RC 16326 (#72348) 11/16/90
Computer Science 31 pages

Research Report

New Algorithms to Color Graphs and Find Maximum Cliques

Thomas K. Philips

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

IBM FILE COPY
DO NOT MICROFILM

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents and will be distributed outside of IBM up to one year after the date indicated at the top of this page. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

IBM Research Division
Almaden • T.J. Watson • Tokyo • Zurich

New Algorithms to Color Graphs and Find Maximum Cliques

Thomas K. Philips

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract We define a quantity associated with every vertex in a graph, which we call the *Clique Potential*, and present a number of new heuristic algorithms for graph coloring and the maximum clique problem that use this quantity. The clique potential of a vertex v is defined to be the sum of the degrees of v and its neighbors, and is an approximate measure of whether or not the vertex belongs to a large clique. We also present a recursive method to transform any given graph coloring algorithm into a new algorithm that usually (though not necessarily) uses fewer colors to color a graph than the algorithm from which it was derived. The algorithms run in polynomial time, and experimental evidence shows that they outperform known algorithms, especially on large, dense, graphs. Tests of the algorithms on random graphs of various sizes are presented.

CONTENTS

I. Introduction

The Graph Coloring and Maximum Clique problems are simply stated, yet find application in many diverse areas of Computer Science and Operations Research. In the Graph Coloring problem, we are required to find an assignment of colors to the vertices of a graph so that no two adjacent vertices are assigned the same color. The Maximum Clique problem requires us to find the largest set of mutually adjacent vertices in a graph. As both problems are NP-complete [29], a great deal of work has been carried out on heuristics and bounds for these problems. In this paper we define a quantity associated with every vertex in the graph which we call the *Clique Potential,* and develop a number of heuristic algorithms for the graph coloring and maximum clique problems that exploit it. A brief survey of previous work in the area, and of some of the engineering applications of graph coloring and the maximum clique problem, follows. We first present some definitions from Bollobas [8].

A *Graph* $G = (V,E)$ is a collection of *Vertices,* $v \in V$ and *Edges,* $(u,v) \in E$. Each edge joins exactly two distinct vertices in V. Loops and multiple edges are disallowed. Two vertices that are joined by an edge are said to be *adjacent*. The *neighborhood* of a vertex v, denoted by $N(v)$, is the set of all vertices that are adjacent to v. The *degree* of a vertex, denoted by $d(v)$, is the number of vertices it is adjacent to. A *Subgraph* of G is a subset of the vertices and edges of G. We define the *Subgraph of G induced by the common neighbors of a set of vertices* U to be a graph $G' = (U', E')$, where $U' = \bigcap_{u \in U} N(u)$, and $E' = \{(u,v) \mid (u,v) \in E;\ u,v \in U'\}$. U' consists of all vertices not in U that are adjacent to every vertex in U and E' contains all the edges that join these vertices. The *complement* of a graph $G = (V,E)$ is the graph $G^C = (V^C, E^C)$, where $V^C = V$, and E^C contains all the edges not contained in E.

Formally, a *Random Graph* is a probability space (Ω, Σ, P) where Ω is a set of graphs, Σ is a σ –field of subsets of Ω, and P is a probability measure on Σ. It is more natural to think of a Random Graph as a graph in which the vertices and edges exist with some probability. We take $G_{n,p}$ to be a random graph on n vertices in which each edge exists, independently of all the other edges, with probability p. Although such a random graph is unlikely to be encountered in practice, experience has shown that heuristic algorithms that perform well on $G_{n,p}$ tend to perform well on problems encountered in practice [23]. In addition, the performance of graph coloring algorithms on $G_{n,p}$ is stable [39], i.e., the number of colors used by an algorithm on one instance of the graph differs by very little from the number of colors used on any other instance of the graph. Consequently, only n and p need be specified to allow valid comparisons of the performance of our algorithms with that of other algorithms. We shall use the notation of

Johnson et. al. [39], and define the random geometric graph $U_{n,r}$ to be the graph generated by positioning n vertices at random in the unit square, and joining vertices that lie at a distance of r or less from each other by an edge. The average degree of a vertex in $U_{n,r}$ is $\pi r^2 n$. Some properties of a closely related model in which points are distributed on the infinite XY plane according to a two dimensional Poisson point process can be found in [31] and [52].

A *Coloring* of a graph is an assignment of colors to the vertices of G so that no two adjacent vertices are assigned the same color. The minimum number of colors required for a valid coloring is known as the *Chromatic Number* of the graph and is denoted by $\chi(G)$. A *Clique* is a set of vertices that are mutually adjacent. The *Order* of a clique is the number of vertices contained in it. The order of the largest clique in G, denoted by $\mathrm{cl}(G)$, is known as the *Clique Number* of G, and a clique of order $\mathrm{cl}(G)$ is called a *Maximum Clique*. A clique whose order cannot be increased (because no vertex adjacent to all the vertices in the clique exists) is called a *Maximal Clique*. Note that while a maximum clique is maximal, a maximal clique need not be a maximum clique. The clique number of a graph is a lower bound on its chromatic number, as each vertex in the clique must be assigned a different color. Lastly, we define the *Clique Potential* of a vertex v, denoted $CP(v)$, to be the sum of the degrees of a vertex and all of its neighbors, i.e., $CP(v) = d(v) + \sum_{u \in N(v)} d(u)$.

In some applications it is natural to think of the vertices of a graph as tasks, and of the edges as conflicts between the tasks. The graph coloring problem may be thought of as a partitioning of the tasks into a minimum number of non-conflicting classes, while the maximum clique problem determines the largest set of tasks which conflict with each other. Applications of graph coloring and maximum clique algorithms include

1. The scheduling of university examinations and related timetabling problems [23, 44, 16]. Each course is represented by a vertex, and two vertices are joined by an edge if one or more students is taking both courses (a student cannot take two exams at the same time). The graph is colored, and examinations for all courses whose corresponding vertices are assigned the same color are conducted simultaneously.
2. The scheduling of jobs on machines [12, 13, 61, 64]. Each job is represented by a vertex, and two vertices are joined by an edge if their corresponding jobs cannot be performed on the same machine. The graph is colored, and all jobs of a given color are assigned to the same machine.
3. Register allocation in optimizing compilers [17, 18, 21, 43]. Variables in a program are represented by vertices, and two vertices are joined by an edge if their corresponding variables

are simultaneously live. The graph is colored, and all variables of a given color are assigned to the same register.

4. Pattern recognition for stereo vision and image comparison [7, 35]. Feature information is extracted from two pictures, and each feature is assigned a corresponding vertex in a graph. Two vertices are joined by an edge if the similarity between their corresponding features exceeds some threshold. The maximum clique identifies the best possible match between the two pictures.

5. Circuit Layout [63]. Transistors and their interconnections are treated as intervals on the real line, and an interval graph is derived from the circuit. The size of the maximum clique in this graph corresponds to the number of tracks required for the layout of the circuit.

6. VLSI design [59]. Variables are represented by vertices, and two vertices are joined by an edge if their corresponding variables can share a physical resource (register, arithmetic unit, interconnection unit etc.). A clique cover is found for the graph, and the vertices in each clique are allowed to share a physical resource.

7. Integer Programming [27]. A maximum clique algorithm is used to generate cutting planes for integer programs so that the solution generated by an LP relaxation of an integer program is closer to the optimal integer solution.

A great deal of work, concentrated in the following five areas, has been done on the graph coloring and maximum clique problems.

1. Exponential time algorithms to optimally solve the graph coloring and maximum clique problem on arbitrary graphs [1, 11, 13, 22, 42, 50, 53, 54, 56, 58].

2. Polynomial time algorithms to optimally (or near optimally) solve the graph coloring and maximum clique problem on specific classes of graphs [2, 3, 14, 20, 30, 33, 36, 37], or to solve it within some (non-constant) factor of optimality on arbitrary graphs [4, 5, 6, 62].

3. Polynomial time algorithms to solve these problems approximately (i.e. possibly suboptimally) on arbitrary graphs [11, 25, 41, 44, 48, 64].

4. Polynomial and sub-exponential time algorithms to solve the problems, often to within some constant factor of optimality, on random graphs [10, 24, 26, 41, 45, 47, 55, 60].

5. Randomized algorithms of various kinds [19, 28, 34, 39, 51] that solve these problems with high probability on random graphs by exploiting the statistical properties of random graphs.

Of these, we believe the first and third areas to hold the most promise in practical applications for the following reasons.

1. For graphs with less than about a hundred vertices, computers are fast enough to allow an exact solution in a reasonable amount of time (usually a few minutes of CPU time).
2. The graphs encountered in practical applications do not seem to be modelled well by any of the random graph models available.
3. With a few notable exceptions, the graphs encountered in practical applications do not seem to be drawn from any family of graphs for which exact polynomial time solutions are possible. Optimizing compilers are one example of such an exception: some types of programs generate interference graphs that are interval graphs.

In view of this, we focus our attention on the first and third areas, and present the following algorithms and evaluate their performance:

1. An $O(|V|^2|E|)$ algorithm to approximately solve the maximum clique problem,
2. An $O(|V|^2|E|)$ preprocessing algorithm, based on the maximum clique algorithm, that attempts to reduce a graph without modifying its clique number or chromatic number so that the graph coloring and maximum clique problem can be solved on a smaller graph. The preprocessing algorithm is not guaranteed to reduce the graph, but is very effective in practice.
3. An algorithm that takes any given graph coloring algorithm and derives from it a new algorithm that usually (though not necessarily) uses fewer colors to color a suboptimally colored graph than the given algorithm. On the classic test case of $G_{1000,.5}$ it reduces by between 20 and 25 the number of colors used by some widely used graph coloring algorithms.

II. Approximating the Maximum Clique

In this section we present a heuristic algorithm, the Clique Potential Algorithm (see box), for the maximum clique problem that runs in $O(|V|^2|E|)$ time, and derive from it a heuristic algorithm, the Independent Set Potential Algorithm, for the maximum independent set. The approximate maximum clique algorithm exploits the following simple observation about cliques: a vertex that belongs to a large clique will tend to have a high degree, as will all of its neighbors that belong to the clique. The clique potential captures this property in a succinct manner. For each vertex $v \in G$, the Clique Potential Algorithm estimates the largest clique containing v as follows. It initializes its estimate of the largest clique containing v to $\{v\}$, and then finds the subgraph induced by the neighbors of $\{v\}$. The clique is then augmented by the addition of the vertex of maximum clique potential in the induced subgraph. The subgraph induced by the common neighbors of the current clique is next determined and this procedure repeated till no vertices

remain, at which point the size of the clique that has just been found is noted. The entire procedure is repeated starting at every vertex in the graph. The largest clique that was found is finally output. While writing this paper, we found that a very similar algorithm had been proposed in [15]. It too, uses the clique potential as a decision variable, but differs from our algorithm in that the maximum clique is assumed to contain the vertex of maximum clique potential. Tests on random graphs (to be presented) show our algorithm to find substantially larger cliques. Our first theorem addresses the complexity of the Clique Potential Algorithm.

The Clique Potential Algorithm

- Set $Maximum\_Clique \leftarrow \phi$
- For each vertex $v \in G$ Do
  1. Set $Clique \leftarrow \{v\}$
  2. Set $C \leftarrow$ Subgraph of G induced by $N(v)$

     Do While C contains at least one vertex

     a. Compute the Clique Potentials of all the vertices in C (using their degrees in C)

     b. Find v_1, the vertex of maximum clique potential in C

     c. Set $Clique \leftarrow Clique \cup \{v_1\}$

     d. Set $C \leftarrow$ Subgraph of G induced by the common neighbors of $Clique$

     End do
  3. If Order($Clique$) > Order($Maximum\_Clique$), $Maximum\_Clique \leftarrow Clique$
- End do

Theorem 1:

The Clique Potential Algorithm runs in $O(|V|^2|E|)$ time.

Proof:

At each vertex v we have to compute the graph induced by v. Clearly, this can be done in $O(|E|)$ time at every vertex, and as every vertex in the graph must be examined, this contributes a $O(|V||E| + |V|)$ term to the final expression. Computing the clique potentials in the subgraph induced by v takes at most $O(d^2(v))$ time. Finding the vertex of maximum clique potential and updating the clique potentials each time a vertex is added to the clique can be done in $O(d(v))$ time, and as this is done at most $d(v)$ times, the computational effort at the vertex is at most $O(d^2(v))$. Updating the subgraph is done $O(d(v))$ times at each vertex v, and each time the computational effort at the vertex is at most $O(d^2(v))$. The total computational effort is therefore

upper bounded by $O(|V||E|) + \sum_{v \in G} d^3(v)$, subject to $d(v) < |V|$ and $\sum_{v \in G} d(v) = 2|E|$. The sum is maximized by setting as many of the degrees as possible to $|V| - 1$, and assigning any remaining edges to a single vertex. A quick calculation shows that $\sum_{v \in G} d^3(v)$ is $O(|V|^2|E|)$, and the result follows.

■

For each vertex v in G, the Clique Potential Algorithm returns the largest clique that it can find that contains v. If one wishes only to search for the maximum clique, the computational effort can be substantially reduced in practice by the use of the the following techniques.

1. Compute the clique potentials of all the vertices in G, and sort the vertices by their clique potential. Process the vertices in descending order (highest clique potential first). If the largest clique found after processing m vertices has order $\tilde{\mathrm{cl}}_m(G)$, ignore subsequent vertices of degree less than $\tilde{\mathrm{cl}}_m(G) - 1$.
2. Similarly, when the $m+1$st vertex is being processed, if at some stage there are c vertices in the current clique, there must be at least $\tilde{\mathrm{cl}}_m(G) + 1 - c$ vertices and $(\tilde{\mathrm{cl}}_m(G) + 1 - c)(\tilde{\mathrm{cl}}_m(G) - c)/2$ edges in the subgraph induced by the first c vertices if this clique is to be of order $\tilde{\mathrm{cl}}_m(G) + 1$ or larger. If fewer vertices or edges exist, the vertex can be set aside, and the processing of a new one started.

To evaluate the performance of the algorithm, we tested it on the random graphs $G_{n,.5}$ for n ranging from 100 to 1000. In addtion, we tested the algorithm of Campers et. al. [15] on the same set of graphs, so as to allow the performance of the two algorithms to be compared. For $n \leq 400$, ten thousand samples were generated for each value of n. For $n \geq 800$, limits on computation time restricted us to five thousand samples for each value of n. For each value of n, Table 1 contains data for all values of k for which the Clique Potential Algorithm found a maximal clique of size k. For convenience, we denote the expectation of the number of cliques of size k in the random graph $G_{n,.5}$ by $\mu(n,k)$, the variance of the number of cliques of size k in the same graph by $\sigma^2(n,k)$, the fraction of trials in which the largest clique found by the Clique Potential Algorithm is of size k by $f(n,k)$, and the fraction of trials in which the largest clique found by the algorithm in [15] is of size k by $g(n,k)$. Table 1 contains the results of these tests for all values of k for which the Clique Potential Algorithm found a maximal clique of size k. The following expressions for $\mu(n,k)$ and $\sigma^2(n,k)$ can be found in [10].

$$\mu(n,k) = \binom{n}{k} 2^{-k(k-1)/2}. \tag{1}$$

n	k	$f(n,k)$	$g(n,k)$	$\mu(n,k)$	$\sigma^2(n,k)$	$p(n,k)$
100	7	.0000	.0454	7.63×10^{3}	8.04×10^{6}	0.879
100	8	.0098	.5274	6.93×10^{2}	1.66×10^{5}	0.743
100	9	.7935	.3897	27.68	9.38×10^{2}	0.450
100	10	.1948	.0366	0.49	3.10×10^{1}	7.24×10^{-3}
100	11	.0019	.0009	0.0039	1.02×10^{-2}	1.52×10^{-3}
200	8	.0000	.0089	2.05×10^{5}	2.25×10^{9}	0.949
200	9	.0000	.3483	1.71×10^{4}	3.08×10^{7}	0.905
200	10	.0686	.5474	6.38×10^{2}	9.60×10^{4}	0.809
200	11	.8830	.0922	10.76	1.28×10^{2}	0.475
200	12	.0481	.0032	0.08	.26	0.026
200	13	.0003	.0000	2.92×10^{-2}	5.27×10^{-2}	1.62×10^{-2}
400	9	.0000	.0019	9.60×10^{6}	1.82×10^{12}	0.981
400	10	.0000	.2156	7.33×10^{5}	1.80×10^{10}	0.968
400	11	.0000	.6113	2.54×10^{4}	3.68×10^{7}	0.946
400	12	.5100	.1634	4.02×10^{2}	1.92×10^{4}	0.893
400	13	.4840	.0077	2.93	1.17×10^{1}	0.424
400	14	.0060	.0001	9.88×10^{-3}	1.91×10^{-2}	5.08×10^{-3}
800	10	.0000	.0002	7.94×10^{8}	4.51×10^{15}	0.993
800	11	.0000	.1270	5.57×10^{7}	3.45×10^{13}	0.989
800	12	.0000	.6244	1.79×10^{6}	5.38×10^{10}	0.983
800	13	.0124	.2358	2.65×10^{4}	1.79×10^{7}	0.975
800	14	.9420	.0124	1.82×10^{2}	1.96×10^{3}	0.944
800	15	.0454	.0002	.58	1.21×10^{1}	0.218
800	16	.0002	.0000	8.70×10^{-2}	1.27×10^{-1}	5.94×10^{-2}
1000	11	.0000	.0304	6.58×10^{8}	2.96×10^{15}	0.993
1000	12	.0000	.4946	2.65×10^{7}	7.16×10^{12}	0.990
1000	13	.0000	.4266	4.91×10^{5}	3.61×10^{9}	0.985
1000	14	.6520	.0470	4.23×10^{3}	4.09×10^{5}	0.978
1000	15	.3450	.0014	16.96	5.12×10^{1}	0.849
1000	16	.0030	.0000	3.19×10^{-2}	5.15×10^{-2}	0.019

Table 1. Performance of the Clique Potential Algorithm on $G_{n,.5}$

$$\sigma^2(n,k) = \binom{n}{k} 2^{-k(k-1)} \sum_{r=0}^{k} \binom{k}{r} \binom{n-k}{k-r} 2^{r(r-1)/2} - [\mu(n,k)]^2. \tag{2}$$

In [40] it is shown that for any non-negative integer random variable X,

$$\frac{(EX)^2}{EX^2} \leq Pr[X > 0] \leq EX, \tag{3}$$

where EX is its mean and EX^2 its second moment. Define

$$p(n,k) = \frac{\mu(n,k)^2}{\sigma^2(n,k) + \mu(n,k)^2}. \tag{4}$$

From equations (1 - 3) we see that $p(n,k)$ is a lower bound on the probability that a clique of size k exists in $G_{n,.5}$. Note that it is <u>not</u> a lower bound on the probability that the size of the maximum clique is k. In spite of this limitation, examining the numbers associated with the largest value of k for which the algorithm found a clique of size k and for which $f(n,k) > p(n,k)$ gives us some insights into the performance of the algorithm. We ignore values of k for which $f(n,k) < p(n,k)$ as the number of events in these cases is not large enough to allow reliable conclusions to be drawn. Therefore, for $n = 100$, we examine the numbers associated with $k = 11$, for 200, those with $k = 12$, and for 400, those with $k = 14$. For $100 \leq n \leq 400$, $f(n,k)$ is about 60 % of $\mu(n,k)$. The upper bound in equation (3), taken with the fact that $f(n,k)$ is a rapidly decreasing function of k, suggests that the largest clique found by the algorithm is a maximum clique at least 60 percent of the time. We expect this estimate to be conservative, as the upper bound in equation (3) is simply Markov's inequality, which is known to be loose. When $n \geq 800$, our requirement that $f(n,k) > p(n,k)$ is not satisfied for any value of k, and consequently the number of trials has to be substantially increased before any conclusion can be drawn.

Table 1 also shows that the Clique Potential Algorithm finds substantially larger cliques than the algorithm published in [15]. In fact, the empirical distribution of the size of the largest clique found by by the Clique Potential Algorithm stochastically dominates the empirical distribution of the size of the largest clique found by the algorithm in [15]; i.e. $\sum_{i>k} f(n,k) \geq \sum_{i>k} g(n,k)$ for all values of n and k in Table 1.

Since a clique in a graph is an independent set in the complement of the graph, the Clique Potential Algorithm can be applied to the complement of a graph to find independent sets in graphs. For completeness, we have appropriately modified the Clique Potential Algorithm so that it can be directly applied to a graph, and the modified algorithm is called the Independent Set Potential Algorithm. We define the *Independent Set Potential* of a vertex v, denoted $ISP(v)$, to be $ISP(v) = (|V| - d(v)) + \sum_{u \notin N(v)} (|V| - d(u))$. The independent set potential of a vertex is simply its clique potential in the complement of the graph. Just as the clique potential helped to identify

The Independent Set Potential Algorithm

- Set $Max\_Ind\_Set \leftarrow \phi$
- For each vertex $v \in G$ Do
  1. Set $Ind\_Set \leftarrow \{v\}$
  2. Set $I \leftarrow$ complement of the subgraph of G induced by the common neighbors of $v - \{v\}$

     Do While I contains at least one vertex

     a. Compute the Independent Set Potentials of all the vertices in I (using their degrees in I)

     b. Find v_1, the vertex of maximum independent set potential in I

     c. Set $Ind\_Set \leftarrow Ind\_Set \cup \{v_1\}$

     d. Set $I \leftarrow$ Complement of the subgraph of G induced by the common neighbors of $Ind\_Set - \{Ind\_Set\}$

     End do
  3. If Order(Ind_Set) > Order(Max_Ind_Set), $Max\_Ind\_Set \leftarrow Ind\_Set$
- End do

vertices of high degree that were adjacent to other vertices of high degree, the independent set potential identifies vertices of low degree that are not adjacent to other vertices of low degree, and which are therefore likely to form a large independent set. As in the Clique Potential Algorithm, the current independent set is initialized to a single vertex. The subgraph induced by the set of all vertices *not* adjacent to the chosen vertex is determined. The independent set is then augmented by the vertex of maximum independent set potential, and the subgraph induced by the set of all vertices not adjacent to these two vertices is determined. This process is continued till the independent set can be augmented no further. It is repeated at every vertex, and the largest independent set that is found is finally output.

Since the complement of an instance of the random graph $G_{n,.5}$ is another instance of $G_{n,.5}$, the distribution of the size of the maximum clique must be identical to that of the maximum independent set, and consequently, the performance of the Clique Potential Algorithm on $G_{n,.5}$ must be statistically identical to that of the Independent Set Potential Algorithm. Consequently, if we reinterpret $f(n,k)$ to be the fraction of trials in which the Independent Set Potential Algorithm finds an independent set of size k in $G_{n,.5}$, $\mu(n,k)$ and $\sigma^2(n,k)$ to be the mean and the

variance of the number of independent sets of size k in $G_{n,.5}$, and $p(n,k)$ to be a lower bound on the probability that an independent set of size k exists in in $G_{n,.5}$, Table 1 gives us the performance of the Independent Set Potential Algorithm on $G_{n,.5}$. As the Independent Set Potential Algorithm is essentially an application of the Clique Potential Algorithm to the complement of a graph, and as the complement of a graph contains $|V|(|V|-1)/2-|E|$ edges, the following result is an immediate consequence of Theorem 1.

Theorem 2:

The Independent Set Potential Algorithm runs in $O(|V|^2(\binom{|V|}{2}-|E|))$ time. ∎

III. Two Preprocessing Algorithms

Since the Graph Coloring and Maximum Clique problems are NP-complete, the worst case time complexity of all known algorithms to determine the chromatic number or the clique number of a graph is at least exponential in $|V|$ [11, 13, 22, 50, 54]. Consequently, even a small reduction in the size of a graph can significantly speed up an exact solution to these problems. In this section, we propose and examine two preprocessing algorithms that take as their input a graph, and output a graph that is (possibly) smaller than that input to them. The chromatic number of the reduced graph is no larger than that of the original graph, while the clique number of the reduced graph is identical to that of the original graph. The graph coloring and maximum clique problems can be solved (usually at a substantial saving in computational effort) on these reduced graphs. In the case of the graph coloring problem, the solution on the reduced graph is extended to the entire graph.

We first examine the graph coloring problem. The method is based on the observation that if one has a lower bound, $\hat{\chi}$, on the chromatic number of a graph, one can construct a minimal coloring of the graph from a minimal coloring of any subgraph created by the removal of all vertices of degree $\hat{\chi}-1$ or less (including those vertices whose degree is reduced to $\hat{\chi}-1$ or less by the removal of their neighbors). The reduced subgraph is first colored, and the removed vertices are then added back to the graph and assigned an available color in the inverse order of their removal. As a vertex of degree less than $\hat{\chi}$ must have fewer than $\hat{\chi}$ distinct colors adjacent to it, there is is always a color available to color it with. A minor variant of this procedure that does not use an explicit lower bound on the chromatic number has been used for many years as a coloring algorithm in register allocation routines for optimizing compilers [17], but it seems not to have been noticed that the introduction of a lower bound can lead to a very useful preprocessing algorithm.

Preprocessing Algorithm A

- Run the Clique Potential Algorithm
- Let $\hat{cl}(G)$ be the size of the largest clique found
- $Removed(i) \leftarrow 0,\ 1 \leq i \leq |V|$
- $n\_removed \leftarrow 0$
- Do While vertices of degree $< \hat{cl}(G)$ remain
  1. Choose a vertex v of degree $< \hat{cl}(G)$
  2. $Removed(n\_removed) \leftarrow v$
  3. $G \leftarrow G - v$
  4. $n\_removed \leftarrow n\_removed + 1$
  5. For each $u \in N(v)$, set $d(u) \leftarrow d(u) - 1$
- End Do
- Input the remaining graph to an exact coloring algorithm
- For $i = 1 \ldots n\_removed$

  Color $Removed(i)$ with any permissible color. If $n\_removed = |V|$, use any permissible color $\leq \hat{cl}(G)$.

Some investigation of graphs encountered in practice shows that they tend to be sparse, and that their clique number is a tight lower bound on their chromatic number. The Clique Potential Algorithm can be used to approximate the clique number, providing the required lower bound. The embodiment of these ideas is called Preprocessing Algorithm A.

It is important to note it is possible to construct simple examples on which Preprocessing Algorithm A performs no reduction at all. One such example is the complete bipartite graph $K_{n,n}$, $n \geq 3$, in which two sets of n vertices are joined by edges as follows. Each vertex in the first set is joined by an edge to each vertex in the second set. No two vertices that belong to the same set are joined by an edge. Clearly, the clique number of such a graph is 2 (as is its chromatic number), and a clique of this size will be found by the Clique Potential Algorithm. However, all vertices have degree 2 or more, and the algorithm fails to remove any vertices. It not difficult to show a much stronger result: the algorithm fails to remove any vertices on the random graph $G_{n,p}$ for any fixed $p > 0$, and consequently, fails to remove any vertices on almost every graph. This is the subject of our next theorem. The proof is deferred to the appendix.

Theorem 3:

On the random graph $G_{n,p}$, Preprocessing Algorithm A does not remove any vertices with probability $1 - o(1)$.

■

As $G_{n,.5}$ is a probability space in which all graphs on n vertices occur with equal probability (i.e. with probability $2^{-n(n-1)/2}$, it follows that Preprocessing Algorithm A will perform no reduction at all on almost every graph with probability $1 - o(1)$. It is very important to note that the fact that the algorithm fails on almost every graph does not imply that the algorithm will fail with probability $1 - o(1)$ on graphs that arise in practical applications. The random graph $G_{n,p}$ has $O(|V|^2)$ edges, and is consequently a poor model for graphs that arise in most practical applications [39]. Experience shows such graphs to be fairly sparse and to be comprised of interconnected "clumps" of vertices. The uniform random graph $U_{n,r}$, better captures this structure [39]. We therefore tested Preprocessing Algorithm A on the uniform random graph $U_{1000,r}$ with r ranging from .02 to .1. The corresponding mean vertex degree varies between 1.26 and 31.42. The mean and the variance of the number of cliques of any given size in $U_{n,r}$ is not known, as the edges are not chosen independently. Ten thousand graphs were generated for each value of r. Table 2 contains the results of these tests.

As can be seen from Table 2, for random graphs with $r \leq .06$ the preprocessing step decomposes the graph to the empty graph (thus allowing an exact coloring) over 85% of the time. Furthermore, the mean and the minimum of the number of vertices that remain in graphs that are not completely reduced are closely spaced. The algorithm was further tested on four graphs that were generated by a component placement program. The first graph had 448 vertices and 7374 edges, the second had 367 vertices and 3850 edges, the third had 326 vertices and 1528 edges, and the fourth had 397 vertices and 2200 edges. All four graphs were substantially reduced in size.

A question of interest when the algorithm terminates without reducing the graph to the empty graph is the following: Does the algorithm terminate because its estimate of the clique number is less than the true clique number, or because the degree of every vertex in the reduced graph is greater than or equal to the clique number? A complete answer to this question necessitates the exact solution of the maximum clique problem on all the reduced graphs. As this is computationally infeasible, we reran the Clique Potential Algorithm on the remaining graph to see if it could find a larger clique. On the 12,280 graphs thus examined a larger clique was not ever found. While this is no means a definitive test, it would seem to indicate that the Clique Potential Algorithm usually finds the maximum clique. In addition, a number of small graphs were exhaustively searched for larger cliques: none was ever found. Interestingly, all the graphs

that were searched exhaustively had a common property: their chromatic number was larger (usually by one or two) than their clique number. As it is difficult to construct small graphs with this property, Preprocessing Algorithm A may find use in the generation of such graphs. One could input a large number of random graphs to the algorithm, discard those graphs that were reduced to the empty graph, and accept those graphs that were not completely reduced. Some final testing would have to be done to ensure that the graph did indeed have the required property, but this should not be a problem for graphs of moderate size.

In view of these results, we advocate the routine use of the algorithm to preprocess graphs that are to be colored. Note that we are not obliged to use an exact coloring algorithm on the reduced graph. If a quick coloring is desired, or if the number of vertices that remains is too large to allow an exact coloring, we could input the reduced graph to an approximate coloring algorithm, and reconstruct the graph in exactly the same way. There would be no guarantee of the optimality of the coloring, but if the reduced graph is small, experience shows that it is unlikely that the number of colors used is much larger than the chromatic number of the graph.

Next, we discuss the preprocessing of graphs for which the maximum clique problem is to be solved. If we have some lower bound $\hat{cl}(G)$ on the clique number of a graph G, we can remove any vertex of degree less than $\hat{cl}(G)-1$, as such a vertex cannot belong to a maximum clique. If the degree of a vertex is reduced to $\hat{cl}(G)-2$ by the removal of vertices adjacent to it, it too can be removed, as it could not have belonged to a maximum clique that contained any of its neighbors that were removed, and as its degree is now less than $\hat{cl}(G)-1$, it cannot belong to a maximum clique containing any of its remaining neighbors. The algorithm terminates in one of two ways:

1. It reduces the graph to a clique of size $\hat{cl}(G)$, or
2. It terminates without reducing the graph to a clique.

In the first case, the remaining clique is a maximum clique. In the second, the reduced graph can be input to a maximum clique algorithm. If the size of the reduced graph is substantially smaller than the size of the original graph, the saving in computational time can be substantial. Note that the Clique Potential Algorithm provides the lower bound on the clique number of the graph. The resulting algorithm is called Preprocessing Algorithm B. It too was tested on the same set of uniform random graphs and graphs drawn from a manufacturing application that Preprocessing Algorithm A was tested on, and the results of the tests are shown in Table 3.

n	r	Average degree	Fraction of graphs that are reduced to the empty graph	Min/Mean/Max number of vertices that remain in graphs that are not completely reduced
1000	.02	1.26	.9925	6 / 6.49 / 12
1000	.04	5.03	.9593	8 / 11.24 / 22
1000	.06	11.31	.8596	12 / 18.52 / 89
1000	.08	20.11	.6382	16 / 45.48 / 447
1000	.10	31.41	.3224	21 / 167.08 / 855
Graph	$\|V\|$	$\|E\|$	Number of remaining vertices	Number of remaining edges
G_1	448	7374	131	1512
G_2	367	3850	16	105
G_3	326	1528	0	0
G_4	397	2200	50	196

Table 2. Performance of Preprocessing Algorithm A on $U_{1000,r}$ and four graphs from a manufacturing application.

As can be seen from Table 3, even when the graph is not completely reduced, the mean number of vertices in the reduced graph is at least an order of magnitude less than the number of vertices in the original graph in all cases except the last, and even in this case it is reduced by a factor of 3 or so. Once again, the mean and the minimum of the number of remaining vertices are quite closely spaced, indicating that the probability that very few vertices are removed is small.

Clearly, a complete reduction of the graph takes place much more often when it preprocessed for the coloring problem than when it preprocessed for the maximum clique problem. This is a direct consequence of the fact that in the first case, vertices of degree $\hat{c}\mathrm{l}(G)-1$ or less can be

n	r	Average degree	Average size of largest clique found	Fraction of graphs that are reduced to a clique	Min/Mean/Max number of vertices that remain in graphs that are not completely reduced
1000	.02	1.26	4.60	.3638	5/ 26.25 / 99
1000	.04	5.03	7.83	.3239	8 / 26.12 / 181
1000	.06	11.31	11.56	.2366	11 / 38.78 / 291
1000	.08	20.11	15.83	.1376	16 / 93.41 / 740
1000	.10	31.41	20.66	.0516	21 / 268.86 / 926
Graph	$\|V\|$	$\|E\|$	Size of largest clique found	Number of remaining vertices	Number of remaining edges
G_1	448	7374	13	179	1985
G_2	367	3850	11	49	336
G_3	326	1528	11	11	55
G_4	397	2200	6	139	546

Table 3. Performance of Preprocessing Algorithm B on $U_{1000,r}$ and four graphs from a manufacturing application.

discarded, while in the second, only vertices of degree $\hat{c}\hat{l}(G)-2$ or less can be discarded. Even so, the reduction in the size of these graphs is substantial enough to make the routine use of the preprocessing algorithm worthwhile when the maximum clique problem is to be solved to optimality, as the size of the search trees generated by exact maximum clique algorithms grows exponentially with the number of vertices in the graph. The following corollary to Theorem 2 follows immediately.

Corollary:

On the random graph $G_{n,p}$, the preprocessing phase of Preprocessing Algorithm B does not remove any vertices with probability $1 - o(1)$.

■

The complexity of these preprocessing Algorithms is the subject of the following theorem.

Preprocessing Algorithm B

- Run the Clique Potential Algorithm
- Let $\hat{\text{cl}}(G)$ be the size of the largest clique found
- While vertices of degree $< \hat{\text{cl}}(G)$ remain
  1. Choose a vertex v of degree $< \hat{\text{cl}}(G) - 1$
  2. $G \leftarrow G - v$
  3. For each $u \in N(v)$, set $d(u) \leftarrow d(u) - 1$
- If the remaining graph is a clique of size $\hat{\text{cl}}(G)$, it is a maximum clique
- Else input the remaining graph to an algorithm to find the maximum clique

Theorem 4:

Preprocessing Algorithms A and B runs in $O(|V|^2|E|)$ time.

Proof:

As the Clique Potential Algorithm must be run in order to determine a lower bound on the chromatic number, the complexity of the preprocessing phase is at least as large $O(|V|^2|E|)$. To remove vertices in the graph efficiently, first form a linked list of all vertices with degree less than $\tilde{\text{cl}}(G)$. This takes $O(|V|)$ time. Remove vertices from the graph starting at one end of the list. Each time a vertex v is removed, scan all of its neighbors, reduce their degrees by one, and remove v from their adjacency lists (or set the appropriate elements in the adjacency matrix to 0). If the degree of a neighbor drops to $\tilde{\text{cl}}(G) - 1$ as a result, add the neighbor to the other end of the linked list. Note that a vertex can be added to or deleted from the linked list at most once each. The cost of adding or deleting a vertex is a constant. While deleting vertex v, an additional $O(d(v))$ cost is incurred in zeroing the non-zero positions in the v^{th} column if the graph is stored as an adjacency matrix. If it is stored as a linked list, the adjacency lists of each of v's neighbors must be searched, and the cost of doing so is at most $O(\sum_{u \in N(v)} d(u)) = O(\sum_v d^2(v))$. This is easily shown to be $O(|V||E|)$. It follows that the total complexity is $O(|V|^2|E|)$.

■

IV. A New Technique for Approximate Graph Coloring

In this section we present a technique based on recursion to transform any given graph coloring algorithm into a new algorithm that is usually (though not necessarily) better that the algorithm from which it was derived. We first present the algorithm and then test its performance using two well known heuristic graph coloring algorithms on the random graph $G_{n,p}$.

The first heuristic algorithm is the Highest Degree First algorithm [48] in which vertices are colored in descending order of degree, with a vertex being assigned the smallest color (or the smallest positive integer) not used on an adjacent vertex. In practice, the list of vertices is sorted by degree, and vertices are colored in order of their appearance in the list. The second heuristic algorithm is the Highest Saturation Degree First algorithm [11]. The saturation degree of a vertex is defined to be the number of distinct colors used on adjacent vertices. For example, if a vertex is adjacent to five vertices that are colored blue, nine that are colored green, one that is colored red, and eleven uncolored vertices, it has a saturation degree of three. The Highest Saturation Degree First algorithm works by coloring the vertex of highest saturation degree, with ties being broken in favor of the vertex with the larger number of uncolored vertices adjacent to it. The algorithm is started by coloring the vertex of highest degree. A vertex of maximum saturation degree has the fewest colors available to color it, and by coloring it, one is staving off the potential need for a new color.

Much practical experience with approximate graph coloring algorithms has led us to make the following observation: The likelihood that an approximate coloring algorithm colors a graph to optimality tends to decrease with with the number of vertices and edges in the graph. While this is a very broad statement, and while counterexamples to it can clearly be found (for example, any reasonable algorithm can color the complete graph on n vertices or its complement for arbitrary n to optimality), experience gained from coloring graphs derived from practical applications in a wide range of engineering disciplines bears out the validity of this observation.

In view of this observation, we exploit the following three techniques to try and reduce the number of colors required by an underlying algorithm to color a graph. The Recursive Coloring Algorithm is the embodiment of these ideas. For convenience, we shall call the original algorithm the *Underlying* algorithm, and the new algorithm the *Derived* algorithm.

1. ***Recursion.*** The graph is first colored using the underlying algorithm. For each color class, the sum of the degrees of all the vertices in that color class is computed. Note that this is just the number of edges incident upon that color class. The color class with the maximum number

of edges incident upon it is removed (ties are broken in favor of the color class with the larger number of vertices), modified as detailed below, and set aside as one of the color classes in the final coloring. The above mentioned procedure is then repeated on the remainder of the graph (i.e. the remainder of the graph is colored with the underlying algorithm, the color class with the maximum number of edges is incident upon it is removed, modified as detailed below, and then set aside as one of the color classes in the final coloring). This process continues till all the vertices have been assigned to one color class or another. In using recursion, we are exploiting the property that a colored graph can be partitioned into a portion of a valid coloring and a new instance of the graph coloring problem.

The idea of using recursion is not new: it has previously been explored in [8] and [47]. Our approach differs in that we remove the color class with the most edges adjacent to it, breaking ties in favor of the larger color class, while in [8] and [47] they remove the color class with the most vertices, breaking ties in favor of the one with more edges adjacent to it. Their strategy works well on random graphs (as there are usually many color classes of maximum size), but works less well on graphs that are encountered in practice (where nothing is known about the size of the color classes). Large color classes often contain many vertices of low degree, and the removal of these vertices does not substantially lower the average degree of a vertex in the portion of the graph that remains.

2. ***Increase the number of edges in the color classes.*** At each pass of the recursive procedure, after selecting a color class to remove from the graph, we check to see if there are any vertices outside this color class that are adjacent to exactly one vertex in it, such that the degree of the vertex outside the chosen color class is greater than the degree of the vertex within it. A list is formed of all such pairs of vertices. The pair for which the difference in degrees between the vertex lying outside the chosen color class and that in the color class is maximized is exchanged i.e. the vertex in the chosen color class is replaced by the vertex outside it, provided of course that such a pair exists. This process is continued till all possible exchanges have been performed, and is done to reduce the average degree of a vertex in the portion of the graph that remains to be colored.

3. ***Ensure that all color classes are maximal.*** After exchanging vertices so as to decrease the number of edges remaining in the graph, a list of all vertices that are not adjacent to any member of the chosen color class is formed, and the vertex of maximum degree (if one exists) is added to the chosen color class. This procedure is repeated till the chosen color class is a maximal independent set. It is well known [57] that every graph can be colored so that all the color classes are maximal, and this technique forms the basis of a number of coloring algorithms, both exact [22] and and approximate [38]. We adopt it because it reduces the

number of vertices that remain in in the graph. As vertices of high degree are removed first, the remaining portion of the graph tends to be somewhat sparser.

We next explore the complexity of these procedures.

Theorem 5:

The complexity of step 4 in the Recursive Coloring Algorithm (Increase the number of edges in the color classes) is $O(|V||E|)$.

Proof:

Consider a color class on which the exchange procedure is to be performed. As only exchanges that increase the degree sum of the vertices in the color class are permitted, a vertex can be exchanged with each of its neighbors at most once, and therefore we can have at most $|E|$ exchanges. The set of all vertices for which exchanges are permissible can be determined in $O(|E|)$ time, and each time a vertex u is exchanged with vertex v, the set can be updated in $O(d(u) + d(v))$ time, so that the complexity of maintaining the set is $O(|E|)$. Finally, note that there are at most $|V|$ color classes, giving us the final result.

■

Theorem 6:

The complexity of step 5 in the Recursive Coloring Algorithm (Ensure that all color classes are maximal) is $O(|V|^2)$.

Proof:

Let the sequence of color classes generated by the Recursive Coloring Algorithm just before we process them to ensure that they are maximal be given by $I_1, I_2, \ldots I_{\hat{\chi}}$. Let the vertices added to I_j to turn it into a maximal independent set be given by δ_j. The resulting maximal color classes C_j are given by $C_j = I_j \cup \delta_j$, $1 \le j \le \hat{\chi}$. δ_j can be determined in at most $O(|V| + \sum_{v \in C_j} d(v))$ time. Summing over j gives us a total complexity of $O(|V| \times \hat{\chi} + |E|) = O(|V|^2)$.

■

In practice, these procedures run substantially faster than Theorems 5 and 6 would indicate, as exchanges tend to be infrequent, and when they do occur, at most one or two exchanges are performed. Since the Recursive Coloring Algorithm does not place any restrictions on the underlying algorithm, a derived algorithm may be thought of as an underlying algorithm and the recursion performed over it, leading to a doubly recursive algorithm. In fact, this procedure can be cascaded indefinitely to create an infinite family of graph coloring algorithms from any given underlying algorithm. In our next theorem we show that on any graph this process must terminate in one of two ways:

1. A minimal coloring is found, or

The Recursive Coloring Algorithm

- Initialization. $i \leftarrow 0$
- Main loop. While $G \neq \phi$
  1. $i \leftarrow i + 1$
  2. Color G with the underlying algorithm
  3. Find C, the color class for which the sum of the degrees of all the vertices in C is maximized. Break ties in favor of the color class with more vertices.
  4. Increase the Number of edges in the color class. Do
     a. $S \leftarrow \{[u,v] | \ u \in G - C, \ v \in C, \ (u,v) \in E, \ (u,w) \notin E \ \forall w \neq v \in C \text{ and } (d(u) > d(v))\}$
     b. Find $[u_{max}, v_{max}]$ such that $d(u_{max}) - d(v_{max}) \geq d(u) - d(v) \ \forall [v,u] \in S$
     c. $C \leftarrow C - v_{max} + u_{max}$

     While $|S| > 0$
  5. Ensure that the color class is maximal. Do
     a. $S \leftarrow \{u | \ u \in G - C, \ (u,v) \notin G \ \forall v \in C\}$
     b. Find u_{max} such that $d(u_{max}) \geq d(u) \ \forall u \in S$
     c. $C \leftarrow C + u_{max}$

     While $|S| > 0$
  6. Color C with Color i.
  7. Set $G \leftarrow G - C$
- End do (Main Loop)

2. The algorithm gets stuck at a suboptimal coloring and increasing the depth of the recursion produces no improvement at all.

For notational convenience, denote the underlying algorithm by A^0, the algorithm derived by applying the Recursive Coloring Algorithm to $A^{i-1}, i \geq 1$ by A^i, and the coloring of G generated by A^i by $C^i(G)$.

Theorem 7:

For any graph G, and for any underlying algorithm A^0, there exists a value of i such that $C^j(G) = C^i(G), \forall j > i$.

Proof:

As the set of all possible color classes (or independent sets) in G is finite, there is at least one color class for which the sum of the degrees of the vertices in it is maximum. If the color class with maximum sum of vertex degrees generated by A^j is replaced by some other color class by A^{j+1}, it must be that the sum of the degrees of the vertices in the second color class is greater than that in the first. As there are a finite number of color classes, and as the sum of the degrees of the vertices in each color class is bounded, the process must terminate, so that for all $j > j^*$ the color class with the largest sum of vertex degrees in $C^j(G)$ is the same as the color class with the largest sum of vertex degrees in $C^{j^*}(G)$. The same argument can now be applied to the color class with the second largest degree sum, and so on, till all the color classes have been accounted for.

■

Note that the procedure need not terminate with the determination of the color class for which the sum of degrees is maximized. The underlying algorithm can generate a coloring of a graph that simply cannot be improved by these techniques, and for which the sum of the degrees of the vertices in any color class is less than maximum. Such an example can be found in [38]. The graph in question is a bipartite graph on $2n$ vertices labelled $1,2,\ldots,n$ and $1',2',\ldots,n'$. Vertex i is joined to vertex j', $j \neq i$. When colored using the Highest Degree First algorithm in the sequence $1, 1', 2, 2', \ldots, n, n'$, color i will be assigned to vertices i and i', requiring $|V|/2$ colors to color the graph. No exchanges that increase the number of edges leaving a color class are possible, and all the color classes are maximal, so that $C^i(G) = C^0(G), \forall j > 0$. The graph can however be colored with two colors, the color classes being given by $\{i\}$ and $\{i'\}$, $1 \leq i \leq |V|/2$. In this case, no benefit accrued from using the Recursive Coloring algorithm. This is, however, atypical: the derived algorithm usually uses fewer colors than the underlying algorithm.

The last example immediately raises the question of whether a derived algorithm can use *more* colors than its corresponding underlying algorithm does to color a graph. The answer is that this is possible but unlikely to occur in practice. In coloring many thousands of graphs, we have come across very few graphs and coloring algorithms that exhibited this behavior. Table 4 contains one example of this behavior: on the random graph $G_{100,.25}$, a doubly recursive algorithm based on the Highest Saturation Degree First algorithm uses close to one color more, on average, than the singly recursive algorithm that it was derived from. This can be explained by observing that the removal of a color class may create a graph on which the underlying algorithm performs poorly. To remedy this problem, the colorings generated by the underlying algorithm and the singly recursive algorithm can be preserved and compared to that generated by the doubly recursive algorithm. The best of these three colorings can then be output. The following example shows that the worst case behavior of the Recursive Coloring Algorithm can be very bad- we can construct a 3 colorable graph which is correctly colored by the Highest Degree First algorithm,

but for which the algorithm derived from the Highest Degree First algorithm uses $O(|V|)$ colors. The graph in question is a tripartite graph with $3n$ vertices and $n(3n-2)$ edges that is constructed as follows. The vertices are partitioned into 3 sets labelled $1, \ldots, n$; $1', \ldots, n'$, and $1'', \ldots, n''$. Edges join vertex i, $1 \le i \le n$ to all vertices j'', $1 \le j < n$. Also joined by edges are vertices i'' and j', $1 \le j \ne i \le n$, and i and j', $1 \le j \ne i \le n$. Vertices $\{i\}$ and $\{i''\}$ have degree $2n-1$, while vertices $\{i'\}$ have degree $2(n-1)$.

The Highest Degree First algorithm will color vertices $\{i\}$ and $\{i''\}$ before coloring vertices $\{i'\}$. As the subgraph induced by $\{i\} \cup \{i''\}$ is a complete bipartite graph, it will two color this subgraph, and then proceed to color $\{i'\}$ with color 3. This coloring is clearly optimal, as the vertices i, i' and i'', $1 \le i \le n$ form a clique of size 3. The color classes $\{i\}$ and $\{i''\}$ have the same sum of vertex degrees, and this is larger than the sum of the degrees of the vertices contained in the color class $\{i'\}$, and consequently the Recursive Coloring Algorithm will remove one of the first two color classes. Assume that it is the first one, i.e. the color class containing all the vertices $\{i\}$, that is removed. No exchanges that increase the degree sum are possible, and as all the color classes are maximal, the set of vertices that is removed is precisely $\{i\}$. The reduced graph is precisely the graph described earlier to color which the Highest Degree First algorithm used $|V|/2$ colors. It has $2n$ vertices and $n(n-1)$ edges, with vertex i', $1 \le i \le n$ being adjacent to vertex j'', $1 \le j \ne i \le n$. If this graph is colored in the sequence $1', 1'', 2', 2'', \ldots, n', n''$, the Highest Degree First algorithm will use n colors to color it. As noted earlier, no exchanges are possible, and consequently the derived algorithm will use $n+1$ colors to color the graph.

In order to gain some insight into the benefits conferred by the use of the Recursive Coloring Algorithm, singly and doubly recursive versions of the two graph coloring algorithms described earlier were coded, and tests were carried out on the random graphs $G_{n,.5}$ and $G_{n,.25}$ with n ranging between 100 and 1000. No use was made of Preprocessing Algorithm A.

In view of the large amount of data to be presented, we follow our earlier notation and label the underlying algorithms A^0, and singly and doubly recursive algorithms by A^1 and A^2. The mean number of colors used by each of the algorithms to color the random graphs $G_{n,.5}$ and $G_{n,.25}$ is shown in Table 4. The maximum and minimum number of colors used by all the algorithms deviated very little from the mean, this being especially true of the derived algorithms, for which the maximum and minimum number of colors used did not ever differ by more than three, and in most cases did not differ by more than two. The number of graphs colored varied with n. For $n \le 200$, we tested each algorithm on 1000 instances of $G_{n,p}$ for each of the two

		Highest Degree First Algorithm			Highest Saturation Degree First Algorithm		
n	p	A^0	A^1	A^2	A^0	A^1	A^2
100	.5	19.02	17.97	17.00	17.04	16.02	16.01
100	.25	11.00	10.00	10.00	10.00	9.02	9.99
200	.5	33.98	28.99	27.02	31.00	28.00	27.00
200	.25	18.99	15.03	15.02	16.01	15.01	15.00
400	.5	57.15	48.99	46.95	53.98	47.96	45.06
400	.25	30.98	26.04	25.97	27.95	25.97	25.02
800	.5	101.94	85.08	79.45	95.94	83.85	78.92
800	.25	52.67	44.97	43.07	47.94	43.23	42.07
1000	.5	123.41	102.11	96.02	115.79	100.87	94.81
1000	.25	62.69	53.90	51.85	56.30	52.07	50.12

Table 4. Mean number of colors used by various algorithms to color $G_{n,.5}$

values of p. For $n = 400$ we tested each algorithm on 300 instances of each random graph, and for $n > 400$ we tested each algorithm on 100 instances of each random graph.

On looking at Table 4, we see that this recursive procedure almost always reduces the number of colors used by an underlying algorithm to color large random graphs. On the classic test case of $G_{1000,.5}$, a single application of the Recursive Coloring Algorithm reduced the number of colors required in both cases by between 15 and 20, and a second iteration further reduced the number of colors required by about 6. Clearly, a law of diminishing returns sets in rather quickly. To put these numbers in perspective, Bollobas and Thomason (1985) and Morgenstern (1989) have shown that the chromatic number of $G_{1000,.5}$ is at least 80 with probability $> 1 - 10^{-12}$ and

argue using evidence derived from the distribution of the size of independent sets in $G_{n,p}$ that it most likely lies in the the range of 84 to 86.

We know of no algorithms other than those specifically designed to exploit the statistical properties of $G_{n,p}$ [10, 47,]) which color large random graphs in fewer colors than the doubly recursive version of the Highest Saturation Degree First Algorithm. For most practical applications, however, this is most likely overkill, and a singly recursive algorithm will meet the need at hand at a substantial saving in CPU time. A singly recursive algorithm has the further important advantage of not needing to store a copy of the graph (which is destroyed as the algorithm progresses). Doubly recursive algorithms require the storage of the graph as each call to the singly recursive algorithm destroys the copy of the graph that is passed to it. The graph can, however, be stored on a disk instead of in main memory with little detriment to the speed of the algorithm, as the graph must be recalled only when the singly recursive algorithm is called.

From Table 4 we see that the number of colors used by the derived algorithms is only weakly dependent on the number of colors used by the underlying algorithms, which suggests that in practice one ought to use simple underlying algorithms, as the extra CPU time used by a complex underlying algorithm is unlikely to result in a better coloring. We would suggest the use of a singly recursive version of the Highest Saturation Degree First algorithm for most applications. It uses fewer colors than the Highest Degree First algorithm without requiring exorbitant amounts of computation, and shows very little further improvement after one iteration. $O(|E| \log |V|)$ implementations of the Highest Saturation Degree First algorithm are described in [49] and [60]. A $\Theta(|E|)$ implementation that runs faster on sparse graphs is also described in [49], though it is substantially more complex.

V. Conclusions

We have presented some new techniques for approximately solving the maximum clique, maximum independent set and graph coloring problems. Extensive tests show these algorithms to work extremely well in practice. A number of open questions remain, and are under investigation. Among these are

1. What is the the expected performance of the Clique Potential Algorithm on the random graph $G_{n,p}$? In particular, can it be shown that the Clique Potential Algorithm finds cliques of size at least $\mathrm{cl}(G)/2$?
2. Similarly, what can be said of the performance of the Recursive Coloring Algorithm when applied to some given underlying algorithm (say the greedy algorithm) on random graphs?

(We suggest the greedy algorithm because it is the simplest possible algorithm, and because its performance on $G_{n,p}$ has been analyzed [32]).

3. Is it possible to characterize the set of graphs for which a derived algorithm requires more colors than its underlying algorithm?

Acknowledgements

I am deeply indebted to Marty Golumbic and Randy Nelson for their many suggestions that led to substantial improvements in the content and presentation of this paper.

References

1. Balas, E., Chang, S.Y., Finding a Maximum Clique in an Arbitrary Graph, SIAM J. Computing, Vol. 15, No. 4, pp. 1054-1068, 1986.

2. Balas, E., Chang, S.Y., On Graphs with Polynomially Solvable Maximum-Weight Clique Problem, Networks, Vol. 19, pp. 247-253, 1989.

3. Balas, E., Chvatal, V., and Nesetril , J., On the Maximum Weight Clique Problem, Math. Oper. Res., Vol. 12, No. 3, pp. 522-535, 1987.

4. Berger, B., and Rompel, J., A Better Performance Guarantee for Approximate Graph Coloring, Algorithmica, to appear.

5. Blum, A., An $\tilde{O}(n^{.4})$ Approximation Algorithm for 3-Coloring (and Improved Approximation Algorithms for k −Coloring), Proc. ACM Symp. on Theory of Computing, May 1989.

6. Blum, A., Some Tools for Approximate 3-Coloring, Preprint, 1990.

7. Bolles, R., C., Robust Feature Matching Through Maximal Cliques, Proc. Soc. Photo-Optical Instrumentation Engineers, Vol. 182, pp. 19-20, 1979.

8. Bollobas, B., ***Random Graphs***, Academic Press, London, 1985.

9. Bollobas, B., and Erdos, P., Cliques in Random Graphs, Math. Proc. Cambridge Phil. Soc., Vol. 80, pp. 419-427, 1976.

10. Bollobas, B., and Thomason, A., Random Graphs of Small Order, Annals of Discrete Math., Vol. 28, pp. 47-97, 1985.

11. Brelaz, D., New Methods to Color the Vertices of a Graph, CACM, Vol. 22, pp. 251-256, 1979.

12. Brelaz, D., Nicolier, Y., and de Werra, D., Compactness and Balancing in Scheduling, Zeit. fur Oper. Res., Vol. 21, pp. 65-73, 1977.

13. Brown, J.R., Chromatic Scheduling and the Chromatic Number Problem, Management Science, Vol. 19, No. 4, pp. 456-463, 1972.

14. Burns, J.E., The Maximum Independent Set Problem for Cubic Planar Graphs, Networks, Vol. 19, pp. 373-378, 1989.

15. Campers, G., Henkes, O., and Leclerq, J.P., Graph Coloring Heuristics: a Survey, Some New Propositions and Computational Experiences on Random and 'Leighton's' Graphs, Proc. 11th Intnl. Conf. on Operations Research, pp. 917-932, 1988.

16. Carter, M.W., A Survey of Practical Applications of Examination Timetabling Algorithms, Operations Research, Vol. 34, No. 2, pp.193-202, 1986

17. Chaitin, G.J., Register Allocation and Spilling via Graph Coloring, Proc. SIGPLAN'82 Symposium on Compiler Construction, pp.98-105, 1982.

18. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., and Markstein, P.W., Register Allocation via Coloring, Computer Languages, Vol. 6, No. 1, pp. 47-57, 1981.

19. Chams, M., Hertz, A., and de Werra, D., Some Experiments with Simulated Annealing for Coloring Graphs, European J. Oper. Res., Vol. 32, pp. 260-266, 1987.

20. Choukhmane, E., and Franco, J., An approximation algorithm for the Independent Set Problem on Planar Graphs, Siam J. Computing, Vol. 11, pp. 663-675, 1982.

21. Chow, F., and Hennessy, J., Register Allocation by Priority-based Coloring, Proc. SIGPLAN'84 Symposium on Compiler Construction, pp. 222-232, 1984.

22. Christofides, N., An Algorithm for the Chromatic Number of a Graph, Computer J., Vol. 14, pp.38-39, 1971.

23. De Werra, D., An Introduction to Timetabling, European J. Oper. Res., Vol. 19, pp. 151-162, 1985.

24. Dieu, P.D., Thanh, L.C., and Hoa, L.T., Average Polynomial Time Complexity of Some NP-Complete Problems, Theoretical Computer Science, Vol. 46, pp. 219-237, 1986.

25. Dutton, R.D., and Brigham, R.C., A New Graph Coloring Algorithm, Computer J., Vol. 24, pp.85-86, 1981.

26. Dyer, M.E., and Frieze, A.M., The Solution of Some Random NP-Hard Problems in Polynomial Expected Time, J. Algorithms, Vol. 10, pp. 451-489, 1989.

27. Escudero, L., IBM Research Report RCxxxx, 1990.

28. Friden, C., Hertz, A., and de Werra, D., STABULUS: A Technique for Finding Stable Sets in Large Graphs with Tabu Search, Computing, Vol. 42, pp. 35-44, 1989.

29. Garey, M.R., and Johnson, D., ***Computers and Intractability: A Guide to the Theory of NP Completeness***, W.H. Freeman, San Francisco, 1979.

30. Gavril, F., Algorithms on Circular Arc Graphs, Networks, Vol. 4, No. 4, pp. 357-369, 1974.

31. Gilbert, E.N., Random Plane Networks, SIAM J., Vol. 9, pp. 533-543, 1961.

32. Grimmet, G.R., and McDiarmid, C.J.H., On Colouring Random Graphs, Math. Proc. Cambridge Phil. Soc., Vol. 77, pp. 313-324, 1975.

33. Gupta, U.I., Lee, D.T., and Leung, J.Y.T, Efficient Algorithms for Interval Graphs and Circular Arc Graphs, Networks, Vol. 12, No. 4, pp. 459-467, 1982.

34. Hertz, A., and de Werra, D., Using Tabu Search Techniques for Graph Coloring, Computing, Vol. 39, pp. 345-351, 1987.

35. Horaud, R., and Skordas, T., Structural Matching for Stereo Vision, Proc. 9th Intnl. Conf. on Pattern Recognition, pp. 14-17, 1988.

36. Hsu, W.L., Recognizing Planar Perfect Graphs, JACM, Vol. 34, No. 2, pp. 255-288, 1987.

37. Hsu, W.L., How to Color Claw-free Perfect Graphs, in ***Studies on Graphs and Discrete Programming***, North Holland, Amsterdam, 1981.

38. Johnson, D.S., Worst Case Behavior of Graph Coloring Algorithms, Proc. 5th Southeastern Conf. Combinatorics and Graph Theory, pp.513-527, 1974.

39. Johnson, D.S., Aragon, C.A., McGeogh, L.A., Schevon, C., Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring, Oper. Res., to appear.

40. Karmarkar, N., Karp, R.M., Lueker, G.S., and Odlyzko, A.M., Probabilistic Analysis of Optimum Partitioning, Journal of Applied Probability, Vol. 23, pp. 626-645, 1986.

41. Kucera, L., Graphs With Small Chromatic Number are Easy to Color, Information Processing Letters, Vol.30, pp. 233-236, 1989.

42. Korman, S.M., The Graph Coloring Problem, in ***Combinatorial Optimization***, Christofides, N., Mingozzi, A., Toth, P., and Sandi, C., (eds.), pp 211-235, 1979.

43. Larus, P., and Hilfinger, P., Register Allocation in the SPUR Lisp Compiler, Coloring, Proc. SIGPLAN'86 Symposium on Compiler Construction, pp. 222-232, 1986.

44. Leighton, F.T., A Graph Coloring Algorithm for Large Scheduling Problems, J. Res. Natnl. Bureau of Standards, Vol. 84, No. 6, pp. 479-506, 1979.

45. Matula, D.W., Expose-and-Merge Exploration and the Chromatic Number of a Random Graph, Combinatorica, Vol. 7, No. 3, pp. 275-284, 1987.

46. Matula, D.W., The Employee Party Problem, Notices Amer. Math. Soc., Vol. 19, pg. A-382, 1972.

47. Matula, D.W., and Johri, A., Probabilistic Bounds and Heuristic Algorithms for Coloring Large Random Graphs, Tech. Report 82-CSE-6, Southern Methodist University, Dallas, 1982.

48. Matula, D.W., Marble, G., and Isaacson, J.D., Graph Coloring Algorithms, in ***Graph Theory and Computing***, R.C. Read, editor, Academic Press, New York, 1972.

49. Morgenstern, C.A., Algorithms for General Graph Coloring, Tech. Report No. CS89-16, University of New Mexico, Albuquerque, 1989.

50. Peemoller, J., A Correction to Brelaz's Modification of Brown's Coloring Algorithm, CACM, Vol. 26, No. 8, pp. 595-597, 1983.

51. Petford, A.D., and Welsh, D.J.A., A Randomized 3-Coloring Algorithm, Discrete Math., Vol. 74, pp. 253-261, 1989.

52. Philips, T.K., Panwar, S.S., and Tantawi, A.N., Connectivity Properties of a Packet Radio Network Model, IEEE Trans. Info. Theory, Vol. 35, No. 5, pp. 1044-1047, 1989.

53. Robson, J.M., Algorithms for Maximum Independent Sets, Journal of Algorithms, Vol. 7, pp. 425-440, 1986.

54. Roschke, S.I., and Furtado, A.L., An Algorithm for Obtaining the Chromatic Number and an Optimal Coloring of a Graph, Information Processing Letters, Vol. 2, pp. 34-38, 1973.

55. Shamir, E., and Upfal, E., Sequential and Distributed Graph Coloring Algorithms with Performance Analysis in Random graph Spaces, J. Algorithms, Vol. 5, pp. 488-501, 1984.

56. Shindo, M., and Tomita, E., A Simple Algorithm for Finding a Clique and its Worst Case Time Complexity, Trans. Inst. Electron. Inf. Comm. Eng. D (Japan), Vol J71D, No. 3, pp. 472-481, 1988.

57. Syslo, M.M., Deo, N., and Kowalik, J.S., ***Discrete Optimization Algorithms***, Prentice Hall, Englewood Cliffs, 1983.

58. Tarjan, R.E., and Trojanowski, Finding a Maximum Independent Set, SIAM J. Computing, Vol. 6, pp. 537-546, 1977.

59. Tseng, C., and Siewiorek, D.P., Automated Synthesis of Data Paths in Digital Systems, IEEE Trans. on Computer Aided Design, Vol. CAD-5, No. 3, pp. 379-395, 1986.

60. Turner, J.S., Almost all k-Colorable Graphs are Easy to Color, J. Algorithms, Vol. 9, pp.63-82, 1988.

61. Welsh, D.J.A., and Powell, M.B., An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problems, Computer J., Vol. 10., pp. 85-87, 1967.

62. Widgerson, A., Improving the Performance Guarantee for Approximate Graph Coloring, JACM, Vol. 10, pp. 729-735, 1983.

63. Wing, O., Automated Gate Matrix Layout, Proc. Intntl. Symposium on Circuits and Systems, pp. 10-12, 1982.

64. Wood, D.C., A Technique for Coloring a Graph Applicable to Large Scale Timetabling Problems, Computer J., Vol. 12, pp.317-319, 1969.

APPENDIX

Proof of Theorem 3:

Denote the minimum degree of a vertex in $G_{n,p}$ by $d_{\min}$. As a vertex can be removed only if its degree is less than the clique number, we need only show that the probability that $d_{\min}$ is less than the clique number of $G_{n,p}$ is $o(1)$.

$$\begin{aligned}
\Pr[d_{\min} < \mathrm{cl}(G)] &= \sum_{k=1}^{n} \Pr[d_{\min} < k \text{ and } \mathrm{cl}(G) = k] \\
&\leq \sum_{k=1}^{np/2} \Pr[d_{\min} < k] + \sum_{k=np/2}^{n} \Pr[\mathrm{cl}(G) = k] \\
&\leq \sum_{k=1}^{np/2} n\Pr[\text{ Vertex 1 has degree } < k] + \sum_{k=np/2}^{n} \mathrm{E}[\text{Number of cliques of size } k]
\end{aligned}$$

The degree of a vertex is binomially distributed with parameters $n-1$ and p. The tail of the distribution can be bounded using the Chernoff bound, giving

$$\Pr[d(v) < np/2] < e^{-np/8},$$

and the expected number of cliques of size k is given by $\binom{n}{k} p^{k(k-1)/2}$, so that

$$\begin{aligned}
\Pr[d_{\min} < \mathrm{cl}(G)] &\leq \sum_{k=1}^{np/2} ne^{-np/8} + \sum_{k=np/2}^{n} \binom{n}{k} p^{k(k-1)/2} \\
&< O(n^2 e^{-np/8}) + n2^n p^{\frac{np}{2}(\frac{np}{2}-1)} \\
&= O(n^2 e^{-np/8}) + n\left(2p^{\frac{-p}{2}} p^{np^2/4}\right)^n \\
&= o(1).
\end{aligned}$$

■