08	*-4.46	*-15.31	*-54.46	*-412.01
S.D.	0.27	0.60	1.46	4.68	13.78	60.16	83.31	1381.41
better	8%	26%	57%	69%	82%	82%	83%	83%
worse	0%	4%	6%	7%	8%	16%	16%	17%

edgeload = 0.9

	n=28	n=32	n=36	n=40	n=44	n=48	n=52	n=56
mean	0.00	0.00	*-0.27	*-0.17	*-0.58	*-0.84	*-2.61	*-7.82
S.D.	0.00	0.00	0.45	0.38	1.18	3.57	12.70	29.93
better	0%	0%	27%	17%	54%	38%	43%	72%
worse	0%	0%	0%	0%	5%	8%	9%	12%

1. Mean and standard deviation of difference in running time.

2. Percentage of trials in which Pactmaxw2 is better (worse) than Pactual.

3. '*' mean is significant at 0.05 level.

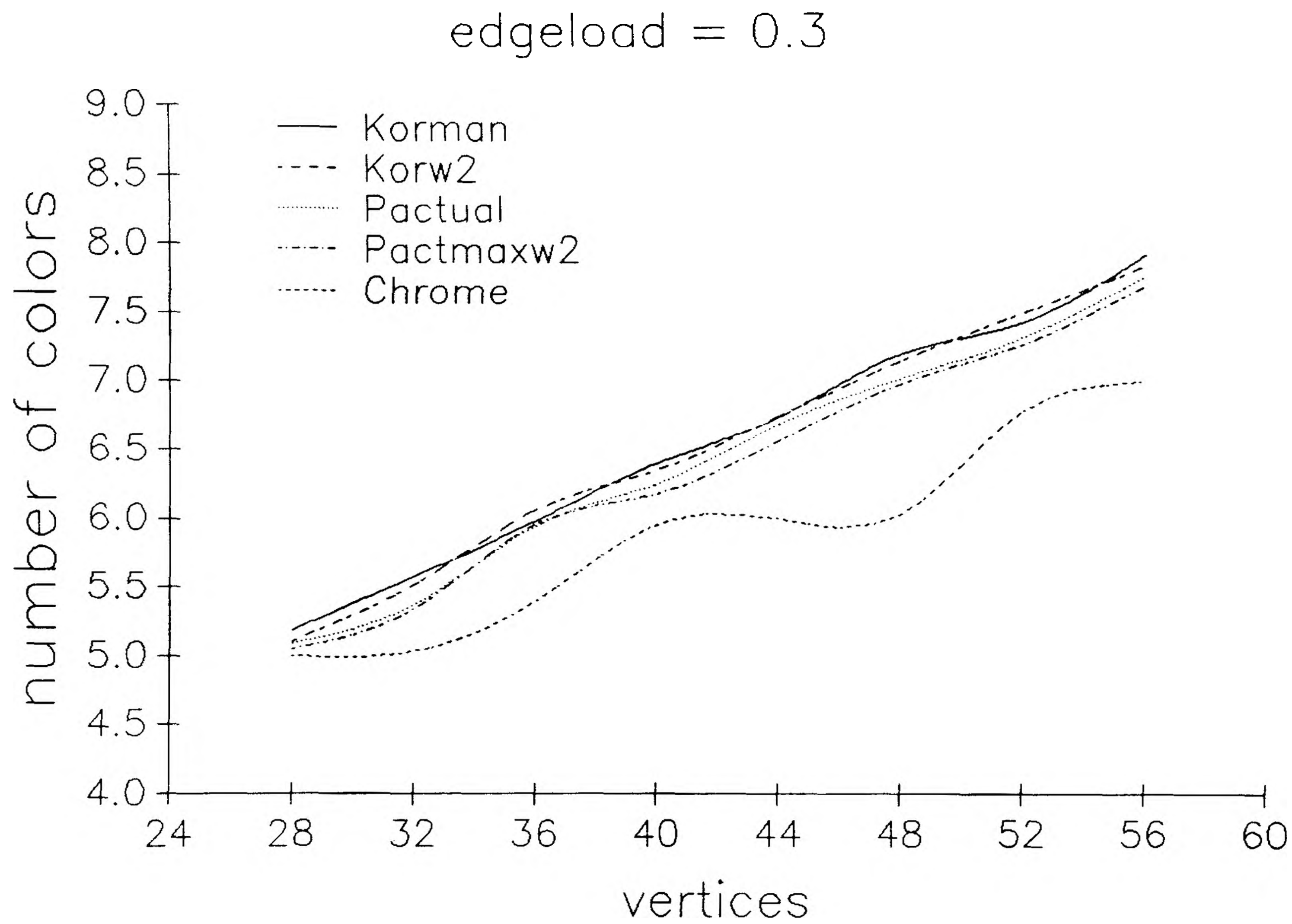


Figure 44. heuristic colors for edgeload = 0.3

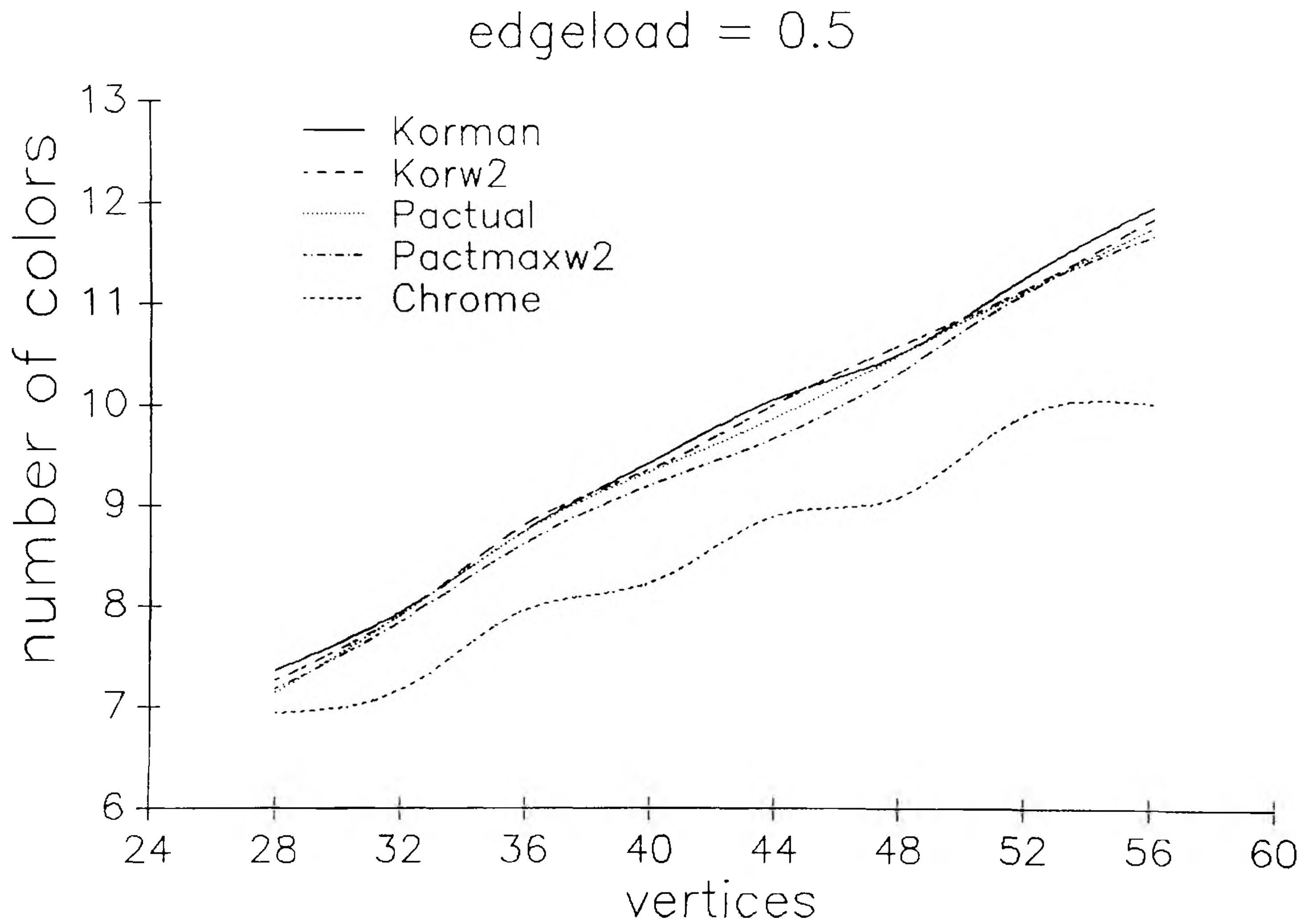


Figure 45. heuristic colors for edgeload = 0.5

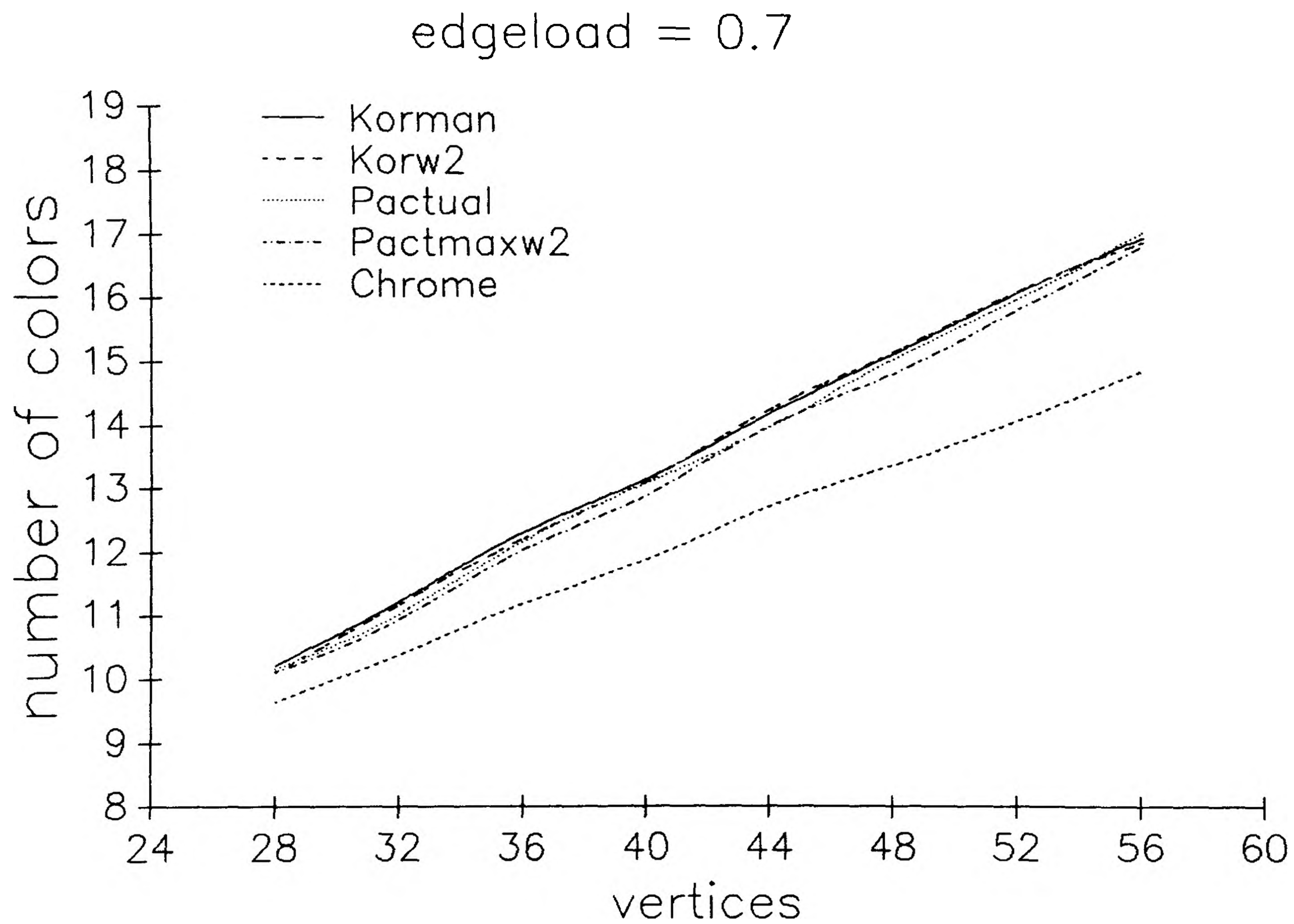


Figure 46. heuristic colors for edgeload = 0.7

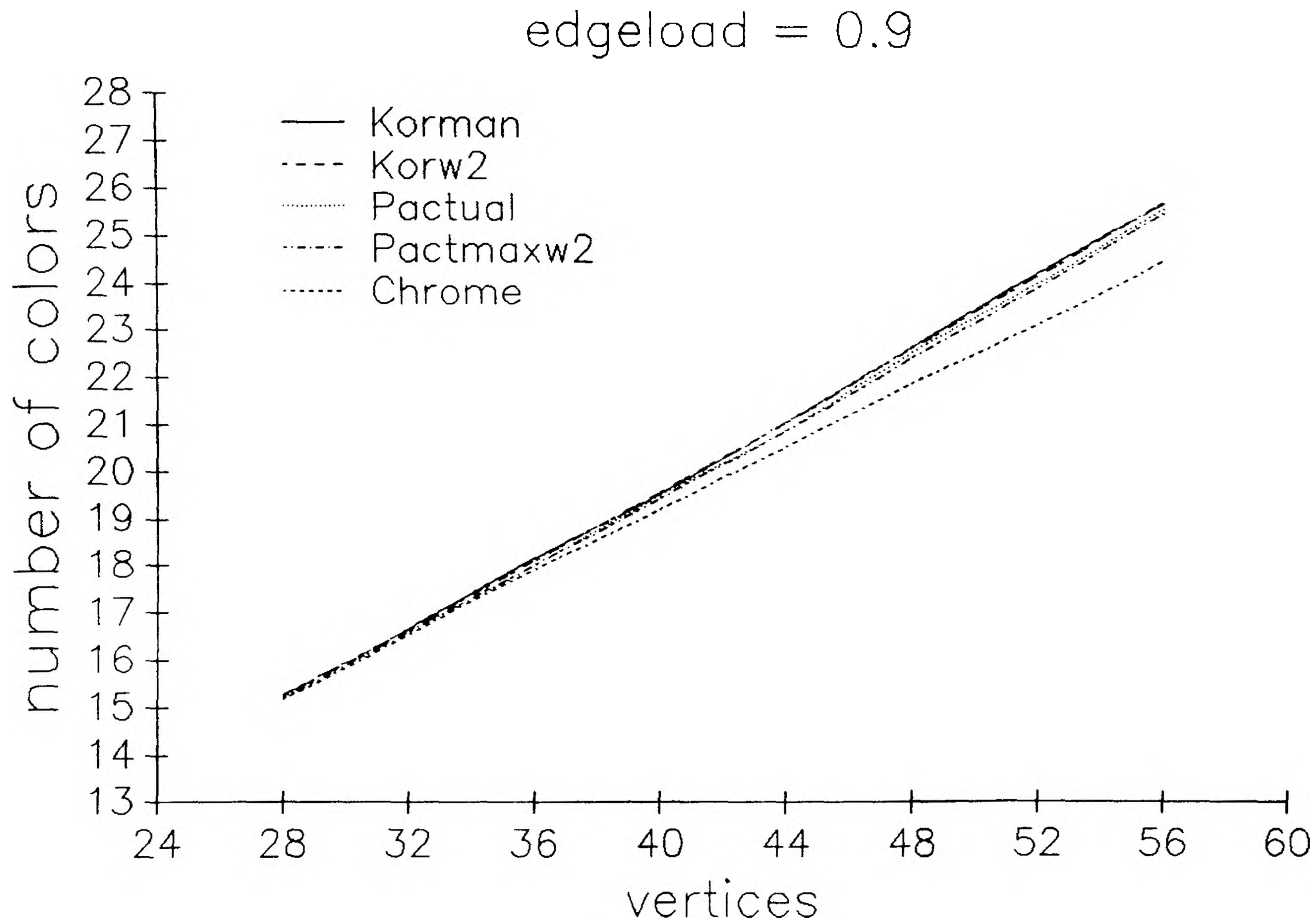


Figure 47. heuristic colors for edgeload = 0.9

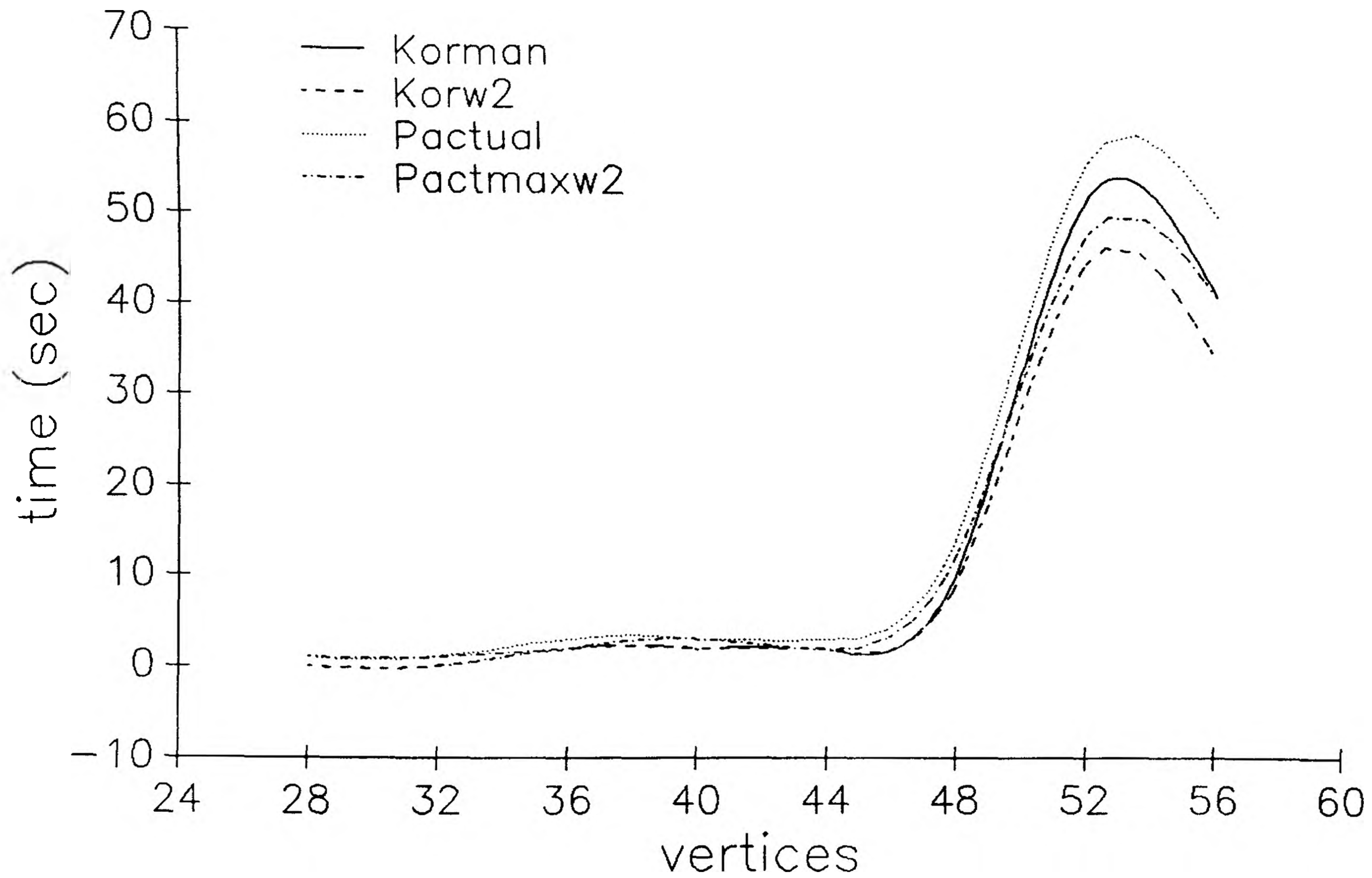


Figure 48. running time for edgeload = 0.3

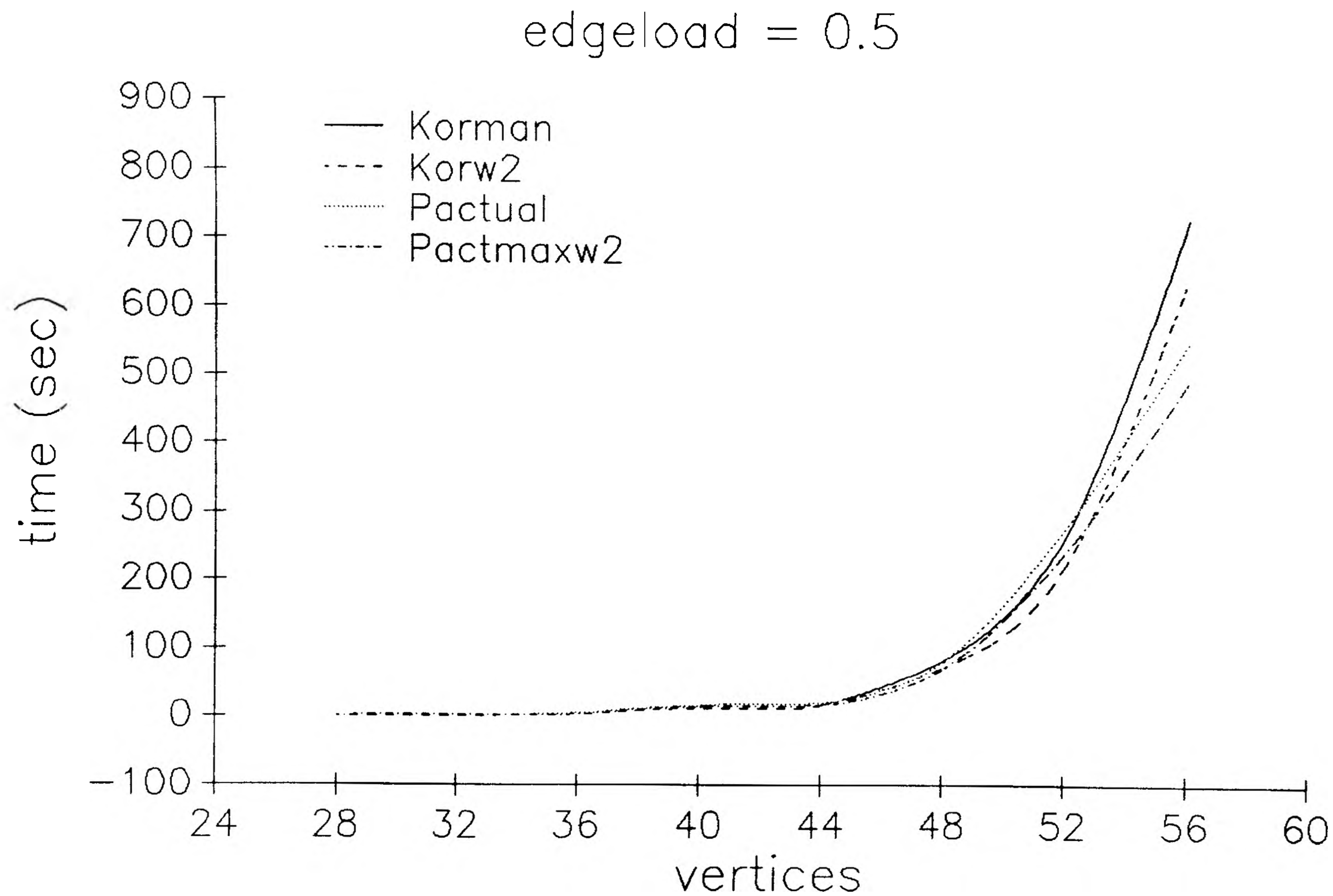


Figure 49. running time for edgeload = 0.5

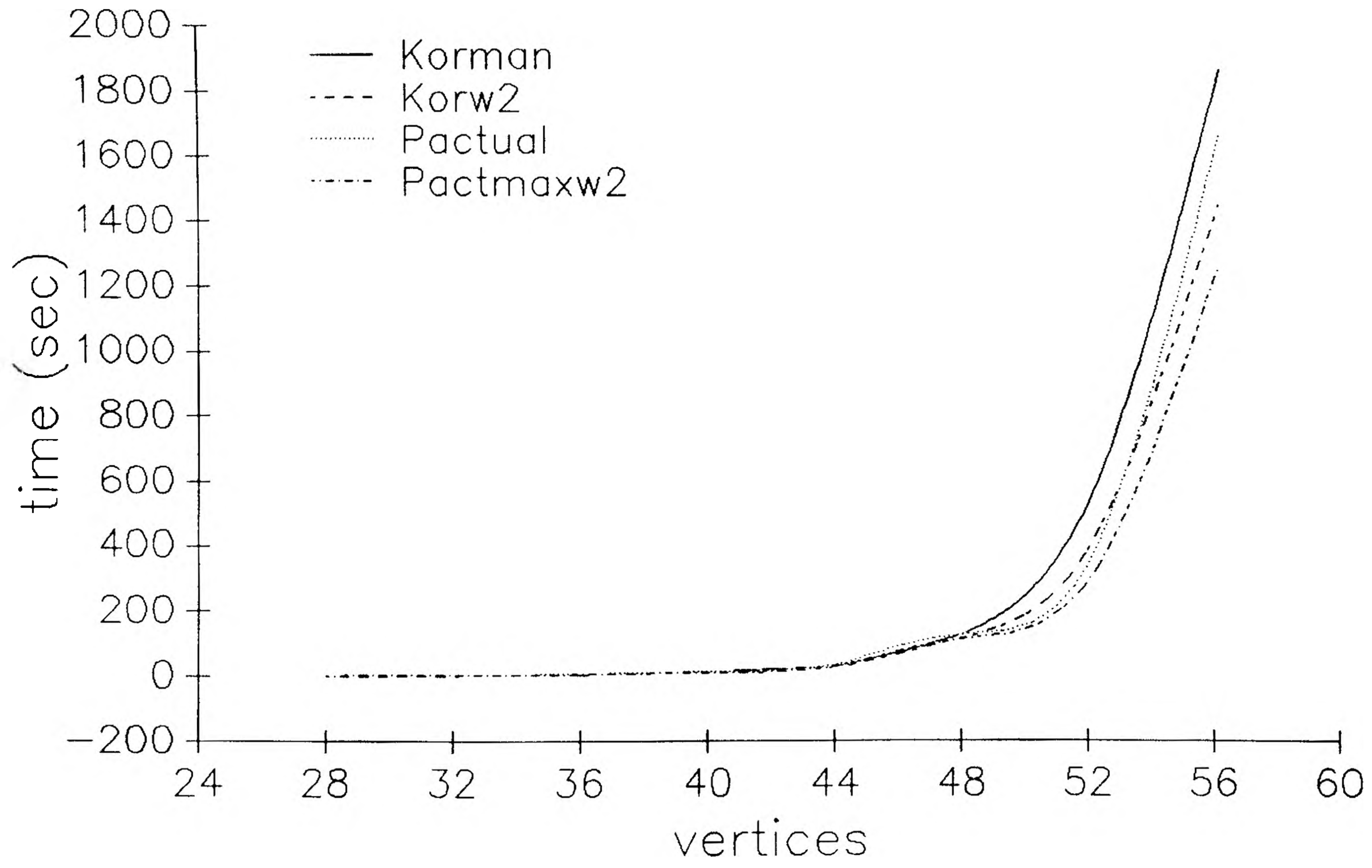


Figure 50. running time for edgeload = 0.7

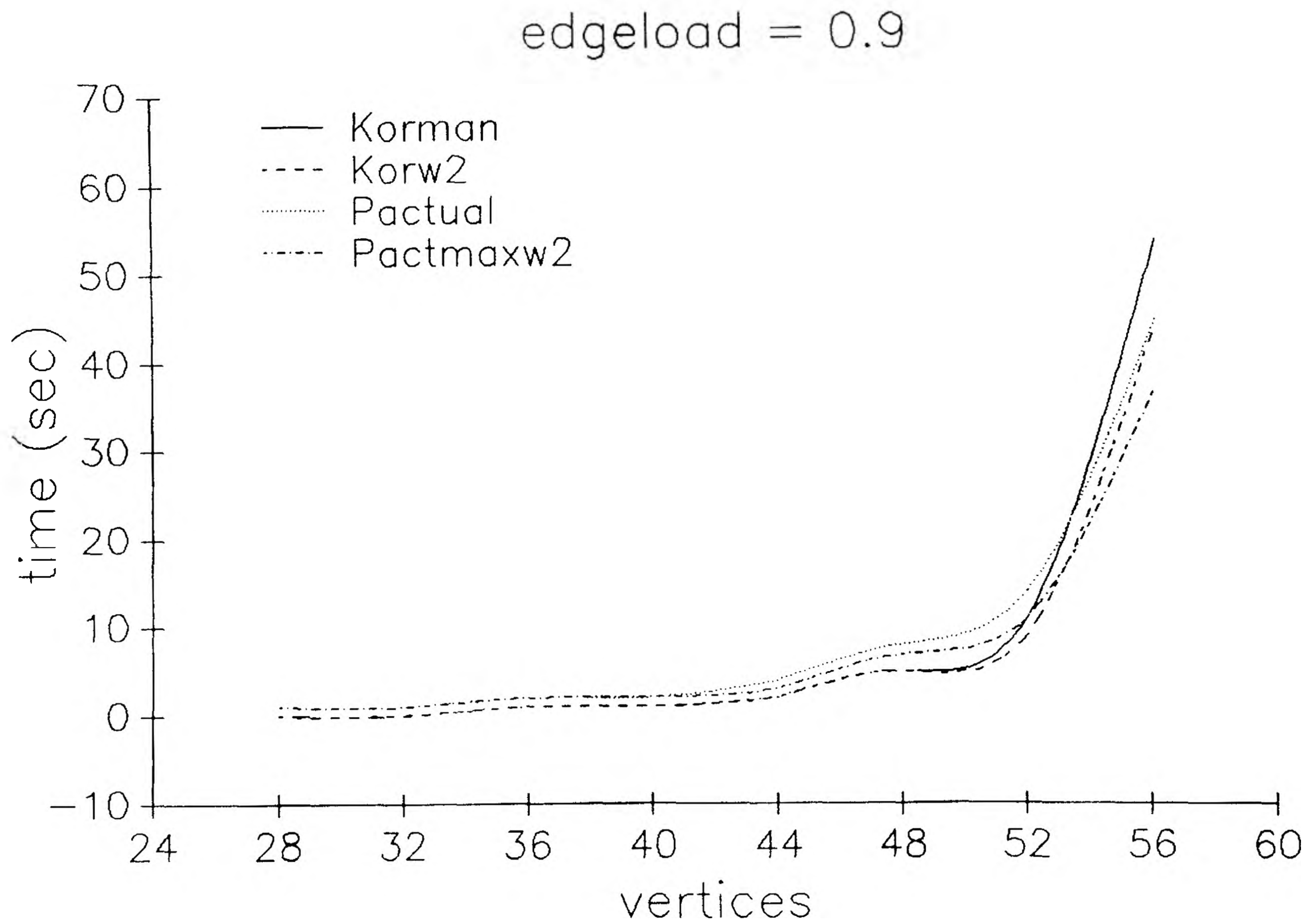


Figure 51. running time for edgeload = 0.9

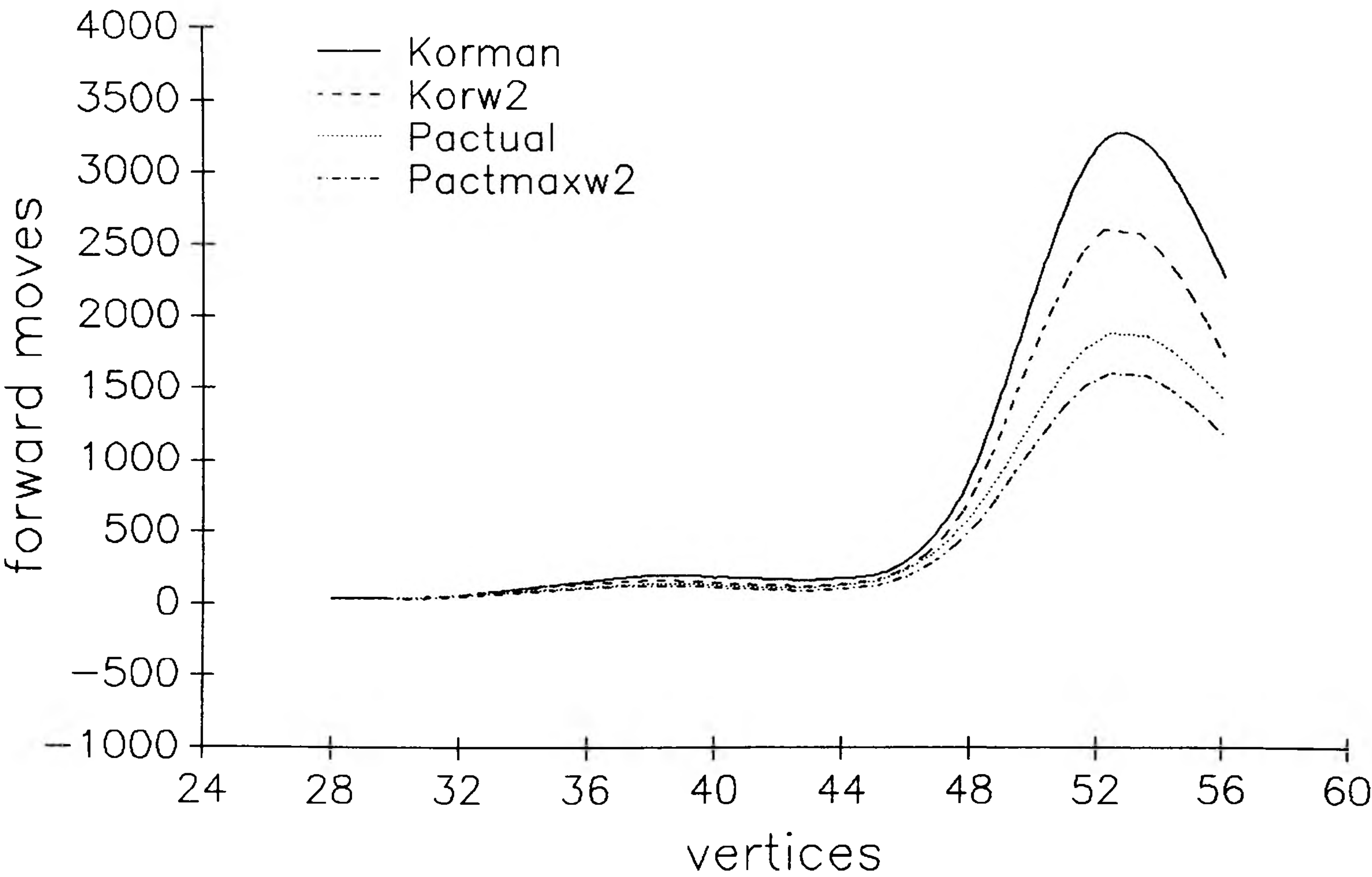


Figure 52. forward moves for edgeload = 0.3

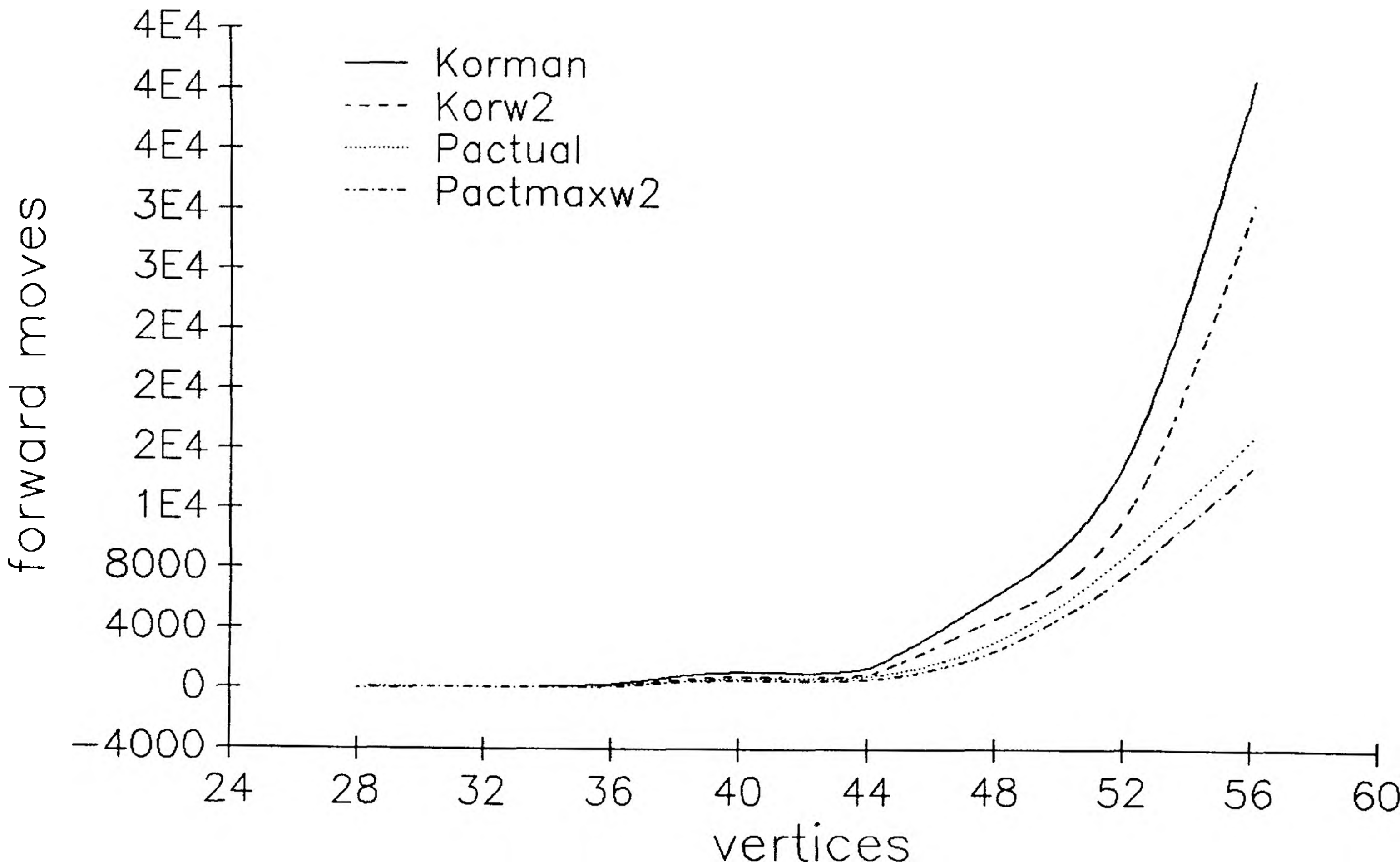


Figure 53. forward moves for edgeload = 0.5

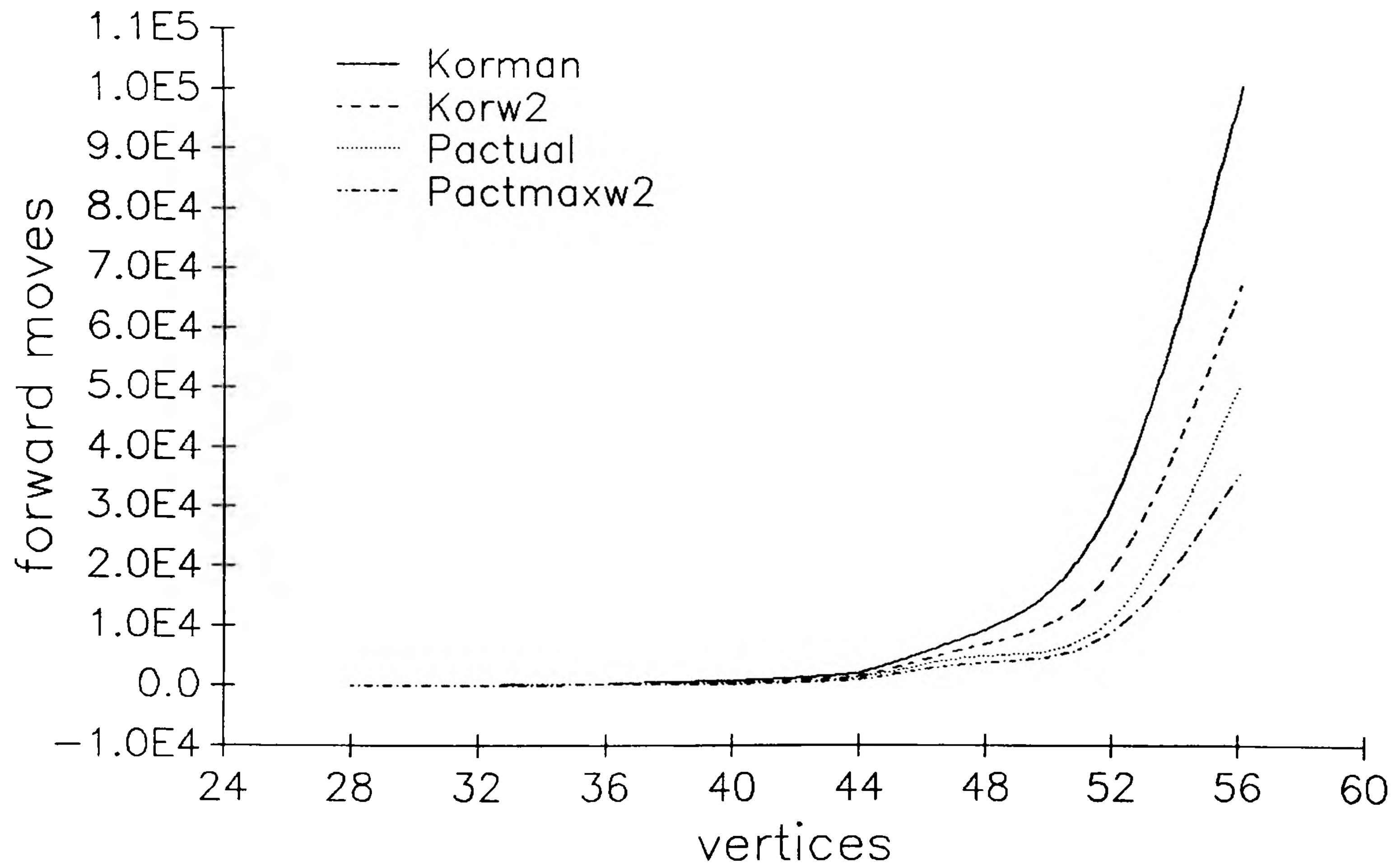


Figure 54. forward moves for edgeload = 0.7

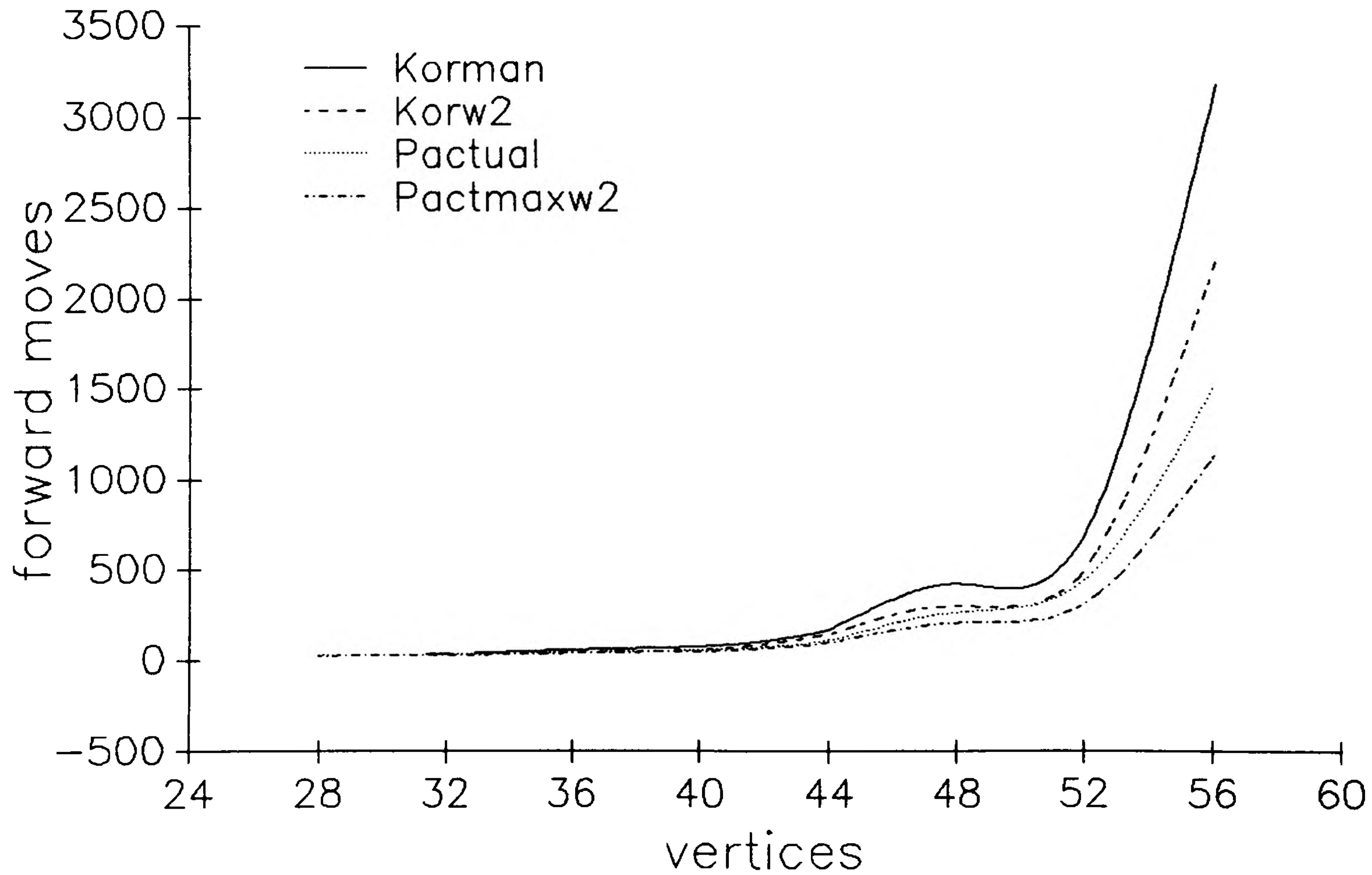


Figure 55. forward moves for edgeload = 0.9

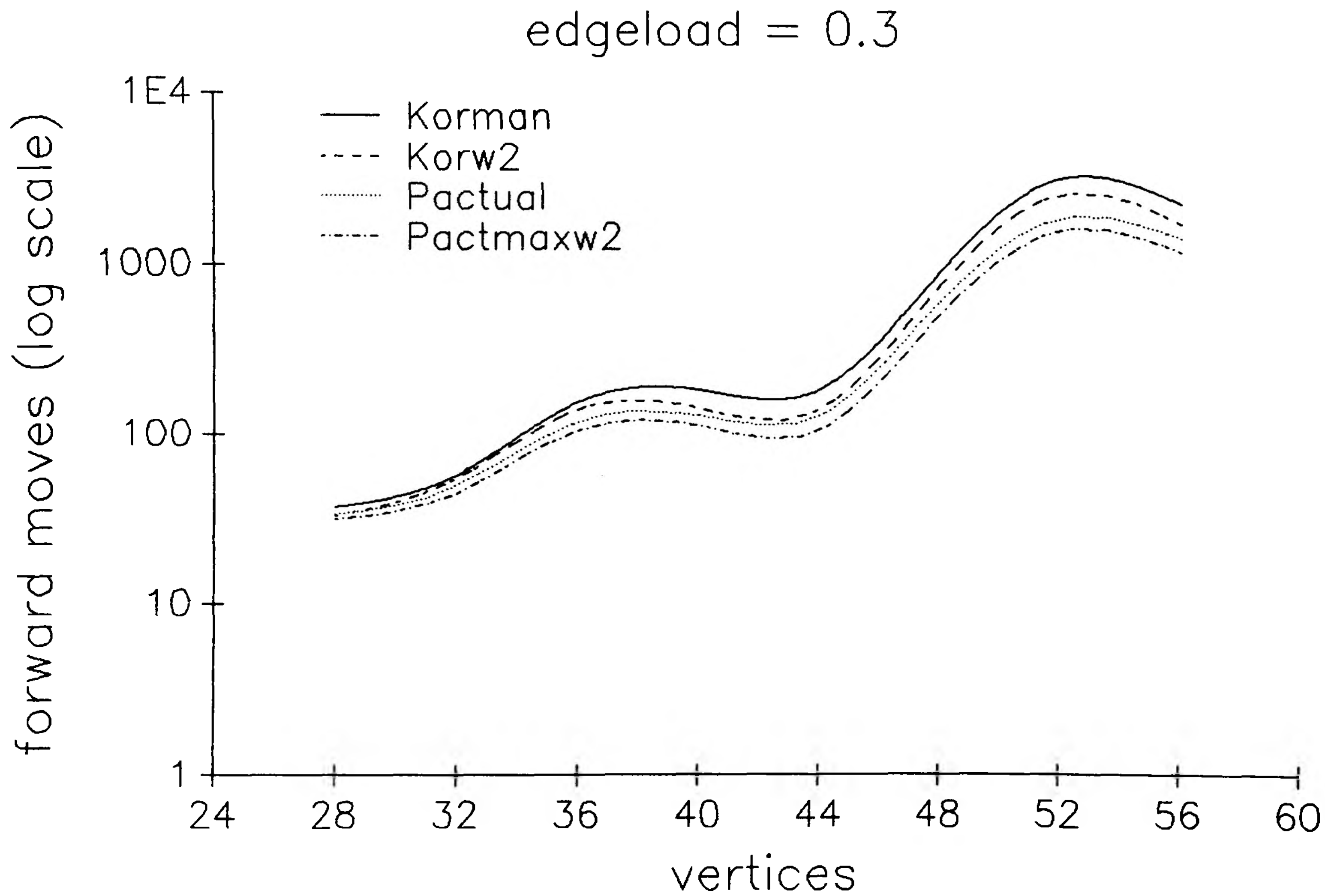


Figure 56. forward moves on log scale for edgeload = 0.3

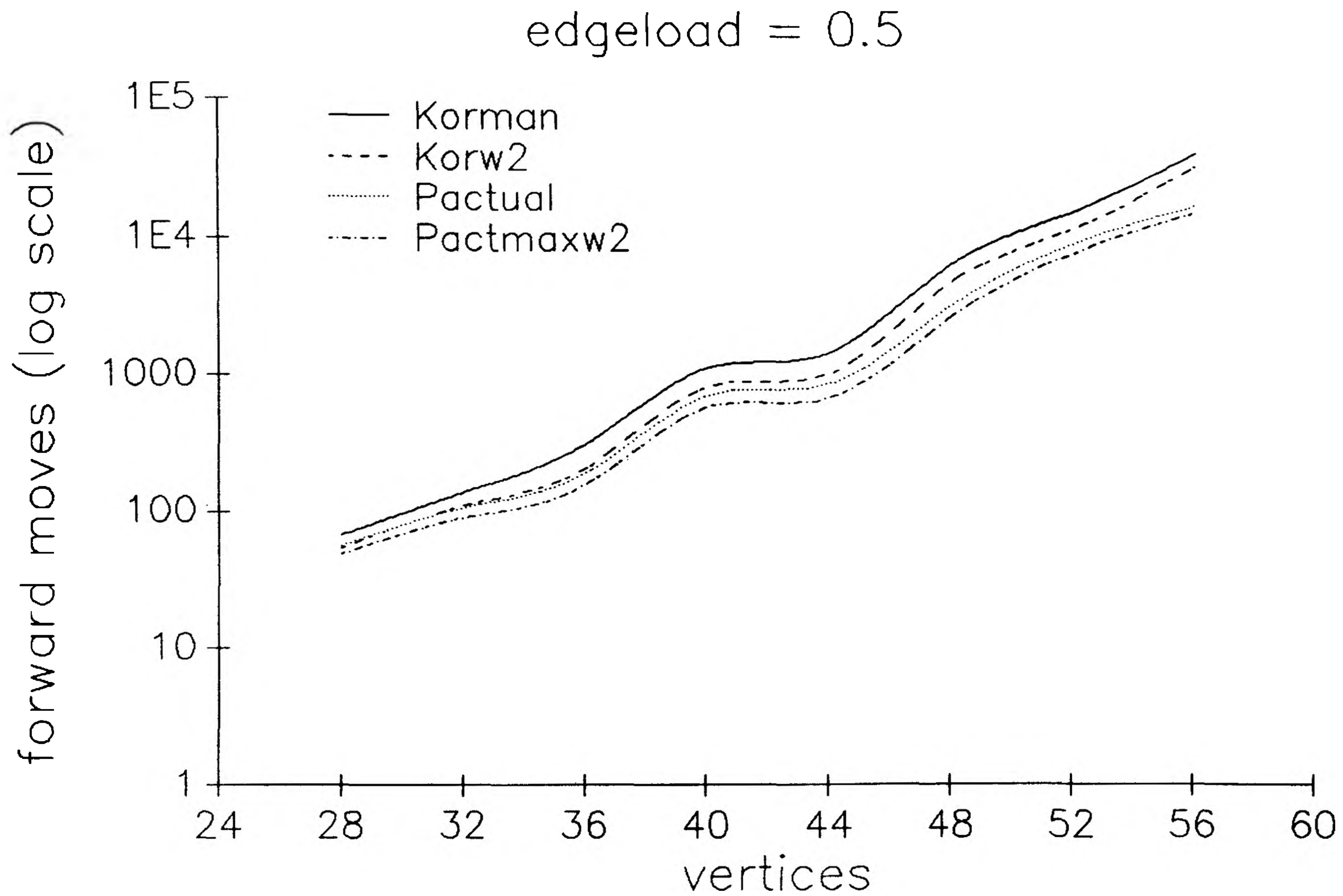


Figure 57. forward moves on log scale for edgeload = 0.5

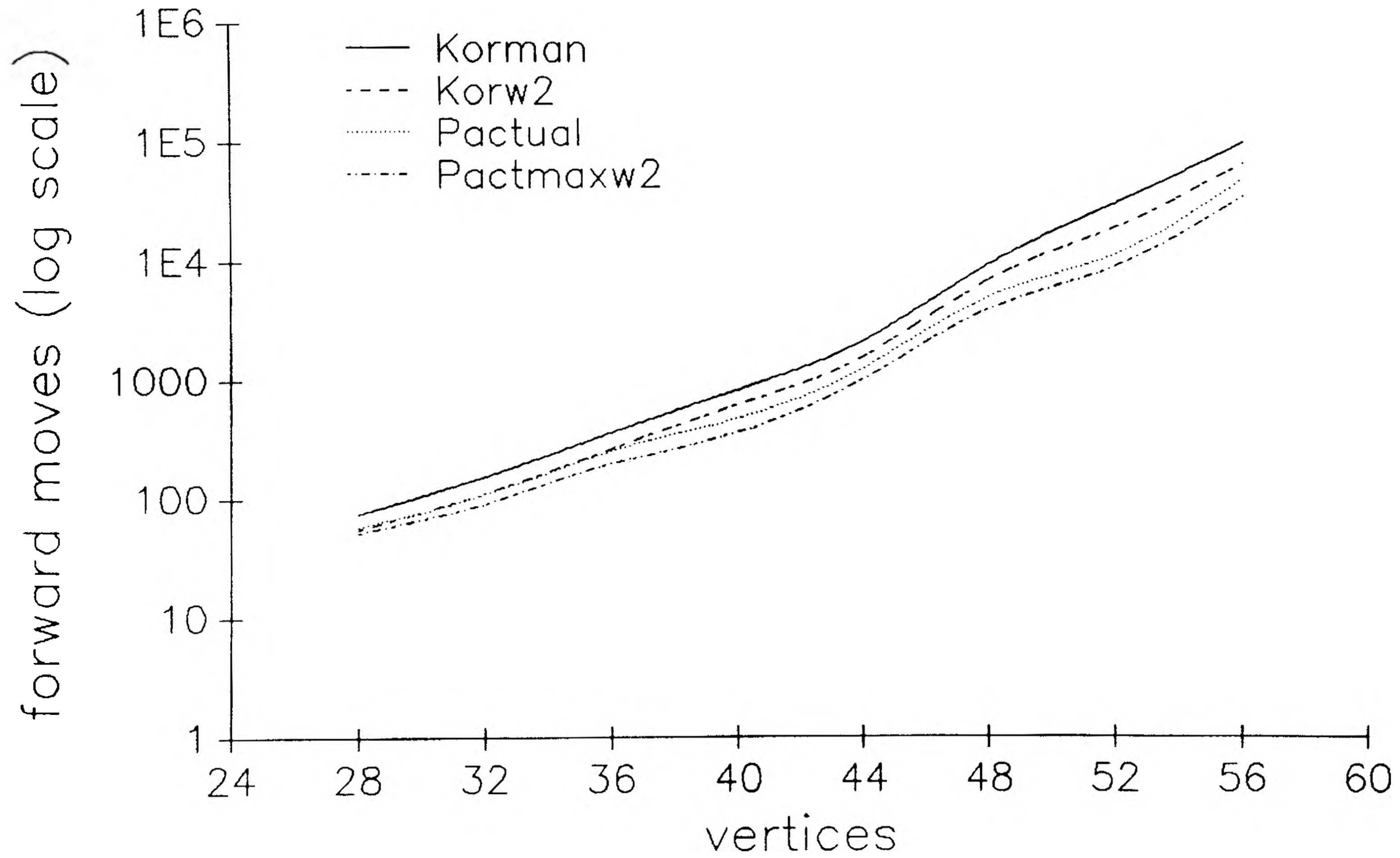


Figure 58. forward moves on log scale for edgeload = 0.7

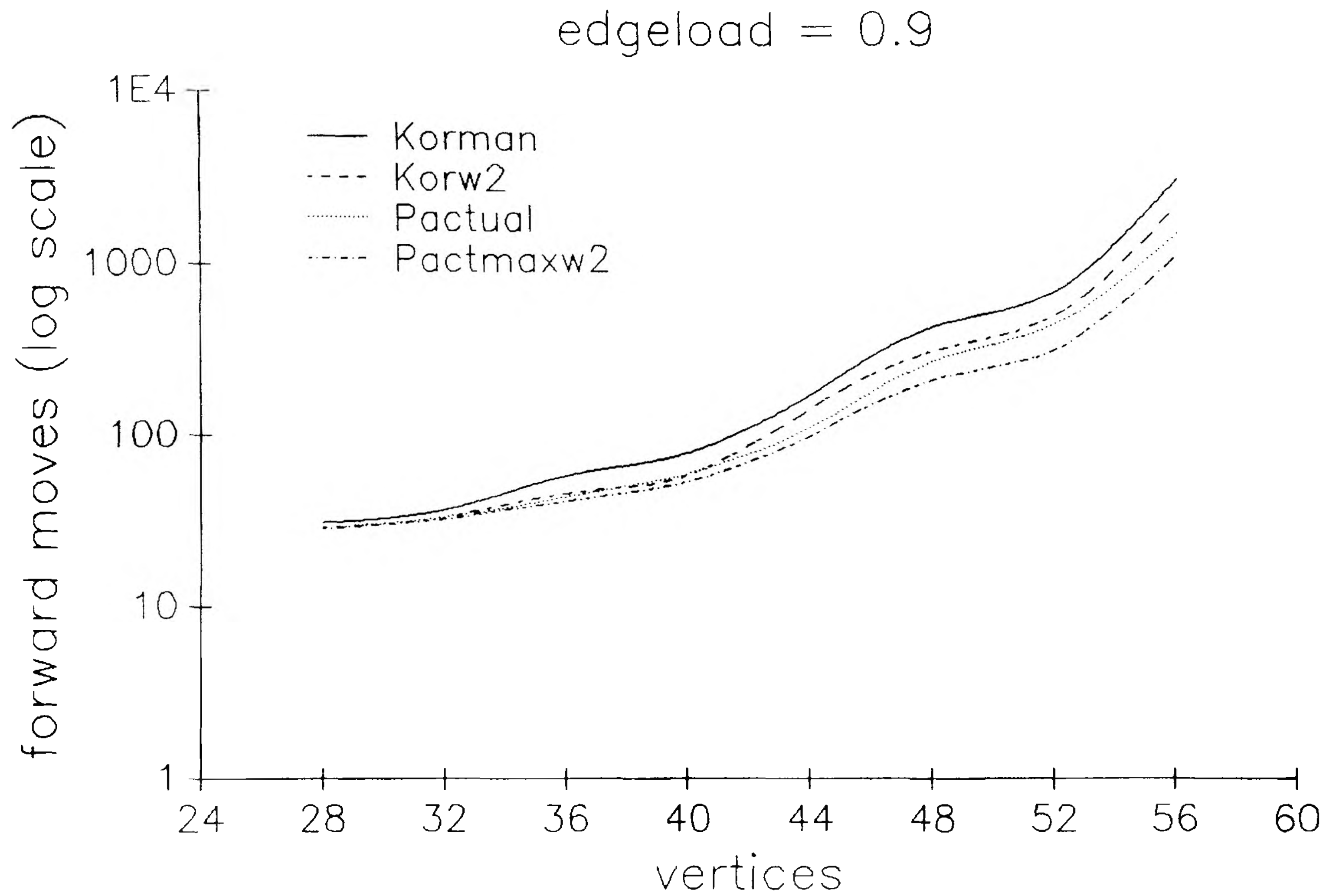


Figure 59. forward moves on log scale for edgeload = 0.9

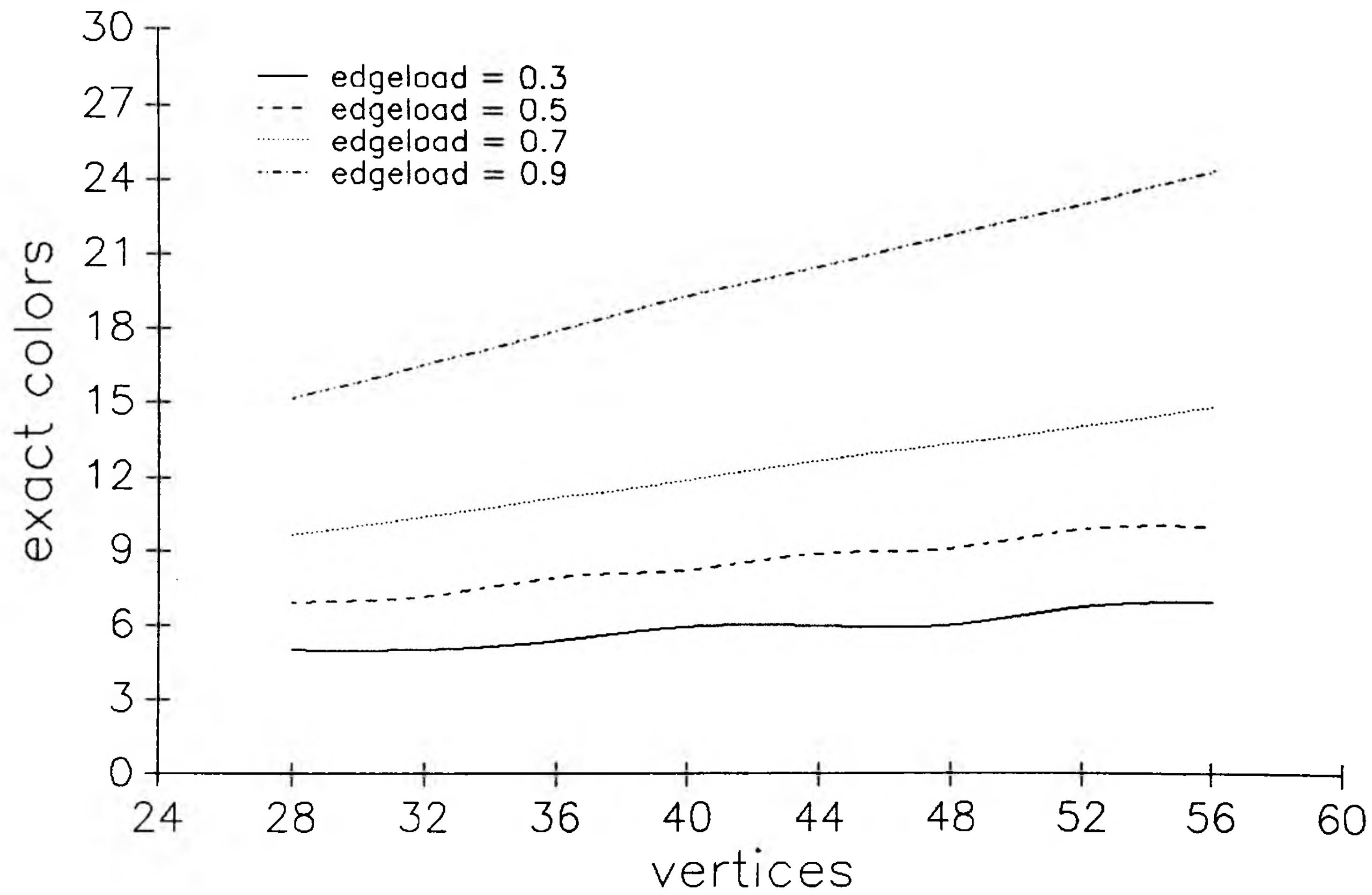


Figure 60. exact colors

Table LIX. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Korman algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ikorman	jkorman	-	5.96	183.86	6.40	2	-	-	-	-
ikorman	jkorlm	2.0	5.96	181.20	6.40	2	100	0	0	0
ikorman	jkorlm	1.9	5.97	147.21		2	99	1	0	0
ikorman	jkorlm	1.8	5.98	132.28		2	98	2	0	0
ikorman	jkorlm	1.7	5.98	122.84		1	98	2	0	0
ikorman	jkorlm	1.6	5.99	95.73		1	97	3	0	0
ikorman	jkorlm	1.5	5.99	87.64		1	97	3	0	0
ikorman	jkorlm	1.4	5.99	75.55		1	97	3	0	0
ikorman	jkorlm	1.3	6.03	67.18		1	93	7	0	0
ikorman	jkorlm	1.2	6.13	59.89		1	83	17	0	0
ikorman	jkorlm	1.1	6.18	52.14		1	78	22	0	0

*** Color column of limit technique is not exact.

Table LX. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Korman algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ikorman	jkorman	-	8.23	1108.45	9.44	15	-	-	-	-
ikorman	jkorlm	2.0	8.26	953.68	9.44	13	97	3	0	0
ikorman	jkorlm	1.9	8.30	792.51		11	93	7	0	0
ikorman	jkorlm	1.8	8.44	532.98		7	79	21	0	0
ikorman	jkorlm	1.7	8.54	432.29		6	69	31	0	0
ikorman	jkorlm	1.6	8.62	309.19		4	61	39	0	0
ikorman	jkorlm	1.5	8.71	228.64		3	53	46	1	0
ikorman	jkorlm	1.4	8.81	174.89		2	43	56	1	0
ikorman	jkorlm	1.3	8.88	115.05		1	35	65	0	0
ikorman	jkorlm	1.2	9.04	87.12		1	25	69	6	0
ikorman	jkorlm	1.1	9.18	58.24		1	21	63	16	0

*** Color column of limit technique is not exact.

Table LXI. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Korman algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ikorman	jkorman	-	11.88	833.55	13.14	11	-	-	-	-
ikorman	jkorlm	2.0	11.88	784.93	13.14	11	100	0	0	0
ikorman	jkorlm	1.9	11.92	630.27		9	96	4	0	0
ikorman	jkorlm	1.8	11.99	521.74		7	89	11	0	0
ikorman	jkorlm	1.7	12.01	433.21		6	87	13	0	0
ikorman	jkorlm	1.6	12.17	331.70		4	71	29	0	0
ikorman	jkorlm	1.5	12.25	260.83		3	64	35	1	0
ikorman	jkorlm	1.4	12.37	178.60		2	54	43	3	0
ikorman	jkorlm	1.3	12.53	126.64		2	38	59	3	0
ikorman	jkorlm	1.2	12.69	85.02		1	28	64	7	1
ikorman	jkorlm	1.1	12.99	55.16		1	16	60	21	3

*** Color column of limit technique is not exact.

Table LXII. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Korman algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ikorman	jkorman	-	19.20	78.83	19.53	1	-	-	-	-
ikorman	jkorlm	2.0	19.20	78.73	19.53	1	100	0	0	0
ikorman	jkorlm	1.9	19.21	75.59		1	99	1	0	0
ikorman	jkorlm	1.8	19.24	71.31		1	96	4	0	0
ikorman	jkorlm	1.7	19.25	68.74		1	95	5	0	0
ikorman	jkorlm	1.6	19.26	64.80		1	94	6	0	0
ikorman	jkorlm	1.5	19.30	61.23		1	90	10	0	0
ikorman	jkorlm	1.4	19.31	57.23		1	89	11	0	0
ikorman	jkorlm	1.3	19.36	51.44		1	84	16	0	0
ikorman	jkorlm	1.2	19.45	47.16		1	75	25	0	0
ikorman	jkorlm	1.1	19.44	44.17		0	77	22	1	0

*** Color column of limit technique is not exact.

Table LXIII. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Pactual algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpactual	-	5.96	129.22	6.25	3	-	-	-	-
ipactual	jpaclm	2.0	5.96	126.79	6.25	3	100	0	0	0
ipactual	jpaclm	1.9	5.97	114.18		3	100	0	0	0
ipactual	jpaclm	1.8	5.98	103.84		3	99	1	0	0
ipactual	jpaclm	1.7	5.98	93.32		2	99	1	0	0
ipactual	jpaclm	1.6	5.99	84.16		2	97	3	0	0
ipactual	jpaclm	1.5	5.99	74.82		2	95	5	0	0
ipactual	jpaclm	1.4	5.99	69.41		2	98	2	0	0
ipactual	jpaclm	1.3	6.03	61.13		2	95	5	0	0
ipactual	jpaclm	1.2	6.13	54.34		1	93	7	0	0
ipactual	jpaclm	1.1	6.18	48.24		1	88	12	0	0

*** Color column of limit technique is not exact.

Table LXIV. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Pactual algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpactual	-	8.23	702.96	9.35	16	-	-	-	-
ipactual	jpaclm	2.0	8.24	594.18	9.35	14	99	1	0	0
ipactual	jpaclm	1.9	8.30	478.19		11	93	7	0	0
ipactual	jpaclm	1.8	8.43	396.20		9	80	20	0	0
ipactual	jpaclm	1.7	8.49	327.42		8	74	26	0	0
ipactual	jpaclm	1.6	8.51	230.38		6	72	28	0	0
ipactual	jpaclm	1.5	8.66	188.70		5	58	41	1	0
ipactual	jpaclm	1.4	8.77	127.46		3	46	54	0	0
ipactual	jpaclm	1.3	8.83	105.58		3	41	58	1	0
ipactual	jpaclm	1.2	8.95	72.74		2	30	68	2	0
ipactual	jpaclm	1.1	9.16	55.33		2	21	65	14	0

*** Color column of limit technique is not exact.

Table LXV. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Pactual algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpactual	-	11.88	479.94	13.08	12	-	-	-	-
ipactual	jpaclm	2.0	11.88	438.69	13.08	11	100	0	0	0
ipactual	jpaclm	1.9	11.92	367.54		9	96	4	0	0
ipactual	jpaclm	1.8	11.95	312.41		8	93	7	0	0
ipactual	jpaclm	1.7	12.03	309.02		8	85	15	0	0
ipactual	jpaclm	1.6	12.06	222.93		6	82	18	0	0
ipactual	jpaclm	1.5	12.17	181.35		5	71	29	0	0
ipactual	jpaclm	1.4	12.20	150.44		4	68	32	0	0
ipactual	jpaclm	1.3	12.51	110.41		3	41	55	4	0
ipactual	jpaclm	1.2	12.60	76.59		2	37	54	9	0
ipactual	jpaclm	1.1	12.82	56.79		2	24	58	18	0

*** Color column of limit technique is not exact.

Table LXVI. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Pactual algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpactual	-	19.20	60.31	19.50	2	-	-	-	-
ipactual	jpaclm	2.0	19.20	60.73	19.50	2	100	0	0	0
ipactual	jpaclm	1.9	19.22	59.17		2	98	2	0	0
ipactual	jpaclm	1.8	19.23	59.17		2	97	3	0	0
ipactual	jpaclm	1.7	19.25	55.91		2	95	5	0	0
ipactual	jpaclm	1.6	19.22	54.84		2	98	2	0	0
ipactual	jpaclm	1.5	19.24	51.88		2	96	4	0	0
ipactual	jpaclm	1.4	19.28	50.43		2	92	8	0	0
ipactual	jpaclm	1.3	19.30	47.97		2	90	10	0	0
ipactual	jpaclm	1.2	19.40	44.11		2	80	20	0	0
ipactual	jpaclm	1.1	19.42	43.08		2	78	22	0	0

*** Color column of limit technique is not exact.

Table LXVII. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Prevent4a algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprevent4a	jprevent4a	-	5.96	111.01	6.26	3	-	-	-	-
iprv4a	jpv4lm	2.0	5.96	109.75	6.26	3	100	0	0	0
iprv4a	jpv4lm	1.9	5.96	99.82		3	100	0	0	0
iprv4a	jpv4lm	1.8	5.97	89.77		2	99	1	0	0
iprv4a	jpv4lm	1.7	5.98	84.61		2	98	2	0	0
iprv4a	jpv4lm	1.6	5.99	75.20		2	97	3	0	0
iprv4a	jpv4lm	1.5	6.00	66.78		2	96	4	0	0
iprv4a	jpv4lm	1.4	6.01	63.33		2	95	5	0	0
iprv4a	jpv4lm	1.3	6.00	58.00		2	97	2	1	0
iprv4a	jpv4lm	1.2	6.06	51.91		1	90	10	0	0
iprv4a	jpv4lm	1.1	6.13	47.08		1	83	17	0	0

*** Color column of limit technique is not exact.

Table LXVIII. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Prevent4a algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprevent4a	jprevent4a	-	8.23	631.19	9.35	15	-	-	-	-
iprv4a	jpv4lm	2.0	8.25	529.16	9.35	13	98	2	0	0
iprv4a	jpv4lm	1.9	8.33	397.95		10	90	10	0	0
iprv4a	jpv4lm	1.8	8.35	320.81		8	88	12	0	0
iprv4a	jpv4lm	1.7	8.48	291.73		7	75	25	0	0
iprv4a	jpv4lm	1.6	8.56	219.60		6	68	31	1	0
iprv4a	jpv4lm	1.5	8.65	166.53		4	58	42	0	0
iprv4a	jpv4lm	1.4	8.78	120.59		3	47	51	2	0
iprv4a	jpv4lm	1.3	8.87	100.86		3	38	60	2	0
iprv4a	jpv4lm	1.2	8.90	77.63		2	33	67	0	0
iprv4a	jpv4lm	1.1	9.09	54.67		2	26	62	12	0

*** Color column of limit technique is not exact.

Table LXIX. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Prevent4a algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprevent4a	jprevent4a	-	11.88	532.47	13.31	13	-	-	-	-
iprv4a	jpv4lm	2.0	11.89	497.11	13.31	12	99	1	0	0
iprv4a	jpv4lm	1.9	11.95	452.62		11	93	7	0	0
iprv4a	jpv4lm	1.8	11.98	351.54		9	90	10	0	0
iprv4a	jpv4lm	1.7	12.04	314.78		8	84	16	0	0
iprv4a	jpv4lm	1.6	12.18	234.28		6	70	30	1	0
iprv4a	jpv4lm	1.5	12.26	175.92		5	67	28	5	0
iprv4a	jpv4lm	1.4	12.44	140.39		4	50	45	4	1
iprv4a	jpv4lm	1.3	12.61	101.48		3	36	56	7	1
iprv4a	jpv4lm	1.2	12.79	78.17		2	29	53	16	2
iprv4a	jpv4lm	1.1	12.99	58.44		2	18	56	23	3

*** Color column of limit technique is not exact.

Table LXX. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Prevent4a algorithm with limit technique on different values of parameter lim

nucv	ncv	lim	color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprevent4a	jprevent4a	-	19.20	59.34	19.52	3	-	-	-	-
iprv4a	jpv4lm	2.0	19.20	59.34	19.52	3	100	0	0	0
iprv4a	jpv4lm	1.9	19.22	59.37		3	98	2	0	0
iprv4a	jpv4lm	1.8	19.21	57.50		3	99	1	0	0
iprv4a	jpv4lm	1.7	19.24	54.53		3	96	4	0	0
iprv4a	jpv4lm	1.6	19.28	51.74		3	92	8	0	0
iprv4a	jpv4lm	1.5	19.28	51.67		3	92	8	0	0
iprv4a	jpv4lm	1.4	19.38	47.70		3	82	18	0	0
iprv4a	jpv4lm	1.3	19.41	47.32		3	79	21	0	0
iprv4a	jpv4lm	1.2	19.39	44.82		3	82	17	1	0
iprv4a	jpv4lm	1.1	19.47	42.30		3	73	27	0	0

*** Color column of limit technique is not exact.

Table LXXI. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Pactual algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpacnh	0.3	1.0	5.96	129.22	6.25	3	100	0	0	0
ipactual	jpacnh		0.9	5.96	129.22		3	100	0	0	0
ipactual	jpacnh		0.8	5.96	128.53		3	100	0	0	0
ipactual	jpacnh		0.7	5.96	124.20		3	100	0	0	0
ipactual	jpacnh		0.6	5.97	115.41		3	99	1	0	0
ipactual	jpacnh		0.5	5.97	101.95		3	99	1	0	0
ipactual	jpacnh		0.4	5.97	83.55		2	99	1	0	0
ipactual	jpacnh		0.3	5.98	74.70		2	98	2	0	0
ipactual	jpacnh		0.2	5.98	59.61		2	98	2	0	0
ipactual	jpacnh		0.1	6.07	49.91		1	89	11	0	0

Table LXXII. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Pactual algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpacnh	0.5	1.0	8.23	702.96	9.35	17	100	0	0	0
ipactual	jpacnh		0.9	8.23	702.15		17	100	0	0	0
ipactual	jpacnh		0.8	8.25	672.20		17	98	2	0	0
ipactual	jpacnh		0.7	8.30	522.32		13	93	7	0	0
ipactual	jpacnh		0.6	8.48	366.15		9	75	25	0	0
ipactual	jpacnh		0.5	8.65	209.67		5	58	42	0	0
ipactual	jpacnh		0.4	8.84	129.26		3	44	51	5	0
ipactual	jpacnh		0.3	9.00	90.25		2	31	61	8	0
ipactual	jpacnh		0.2	9.09	64.18		2	27	60	13	0
ipactual	jpacnh		0.1	9.21	51.05		1	19	64	17	0

Table LXXIII. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Pactual algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpacnh	0.7	1.0	11.88	479.94	13.08	12	100	0	0	0
ipactual	jpacnh		0.9	11.89	445.24		11	99	1	0	0
ipactual	jpacnh		0.8	11.96	331.52		8	92	8	0	0
ipactual	jpacnh		0.7	12.15	179.18		5	74	25	1	0
ipactual	jpacnh		0.6	12.41	98.31		3	50	47	3	0
ipactual	jpacnh		0.5	12.66	72.39		2	37	50	11	2
ipactual	jpacnh		0.4	12.80	58.85		2	30	50	18	2
ipactual	jpacnh		0.3	12.85	55.61		2	26	53	19	2
ipactual	jpacnh		0.2	12.91	51.31		2	22	55	21	2
ipactual	jpacnh		0.1	12.99	45.10		2	19	53	26	2

Table LXXIV. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Pactual algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
ipactual	jpacnh	0.9	1.0	19.20	60.31	19.50	2	100	0	0	0
ipactual	jpacnh		0.9	19.22	54.27		2	98	2	0	0
ipactual	jpacnh		0.8	19.34	46.84		2	86	14	0	0
ipactual	jpacnh		0.7	19.41	43.80		2	79	21	0	0
ipactual	jpacnh		0.6	19.44	42.65		2	76	24	0	0
ipactual	jpacnh		0.5	19.44	42.53		2	76	24	0	0
ipactual	jpacnh		0.4	19.44	42.53		2	76	24	0	0
ipactual	jpacnh		0.3	19.44	42.49		2	76	24	0	0
ipactual	jpacnh		0.2	19.44	42.20		2	76	24	0	0
ipactual	jpacnh		0.1	19.45	41.39		2	75	25	0	0

Table LXXV. VERTICES = 40, EDGELOAD = 0.3 (backtracking scheme). Prevent4a algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprv4a	jpv4nh	0.3	1.0	5.96	111.01	6.26	3	100	0	0	0
iprv4a	jpv4nh		0.9	5.96	110.79		3	100	0	0	0
iprv4a	jpv4nh		0.8	5.96	110.31		3	100	0	0	0
iprv4a	jpv4nh		0.7	5.97	106.78		3	99	1	0	0
iprv4a	jpv4nh		0.6	5.97	102.69		3	99	1	0	0
iprv4a	jpv4nh		0.5	5.97	91.41		3	99	1	0	0
iprv4a	jpv4nh		0.4	5.97	78.08		2	99	1	0	0
iprv4a	jpv4nh		0.3	5.97	66.79		2	99	1	0	0
iprv4a	jpv4nh		0.2	5.98	57.24		2	95	5	0	0
iprv4a	jpv4nh		0.1	6.11	47.45		1	85	15	0	0

Table LXXVI. VERTICES = 40, EDGELOAD = 0.5 (backtracking scheme). Prevent4a algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprv4a	jpv4nh	0.5	1.0	8.23	631.19	9.35	17	100	0	0	0
iprv4a	jpv4nh		0.9	8.23	626.65		17	100	0	0	0
iprv4a	jpv4nh		0.8	8.23	612.64		16	100	0	0	0
iprv4a	jpv4nh		0.7	8.24	567.95		15	99	1	0	0
iprv4a	jpv4nh		0.6	8.32	449.68		12	91	9	0	0
iprv4a	jpv4nh		0.5	8.59	216.44		6	64	36	0	0
iprv4a	jpv4nh		0.4	8.79	120.09		3	47	50	3	0
iprv4a	jpv4nh		0.3	8.96	74.37		2	33	61	6	0
iprv4a	jpv4nh		0.2	9.06	61.59		2	29	59	12	0
iprv4a	jpv4nh		0.1	9.20	50.51		2	23	57	20	0

Table LXXVII. VERTICES = 40, EDGELOAD = 0.7 (backtracking scheme). Prevent4a algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprv4a	jpv4nh	0.7	1.0	11.88	532.47	13.31	14	100	0	0	0
iprv4a	jpv4nh		0.9	11.88	522.30		14	100	0	0	0
iprv4a	jpv4nh		0.8	11.90	468.44		13	98	2	0	0
iprv4a	jpv4nh		0.7	12.06	264.80		7	82	18	0	0
iprv4a	jpv4nh		0.6	12.43	113.66		3	50	45	5	0
iprv4a	jpv4nh		0.5	12.86	70.81		2	28	48	22	2
iprv4a	jpv4nh		0.4	13.04	55.16		2	20	48	28	4
iprv4a	jpv4nh		0.3	13.12	51.59		2	18	44	34	4
iprv4a	jpv4nh		0.2	13.16	49.19		2	18	42	34	6
iprv4a	jpv4nh		0.1	13.24	44.83		2	16	39	38	7

Table LXXVIII. VERTICES = 40, EDGELOAD = 0.9 (backtracking scheme). Prevent4a algorithm with epsilon technique on different values of parameter eps

nucv	ncv	edge load	eps	heur. color	moves	first color	time (sec)	0-ap	1-ap	2-ap	3-ap
iprv4a	jpv4nh	0.9	1.0	19.20	59.34	19.52	3	100	0	0	0
iprv4a	jpv4nh		0.9	19.22	55.35		3	98	2	0	0
iprv4a	jpv4nh		0.8	19.37	44.62		3	83	17	0	0
iprv4a	jpv4nh		0.7	19.43	42.28		3	77	23	0	0
iprv4a	jpv4nh		0.6	19.47	41.24		3	74	25	1	0
iprv4a	jpv4nh		0.5	19.47	41.24		3	74	25	1	0
iprv4a	jpv4nh		0.4	19.47	41.24		3	74	25	1	0
iprv4a	jpv4nh		0.3	19.47	41.24		3	74	25	1	0
iprv4a	jpv4nh		0.2	19.47	41.20		3	74	25	1	0
iprv4a	jpv4nh		0.1	19.49	40.66		3	72	27	1	0

VII. A BRANCH-AND-BOUND PREFERENCE FUNCTION

Combining a pair of nucv-selection and ncv-selection functions mentioned in chapter 6 with the branch-and-bound scheme mentioned in chapter 4 forms a heuristic algorithm. How can we build a branch-and-bound preference function? From Figure 60, we find that the "exact color" is nearly a *linear* function of the number of vertices. Thus, we construct a preference function, every datum of which is the mean of the exact colors of a sequence of random graphs are stored in tabular form, and then use the *linear* interploation to find the value of the preference function on a given graph. In this chapter, we describe a branch-and-bound preference function, and present computational results under the branch-and-bound scheme.

A. CONSTRUCTION OF A PREFERENCE FUNCTION

For a c-partially colored graph $G' = (V', E')$, the *preference function* $P(n, c, m)$, where $n = |V'|$, and $m = |E'| - \frac{c(c-1)}{2}$, is defined to be the difference between the expected-colors of G' and c. The *expected-colors* of G' is the mean value of a sequence of chromatic numbers of random graphs having the following properties: each of them has $|V'|$ vertices, a completely connected subgraph G'' of order c, and $|E'| - \frac{c(c-1)}{2}$ edges in addition to those that are in G''. The larger the value of the preference function is, the less the degree of preference is. In order to compute the value $P(n, c, m)$, we interpolate linearly along each parameter of the preference function P, whose data are stored in tabular form. We are going to describe the construction of the preference function by two steps. First we portray how to create the reference table of the preference function. Second we depict how to interpolate the reference table for a partially colored graph.

There are three parameters for the preference function P. While considering the first parameter, the number of vertices, we start at 0 and increase it by *vertex-increment* each time. Because every exact algorithm we developed takes $O(2^{|V|})$ amount of time to get the chromatic number, we limit the number of vertices in order to obtain the reference table within a reasonable amount of computational time. For a given number of vertices, n, mentioned above, the order of any possible completely connected subgraph is between 0 and n. When taking a closer look, we discover that the case, c = 0, only occurs at the beginning movement in the search tree, and the case, c = 1, can be converted to another case, c = 2, by making a minor adjustment. Also P(n, n, m) is always equal to 0. Thus, we divide the interval [2 .. n − 1] into *core-shares* subintervals. For a given number of vertices, n, and a given order, c, of a completely connected subgraph, the number of edges except those that are in the given completely connected subgraph is within the interval $[0 .. \frac{v(v-1)}{2} - \frac{c(c-1)}{2}]$. For any case, m < c, p(n, c, m) = 0 because no new color will be introduced during the remaining part of coloring. Also $p(n, c, \frac{v(v-1)}{2} - \frac{c(c-1)}{2}) = n - c$ because the original graph is a complete graph of order n. Consequently, we divide the interval $[c - 1 .. \frac{v(v-1)}{2} - 1 - \frac{c(c-1)}{2}]$ into *edge-shares* subintervals. That is, there are usually (edge-shares + 1) data entries in the reference table for both a graph of fixed order and a completely connected subgraph of fixed order of the given graph, and (core-shares + 1) * (edge-shares + 1) data entries for a graph of fixed order. If edge-shares > $(\frac{v(v-1)}{2} - \frac{c(c-1)}{2} - c)$, all integers within $[c - 1 .. \frac{v(v-1)}{2} - 1 - \frac{c(c-1)}{2}]$ are taken. In like manner, all integers in [2 .. n − 1] are taken for a specific number of vertices n if core-shares > n − 3. For each entry, (n, c, m), of the reference table, *r* random graphs, which have properties: each of th[illegible]s n vertices, a completely connected subgraph of order c, and m edges besides those that are in the given completely connected subgraph, are generated, any exact algorithm can be used to obtain the

chromatic numbers of the sequence of random graphs, the expected-colors is then found, and finally the difference between the computed expected-colors and c is placed into the content of this entry. For some entries, the number of all possible graphs having the same properties may be less than *r*. In this case, we compute the expected-colors by using all possible graphs.

Before describing the interpolation at a partially colored graph, we introduce two terms. For a c-partially colored graph of order n, the ***coredensity*** is defined to be the ratio of c to n, and the ***edgedensity*** is the ratio of the number of edges except those that are in the core to $\frac{n(n-1)}{2} - \frac{c(c-1)}{2}$. The procedure of interpolation is (1) to formalize the given partially coloring graph into the form (n, c, m), where n is the number of vertices, c is the number of colors having been used so far, and m is the number of edges except those that are in the core, (2) to interpolate in the vertex direction, (3) to interpolate (if necessary) in the core direction, and (4) to interpolate (if necessary) in the edge direction. We assume that procedures (2)-(4) always try to keep every two graphs, which are either within an interpolation or between two consecutive interpolations, in the same (or closer) coredensity and edgedensity. Figures 61-63 show that (1) function ***l-interpolation*** is a linear interpolation function which is to interpolate at the target, whose value is between the lower neighbor *lownb* and the higher neighbor *highnb* ; (2) function ***edge-interpolation*** interpolates linearly along the edge coordinate if n is a given datum in the vertex coordinate, and c is also a given datum with respect to n in the core coordinate; (3) function ***core-interpolation*** interpolates linearly along the core coordinate if n is a given datum in the vertex coordinate; (4) function ***vertex-interpolation*** does linear interpolation along the vertex coordinate if n is not a given datum; and (5) function ***preference-degree*** triggers off the interpolations, and returns the degree of preference of a graph. For example, P(4, 1, 5) = 3 − 1 = 2 because any graph having 5 edges of order 4 is 3-colorable; P(4, 2, 2)

= 2.2 − 2 = 0.2 because there are 10 possible graphs, and the mean of all chromatic numbers is 2.2.

```
/* linear interpolation at targer, whose value is between lownb
   and highnb */
function l-interpolation(
      lownb,      /* lower neighbor */
      lowval,     /* score value of lower neighbor */
      highnb,     /* higher neighbor */
      highval,    /* score value of higher neighbor */
      target:   integer): real;

   begin
      return (lowval * (highnb − target) + highval * (target − lownb))
         / (highnb − lownb)
   end;


function edge-interpolation(n, c: integer; edensity: real): real;
/* n is a given datum in the vertex coordinate, and c is also a given
   datum w. r. t. n in the core coordinate. */
   var
      em:   integer; /* estimated edges */

   begin
      em ← edensity * (n(n − 1) − c(c − 1))/2;
      if em is a given datum in the edge coordinate w. r. t. (n, c)
         then return P(n, c, em)
      else begin
         Among the data points in the edge coordinate w. r. t. (n, c),
            find m1 which  is the closest lower neighbor to em and
            m2, which is the closest upper neighbor to em;
         return l-interpolation(m1,  P(n, c, m1),  m2,  P(n, c, m2),  em);
      end
   end;
```

Figure 61. part 1 of the preference function

```
function core-interpolation(n: integer; cdensity, edensity: real): real;
/* n is a given data in the vertex coordinate */
    var
        ec: integer; /* estimated size of core */

    begin
        ec ← n * cdensity;
        if ec is a given datum in the core coordinate w. r. t. n
            then return edge-interpolation(n, ec, edensity)
        else begin
            Among the given datum points in the core coordinate w. r. t. n,
                find c1 which  is the closest lower neighbor to ec and
                c2 which is the closest upper neighbor to ec;
            return l-interpolation(c1, edge-interpolation(n, c1, edensity), c2,
                edge-interpolation(n, c2, edensity), ec);
        end
    end;


function vertex-interpolation(n: integer; cdensity, edensity: real): real;

    begin
        if n is a given datum in the vertex coordinate
            then return core-interpolation(n, cdensity, edensity)
        else begin
            Among the given data points in the vertex coordinate
                find n1 which  is the closest lower neighbor to n and n2 which
                is the closest upper neighbor to n;
            return l-interpolation(n1, core-interpolation( n1,  cdensity, edensity),
                n2,core-interpolation( n2,  cdensity, edensity), n);
        end
    end;
```

Figure 62. part 2 of the preference function

B. COMPUTATIONAL RESULTS

While building the reference table of entry form (n, c, m), we let the vertex-increment = 4, core-shares = 8 between 2 and $n - 1$, edge-shares = 10 between $c - 1$ and $\frac{v(v-1)}{2} - 1 - \frac{c(c-1)}{2}$, and 100 random graphs are generated in order to

```
function preference-degree(n, c, m: integer): real;
/* n: number of vertices,
   c: number of vertices in core,
   m: number of edges except those that are in core. */
   var
      cdensity, edensity: real; /* coredensity and edgedensity */

   begin
      cdensity ← c/n;
      edensity ← (2 * m) / (n(n − 1) − c(c − 1));
      return vertex-interpolation(n, cdensity, edensity);
   end
```

Figure 63. part 3 of the preference function

calculate the expected-colors. In order to build the reference table in a reasonable amount of time, we compute the given data entries up to 52 vertices. We assume *inbuf* can hold pending nodes up to 128.

Given a graph, we initially convert it to a 0-partially coloring node (refer to chapter 4) and place it into *outbuf*. The coloring process is to repeatly make use of the following cycle: using DFS to find all m-partially colored nodes, which are sorted by the preference degrees, in *outbuf*, placing them into *inbuf* according to corresponding values of calling *preference-degree* function with c = m, and transforming the heapsort tree, *inbuf*, to the sorted list, *outbuf*. The coloring process stops as soon as the leaf of a complete coloring appears.

In Tables LXXIX-LXXXVI, the "best color" column presents the number of colors required for a complete coloring of various selection functions under the branch-and-bound scheme. A minor modification on the 2-1 swapping and 3-2 swapping has been done as follow: for each swapping method, instead of doing the

swapping cycle until there is no swapping candidate, we do the swapping cycle no more than once.

Tables LXXIX-LXXXVI show that (1) the Korman algorithm with either 2-1 swapping or 2-1 swapping plus 1-1 swapping is superior to the Korman algorithm; and (2) the Pactual algorithm with 2-1 swapping is superior to the Pactual algorithm; moreover, it is slightly better than Korman algorithm except for v = 40 and edgeload = 0.9.

C. DISCUSSION

The precision of this heuristic algorithm is determined by the size of buffer, *inbuf*, for holding pending nodes of the same number of partial colors (refer to chapter 4). The preference function proposed in this chapter is based on the data which are obtained from the mean of chromatic numbers, using an exact algorithm, of 100 random graphs.

From the experimental results, a selection function within the branch-and-bound scheme takes fewer forward moves but more running time (over 50%) than the same selection function within the backtracking scheme. The look-ahead procedure does nothing in the branch-and-bound scheme because the branch-and-bound scheme stops running upon finishing a complete coloring. The swapping cycle, finding a candidate and doing swapping, of a swapping method will be done once for each forward move. In the branch-and-bound scheme as we have previously shown for the backtracking scheme, the Korman algorithm with swapping is superior to the Korman algorithm.

Table LXXIX. VERTICES = 30, EDGELOAD = 0.3 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	5.02	38.81	0.303	5.02	1	100	0
ikorman	jsucadja		38.96	0.337	5.02	1	100	0
ipkorman	jkorman		37.51	0.315	5.02	1	100	0
ipactual	jpactual		35.39	0.257	5.02	1	100	0
iprevent1a	jprevent1a		37.05	0.345	5.02	1	100	0
iprevent2a	jprevent2a		36.60	0.316	5.02	1	100	0
iprevent3a	jprevent3a		36.19	0.243	5.02	1	100	0
iprevent4a	jprevent4a		35.33	0.236	5.02	1	100	0
iconnecta	jconnecta		36.97	0.240	5.02	1	100	0
ikorqk2	jkorman		37.72	0.279	5.02	1	100	0
ikorw2	jkorman		37.09	0.281	5.02	0	100	0
ikorw21e	jkorman		38.34	0.281	5.02	1	100	0
ikormaxw2	jkorman		37.44	0.281	5.02	0	100	0
ikormaxw21e	jkorman		38.67	0.281	5.02	1	100	0
ikorqk23	jkorman		36.30	0.245	5.02	1	100	0
ikorw23	jkorman		35.90	0.246	5.02	1	100	0
ipactqk2	jpactual		35.23	0.237	5.02	1	100	0
ipactmaxw2	jpactual		35.24	0.227	5.02	1	100	0

Table LXXX. VERTICES = 30, EDGELOAD = 0.5 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	7.02	78.92	0.920	7.02	2	100	0
ikorman	jsucadja		80.21	0.916	7.02	2	100	0
ipkorman	jkorman		76.39	0.876	7.02	2	100	0
ipactual	jpactual		69.39	0.967	7.02	2	100	0
iprevent1a	jprevent1a		71.96	0.950	7.02	2	100	0
iprevent2a	jprevent2a		75.68	0.997	7.02	2	100	0
iprevent3a	jprevent3a		66.93	0.918	7.02	2	100	0
iprevent4a	jprevent4a		62.93	1.016	7.02	2	100	0
iconnecta	jconnecta		77.08	0.810	7.02	2	100	0
ikorqk2	jkorman		72.50	1.150	7.02	1	100	0
ikorw2	jkorman		75.39	1.207	7.02	1	100	0
ikorw21e	jkorman		76.67	1.160	7.02	1	100	0
ikormaxw2	jkorman		70.43	1.155	7.02	1	100	0
ikormaxw21e	jkorman		74.29	1.149	7.02	1	100	0
ikorqk23	jkorman		70.25	1.335	7.02	2	100	0
ikorw23	jkorman		65.16	1.209	7.02	2	100	0
ipactqk2	jpactual		70.75	1.085	7.02	2	100	0
ipactmaxw2	jpactual		70.91	1.087	7.02	2	100	0

Table LXXXI. VERTICES = 30, EDGELOAD = 0.7 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	10.00	73.91	0.603	10.00	2	100	0
ikorman	jsucadja		76.50	0.599	10.00	2	100	0
ipkorman	jkorman		73.42	0.631	10.00	2	100	0
ipactual	jpactual		60.78	0.686	10.00	2	100	0
iprevent1a	jprevent1a		68.72	0.734	10.00	2	100	0
iprevent2a	jprevent2a		62.85	0.637	10.00	2	100	0
iprevent3a	jprevent3a		63.92	0.655	10.00	2	100	0
iprevent4a	jprevent4a		62.06	0.603	10.00	2	100	0
iconnecta	jconnecta		84.48	0.861	10.00	3	100	0
ikorqk2	jkorman		65.36	0.623	10.00	1	100	0
ikorw2	jkorman		63.46	0.585	10.00	1	100	0
ikorw21e	jkorman		65.31	0.570	10.00	1	100	0
ikormaxw2	jkorman		64.53	0.625	10.00	1	100	0
ikormaxw21e	jkorman		66.32	0.608	10.00	1	100	0
ikorqk23	jkorman		57.73	0.556	10.00	2	100	0
ikorw23	jkorman		55.98	0.509	10.00	2	100	0
ipactqk2	jpactual		57.05	0.644	10.00	2	100	0
ipactmaxw2	jpactual		57.03	0.645	10.00	2	100	0

Table LXXXII. VERTICES = 30, EDGELOAD = 0.9 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	15.87	31.88	0.163	15.87	1	100	0
ikorman	jsucadja		32.37	0.179	15.87	1	100	0
ipkorman	jkorman		30.92	0.124	15.87	1	100	0
ipactual	jpactual		30.63	0.095	15.87	1	100	0
iprevent1a	jprevent1a		30.69	0.100	15.87	1	100	0
iprevent2a	jprevent2a		30.60	0.098	15.87	1	100	0
iprevent3a	jprevent3a		30.58	0.090	15.87	1	100	0
iprevent4a	jprevent4a		31.72	0.154	15.87	2	100	0
iconnecta	jconnecta		30.65	0.093	15.87	1	100	0
ikorqk2	jkorman		31.51	0.112	15.87	1	100	0
ikorw2	jkorman		31.51	0.112	15.87	1	100	0
ikorw21e	jkorman		32.41	0.083	15.87	1	100	0
ikormaxw2	jkorman		31.51	0.112	15.87	1	100	0
ikormaxw21e	jkorman		32.42	0.083	15.87	1	100	0
ikorqk23	jkorman		31.09	0.052	15.87	1	100	0
ikorw23	jkorman		31.00	0.050	15.87	1	100	0
ipactqk2	jpactual		30.73	0.069	15.87	1	100	0
ipactmaxw2	jpactual		30.73	0.069	15.87	1	100	0

Table LXXXIII. VERTICES = 40, EDGELOAD = 0.3 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	5.96	172.85	0.882	5.96	5	100	0
ikorman	jsucadja		172.52	0.882	5.96	5	100	0
ipkorman	jkorman		148.46	0.856	5.96	5	100	0
ipactual	jpactual		132.45	0.784	5.96	6	100	0
iprevent1a	jprevent1a		140.32	0.826	5.96	6	100	0
iprevent2a	jprevent2a		133.53	0.818	5.96	6	100	0
iprevent3a	jprevent3a		134.39	0.827	5.96	6	100	0
iprevent4a	jprevent4a		124.97	0.751	5.96	5	100	0
iconnecta	jconnecta		162.78	0.800	5.96	7	100	0
ikorqk2	jkorman		156.05	0.871	5.96	4	100	0
ikorw2	jkorman		153.38	0.889	5.96	4	100	0
ikorw21e	jkorman		157.20	0.830	5.96	4	100	0
ikormaxw2	jkorman		156.65	0.889	5.96	4	100	0
ikormaxw21e	jkorman		161.26	0.837	5.96	4	100	0
ikorqk23	jkorman		126.72	0.828	5.96	4	100	0
ikorw23	jkorman		125.50	0.865	5.96	5	100	0
ipactqk2	jpactual		126.65	0.790	5.96	5	100	0
ipactmaxw2	jpactual		128.80	0.786	5.96	5	100	0

Table LXXXIV. VERTICES = 40, EDGELOAD = 0.5 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	8.23	809.70	1.456	8.23	22	100	0
ikorman	jsucadja		804.02	1.477	8.23	24	100	0
ipkorman	jkorman		795.17	1.627	8.23	27	100	0
ipactual	jpactual		564.77	1.533	8.23	24	100	0
iprevent1a	jprevent1a		700.98	2.108	8.23	28	100	0
iprevent2a	jprevent2a		646.85	1.755	8.23	27	100	0
iprevent3a	jprevent3a		649.86	1.876	8.23	27	100	0
iprevent4a	jprevent4a		600.05	1.623	8.23	26	100	0
iconnecta	jconnecta		920.15	1.702	8.23	38	100	0
ikorqk2	jkorman		697.71	1.465	8.23	18	100	0
ikorw2	jkorman		693.52	1.432	8.23	18	100	0
ikorw21e	jkorman		701.82	1.455	8.23	17	100	0
ikormaxw2	jkorman		719.68	1.403	8.23	18	100	0
ikormaxw21e	jkorman		723.60	1.437	8.23	18	100	0
ikorqk23	jkorman		549.69	1.458	8.23	24	100	0
ikorw23	jkorman		562.62	1.419	8.23	30	100	0
ipactqk2	jpactual		541.32	1.551	8.23	21	100	0
ipactmaxw2	jpactual		542.22	1.548	8.23	20	100	0

Table LXXXV. VERTICES = 40, EDGELOAD = 0.7 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	11.88	575.70	0.819	11.88	17	100	0
ikorman	jsucadja		581.97	0.809	11.88	18	100	0
ipkorman	jkorman		546.78	0.888	11.88	20	100	0
ipactual	jpactual		366.46	0.906	11.88	16	100	0
iprevent1a	jprevent1a		435.31	0.868	11.88	19	100	0
iprevent2a	jprevent2a		394.58	0.807	11.88	18	100	0
iprevent3a	jprevent3a		383.48	0.797	11.88	17	100	0
iprevent4a	jprevent4a		398.99	0.846	11.88	19	100	0
iconnecta	jconnecta		551.03	1.082	11.88	24	100	0
ikorqk2	jkorman		485.78	0.822	11.88	13	100	0
ikorw2	jkorman		477.00	0.809	11.88	13	100	0
ikorw21e	jkorman		452.62	0.784	11.88	11	100	0
ikormaxw2	jkorman		503.17	0.891	11.88	14	100	0
ikormaxw21e	jkorman		471.21	0.825	11.88	12	100	0
ikorqk23	jkorman		400.75	0.929	11.88	29	100	0
ikorw23	jkorman		394.14	0.932	11.88	35	100	0
ipactqk2	jpactual		336.00	0.944	11.88	14	100	0
ipactmaxw2	jpactual		337.41	0.961	11.88	13	100	0

Table LXXXVI. VERTICES = 40, EDGELOAD = 0.9 (branch-and-bound scheme). variations of Korman with and without swapping

nucv	ncv	exact color	moves	var cof.	best color	time (sec)	0-ap	1-ap
ikorman	jkorman	19.20	72.87	0.630	19.20	2	100	0
ikorman	jsucadja		73.62	0.619	19.20	2	100	0
ipkorman	jkorman		56.49	0.541	19.20	3	100	0
ipactual	jpactual		52.17	0.415	19.20	3	100	0
iprevent1a	jprevent1a		55.13	0.489	19.20	3	100	0
iprevent2a	jprevent2a		52.20	0.455	19.20	3	100	0
iprevent3a	jprevent3a		54.94	0.534	19.20	3	100	0
iprevent4a	jprevent4a		61.25	0.557	19.20	4	100	0
iconnecta	jconnecta		75.18	1.642	19.21	4	99	1
ikorqk2	jkorman		56.61	0.477	19.20	1	100	0
ikorw2	jkorman		56.73	0.471	19.20	1	100	0
ikorw21e	jkorman		61.57	0.633	19.20	1	100	0
ikormaxw2	jkorman		56.44	0.472	19.20	1	100	0
ikormaxw21e	jkorman		61.43	0.636	19.20	1	100	0
ikorqk23	jkorman		51.42	0.415	19.20	5	100	0
ikorw23	jkorman		50.14	0.377	19.20	6	100	0
ipactqk2	jpactual		51.46	0.428	19.20	3	100	0
ipactmaxw2	jpactual		51.46	0.428	19.20	3	100	0

VIII. CONCLUSIONS

A. SUMMARY

The graph coloring problem (GCP) is a classic graph related problem. In addition, it is NP-complete. There are many well-known algorithms for the GCP. We distinguish a good graph coloring algorithm by two parameters, the number of forward moves and the running time, which are obtained by running the algorithms on a sequence of random graphs. From all the algorithms we have tried, we have found two variations on Korman which seem to be beneficial especially for finding exact colorings. We have not tested these algorithms on graphs of order more than 56. Thus, although one might suspect that similar conclusions will hold for larger graphs, we do not know this. The evidence is entirely experimental. No asymptotic results have been proved to show that the trend established on graphs of up to 56 vertices will continue. Although we have looked at some heuristic algorithms, our results are mainly in exact colorings. From the observation of experimental results, we draw the following conclusions.

(1) The Korman algorithm which has been the de facto standard for many years is a simple and efficient algorithm. For small graphs, whose order is smaller than 40, the Korman algorithm with backtracking finds the chromatic number reasonably quickly.

(2) The Pactual algorithm substantially cuts the number of forward moves, but the additional computational time is, however, also significant. For graphs of order 36, 40, 44, 48, 52, and 56 on edgeload = 0.3, Pactual saves 25% - 42% forward moves over Korman. For graphs of order 36, 40, 44, 48, 52, and 56 on edgeload = 0.5, Pactual saves 36% - 59% forward moves over Korman. For graphs of order 32, 36, 40, 44, 48, 52, and 56 on edgeload = 0.7, Pactual saves 29% - 63% forward moves over

Korman. For graphs of order 36, 40, 44, 48, 52, and 56 on edgeload = 0.9, Pactual saves 23% - 52% forward moves over Korman. However, Pactual is slower than Korman except that n = 52 on edgeload = 0.7 (saving 35%), and n = 56 on edgeload = 0.5, 0.7, and 0.9 (saving 10% - 24%). One might conjecture that for large graphs, the comparison would be similar to graphs of order 56. However, this is not certain.

(3) The effect of adding the 2-1 swapping to the Korman algorithm or any of its variation is significantly beneficial, especially when edgeload = 0.5 and 0.7. For graphs of order 48, 52, and 56 on edgeload = 0.3, Korw2 is 10% to 18% faster than Korman. For graphs of order 40, 44, 48, 52, and 56 on edgeload = 0.5, Korw2 is 12% to 20% faster than Korman. For graphs of order 40, 44, 48, 52, and 56 on edgeload = 0.7, Korw2 is 10% to 16% faster than Korman. It seems that Korw2 will not run faster than Korman for graphs of order less than 52 on 90% edgeload. For graphs of order 36, 44, 48, 52, and 56 on edgeload = 0.3, Pactmaxw2 is 14% to 33% faster than Pactual. For graphs of order 36, 40, 44, 48, 52, and 56 on edgeload = 0.5, Pactmaxw2 is 10% to 25% faster than Pactual. For graphs of order 36, 40, 44, 48, 52, and 56 on edgeload = 0.7, Pactmaxw2 is 12% to 25% faster than Pactual. For graphs of order 44, 48, 52, and 56 on edgeload = 0.9, Pactmaxw2 is 13% to 25% faster than Pactual.

Swapping appears to be beneficial in the mean except perhaps on some graphs of small order. Pactual only begins to be beneficial in terms of running time on graphs of order approximately 56. However, Pactual and swapping together, Pactmaxw2, appear to be exceptionally beneficial for graphs of order 56. Although we do not know this for certain, we suspect that for graphs of higher order, these variations on Korman will prove to be at least as beneficial as on graphs of order 56.

(4) The look-ahead procedure, which eliminates from further consideration a block vertex (refer to chapter 6) and removes block colors (refer to chapter 6) from the

feasible color set, significantly improved running time and the number of forward moves.

(5) Under non-backtracking, the Pactual algorithm with 2-1 swapping colors a graph with fewer colors than the Korman algorithm. Pactmaxw2 uses the number of heuristic colors which is about 0.13 - 0.38 colors closer to the chromatic number than Korman except n = 28 on edgeload = 0.9, n = 32 on edgeload = 0.5 and 0.9, n = 36 on edgeload = 0.3, and n = 40 on edgeload = 0.9, about 0.02 - 0.09 colors closer than Korman. For graphs of order 56 on edgeload = 0.5, 0.7, and 0.9, Pactmaxw2 is better than Korman by the amount of the difference in running time which is larger than the amount of the difference in running time between Korw2 and Korman. Pactmaxw2 without backtracking appears to be the best heuristic algorithm among the variations of the vertex-color sequential algorithm which we have programmed.

(6) In comparing branch-and-bound as any exact algorithm with backtracking, branch-and-bound is much slower due to the necessity of saving partially colored graphs, and is not recommended.

(7) Among the heuristic algorithm, we have considered *limit*, *epsilon*, and *branch-and-bound*. *Limit* is a probabilistic algorithm which narrows the average number of branches from a node to a real number between 1 and 2. *Epsilon* narrows the number of branches of a node by setting a threshold on the scores of each branch. Branch-and-bound uses a preference function and a preset buffer size to prune the search tree in a breadth first search. It appears that the Pactual algorithm with 2-1 swapping is also beneficial for heuristic coloring of large graphs. We have not compared these heuristic algorithms sufficiently with other heuristic algorithms to yield any definitive conclusion.

B. FURTHER RESEARCH TOPICS

(1) Compare the 4 exact algorithms, Korman, Korw2, Pactual, and Pactmaxw2, on graphs of order more than 56. This will require a considerable amount of computer time.

(2) The new color, in the feasible color set, is always the last choice in our developed ncv-selection functions. However, this kind of arrangement may need more backward moves and forward moves for some graph. Is there a ncv-selection function which handles the new color better than we did?

(3) The weight function mentioned in *ipkorman* in chapter 6 is helpful in selecting the nucv which is adjacent to a neighbor having a particularly high chrome-degree. While observing the experimental results of various bases, we only know it is helpful in the situation, weight $\neq 0$ or 1. We wonder if there are *certain* weights that are exceptionally helpful in improving performance.

(4) In the selection functions (nucv-selection plus ncv-selection function), we use three parameters, the set of adjacent vertices, the chrome-degree, and the white-degree, to decide the *nucv* and the *ncv*. Is there an exceptionally good selection function whose complexity is close to that of the Korman algorithm using those three parameters or an extra parameter?

(5) In the branch-and-bound scheme, is there a good preference function which obtains a coloring close to the chromatic number using a small number of buffers?

(6) Compare the heuristic algorithms, *limit*, *epsilon*, and *branch-and-bound*, with other known heuristic algorithms and attempt to find optimal values on the parameters of *limit*, *epsilon*, and *branch-and-bound*. Test whether heuristic algorithms given here with optimal parameter are preferable to other heuristic algorithms on large graphs.

REFERENCES

[*AH*74] Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1974). *The Design and Analysis of Computer Algorithms,* Addison-Wesley Publishing Co. Inc., Reading, Massachusetts.

[*AH77a*] Appel, K. and Haken, W. (1977). "Every planar map is four-colorable part I: discharging." *Illinois J. of Mathematics* **21, 3,** pp. 429-490.

[*AH77b*] Appel, K., Haken, W., and Koch, J. (1977). "Every planar map is four-colorable part II: reducibility." *Illinois J. of Mathematics* **21, 3,** pp. 491-567.

[*AH*83] Aho, A. V., J. E. Hopcroft and J. D. Ullman (1983). *Data Structures and Algorithms,* Addison-Wesley Publishing, Co. Inc.

[*Be*73] Berge, C. (1973). *Graphs and Hypergraphs,* North-Holland Publishing, Amsterdam.

[*Be*77] Bernhart, F. R. (1977). "A digest of the four color theorem." *J. Graph Theory* **1,** pp. 207-225.

[*BF*87] Bratley, P., Fox, B. L., and Schrage, E. L. (1987). *A Guide to Simulation, Second Edition,* Springer-Verlag, New York.

[*BK*73] Bron, C. and J. Kerbosch (1973). "Finding all cliques of an undirected graph." *Comm. of ACM* **16,** pp. 575-577.

[*Br*41] Brooks, R. L. (1941). "On colouring the nodes of a network." *Proc. Cambridge Phil. Soc.* **37,** pp. 194-197.

[*Br*72] Brown, J. R. (1972). "Chromatic scheduling and the chromatic number problem." *Management Science* **19,** pp. 456-463.

[*Br*77] Brualdi, R. A. (1977). *Introductory Combinatorics,* Elsevier Science publishing Co. Inc., New York.

[*Br*79] Brèlaz, D. (1979). "New methods to color the vertices of a graph." *Comm. of ACM* **22,** pp. 251-256.

[*Ch*71] Christofides, N. (1971). "An algorithm for the chromatic number of a graph." *The Computer J.* **14,** pp. 38-39.

[*Ch*75] Christofides, N. (1975). *Graph Theory,* Academic Press, London.

[*CG*73] Corneil, D. G. and Graham, B. (1973). "An algorithm for determining the chromatic number of a graph." *SIAM J. Comput.* **2,** pp. 311-318.

[*Co*71] Cook, S. A. (1971). "The complexity of theorem proving procedures." *Proc. 3rd Annual ACM Symposium on Theory of Computing,* pp. 151-158.

[*DB*81] Dutton, R. D. and Brigham, R. C. (1981). "A new graph colouring algorithm." *Computer J.* **24**, pp. 85-86.

[*Di*59] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs." *Numerische Mathematik* **1**, pp. 269-271.

[*EC*71] Eilon, S. and Christofides, N. (1971). "The loading problem." *Management Science* **17**, pp. 259-268.

[*FM*86] Fishman, G. S. and Moore, L. R. (1986). "An exhaustive analysis of multiplicative congruential random number generators with modulus 2^{31}-1." *SIAM J. Sci. Stat. Comput.* **7**, **1**, pp. 24-45.

[*GJ*76] Garey, M. R. and Johnson, D. S. (1976). "The complexity of near-optimal graph coloring." *J. of ACM* **23**, **1**, pp. 43-49.

[*GJ*79] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, New York.

[*GM*75] Grimmett, G. R. and McDiarmid, C. J. H. (1975). "On colouring random graphs." *Math. Proc. Comb. Phil. Soc.* **77**, pp. 313-324.

[*Ha*77] Haken, W. (1977). "An attempt to understand the four color problem." *J. Graph Theory* **1**, pp. 193-206.

[*He*90] Heawood, P. J. (1890). "Map-colour theorems." *Quarterly J. of Math. Oxford Ser.* **24**, pp. 322-338.

[*HS*78] Horowitz, E. and Sahni, S. (1978). *Fundamentals of Computer Algorithms* Computer Science Press, Inc.

[*Hu*82] Hu, T. C. (1982). *Combinatorial Algorithms* Addison-Wesley Publishing Co., Inc.

[*Jo*74] Johnson, D. S. (1974) "Worst case behavior of graph coloring algorithms." *Proc. 5-th S-E Conf. Combinatorics, Graph Theory and Computing*, pp. 513-527.

[*Ka*72] Karp, R. M. (1972) "Reducibility among combinatorial problems." in *Complexity of Computer Computations*, eds. R. E. Miller and J. W. Thatcher, Plenum Press, New York, pp. 85-103.

[*Ke*79] Kempe, A. B. (1879) "On the geographical problems of the form colours." *Amer. J. Maths.* **2**, pp. 193-200.

[*KJ*85] Kubale, M. and Jackowski, B. (1985). "A generalized implicit enumeration algorithm for graph coloring." *Comm. of ACM* **28**, pp. 412-418.

[*Ko*79] Korman, S. M. (1979). "The graph-colouring problem." in *Combinatorial Optimization*, eds. N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, Wiley, New York, pp. 211-235.

[*Kr*56] Kruskal, J. B. Jr. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem." *Proc. Amer. Math. Soc.* **7**, pp. 48-50.

[*Le*51] Lehmer, D. H. (1951). "Mathematical methods in large-scale computing units." *Annu. Comput. Lab. Harvard Univ.* **26**, pp. 141-146.

[*Le*79] Leighton, F. T. (1979). "A graph coloring algorithm for large scheduling problems." *J. Res. Nat. Bur. Standards* **84, 6** pp. 489-506.

[*LW*66] Lawler, E. L. and Wood, D. E. (1966). "Branch-and-bound methods: a survey." *Operations Research*, **14**, pp. 699-719.

[*Ma*81] Manvel, B. (1981). "Coloring large graphs." *Congressus Numerantium*, **33**, pp. 197-204.

[*Mc*79] McDiarmid, C. (1979). "Colouring random graphs badly." in *Graph Theory and Combinatorics*, ed. R. J. Wilson, Pitman Research Notes in Mathematics 34, Pitman, London, pp. 76-86.

[*Mi*70] Mitten, L. (1970). "Branch-and-bound methods: general formulation and properties." *Operations Research*, **18**, pp 24-34.

[*MM*72] Matula, D. W., Marble, G., and Isaacson, J. D. (1972). "Graph coloring algorithm." in *Graph Theory and Computing*, ed. Read, R. C., Academic Press, New York, pp. 109-122.

[*My*55] Mycielski, J. (1955). "Sur le coloriage des graphes." *Colloq. Math.* **3**, pp. 161-162.

[*Pe*83] Peemöller, J. (1983). "A correction to Brélaz's modification of Brown's coloring algorithm." *Comm. of ACM* **26**, pp. 595-597.

[*PM*88] Park, S. K. and Miller, K. W. (1988). "Random number generators: good ones are hard to find." *Comm. of ACM* **31, 10** pp. 1192-1201.

[*PS*82] Papadimitriou, C. H. and Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

[*PW*66] Peck, J. E. L. and Williams, M. R. (1966). "Examination Scheduling." Algorithm 286, *Comm. of ACM* **9, 6**, pp. 433-434.

[*RF*73] Roschke, S. I. and Furtado, A. L. (1973). "An algorithm for obtaining the chromatic number and an optimal coloring of a graph." *Information Processing Letters* **2**, pp. 34-38.

[*SD*83] Syslo, M. M., Deo, N., and Kowalik, J. S. (1983). *Discrete Optimization Algorithms with Pascal Programs*. Prentice Hall, Inc., Englewood Cliffs, New Jersey.

[*SW*68] Szekeres, G. and Wilf, H. S. (1968). "An inequality for the chromatic number of graph." *J. Combinatorial Theory* **4**, pp. 1-3.

[*Wa*74] Wang, C. C. (1974). "An algorithm for the chromatic number of a graph." *J. of ACM* **21**, pp. 385-391.

[*Wi*69] Williams, M. R. (1969). "The colouring of very large graphs." *Proceedings of the Calgary International Conf. on Combinatorial structures and their Application* held at the University of Calgary, Canada, pp. 477-478.

[*Wo*68] Wood, D. C. (1968). "A technique for colouring a graph applicable to large scale timetabling problems." *Computer J.* **12**, pp. 317-319.

[*WP*67] Welsh, D. J. A. and Powell, M. B. (1967). "An upper bound for the chromatic number of a graph and its application to timetabling problems." *Comput. J.* **10**, pp. 85-86.

[*Zy*52] Zykov, A. A. (1952). "On some properties of linear complexes." *Amer. Math. Soc.* Translation **79**, pp. 163-188.

VITA

Shi-Jen Lin was born on October 23, 1955 in Hsinchu, Taiwan. He received his primary and secondary education in Hsinchu.

He received his undergraduate education at National Taiwan Normal University in Taipei, Taiwan. While Completing his Bachelor's degree, he worked as a mathematics teacher for Pei-In Junior High School in Hsinchu.

From August, 1982 to July, 1984 he was a graduate student in Computer Science at the University of Missouri-Rolla in Rolla, Missouri. He was awarded the Master of Science degree in Computer Science in July, 1984.

Since January, 1985 he has been a graduate student at the University of Missouri-Rolla, pursuing the Ph.D. degree in Computer Science.

He has been married to Tan Li since January, 1987. They have one daughter, I-Shen Lin.