INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Maximum Cliques and Vertex Colorings in Random and k-partite Graphs

Thomas Hughes

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

November 2000

ABSTRACT

Maximum Cliques and Vertex Colorings in
Random and k-partite Graphs

Thomas Hughes

In this project we search for maximum cliques in some random graphs and random k-partite graphs, using either the greedy coloring algorithm or the Dsatur coloring algorithm as bounding functions. We compare the efficiency of the algorithms by looking at the sizes of the resulting state space trees, as well as the number of colors required by the respective coloring algorithms. Finally, we implement a bichromatic interchange algorithm in an attempt to improve the coloring algorithms.

Acknowledgments

I would like to thank my supervisor, Professor Clement Lam, for his support, guidance, and encouragement. He has offered many valuable suggestions in all phases of this project. He has been more than generous with his time.

I would also like to thank my wife Claudia for her encouragement and support.

Table of Contents

Chapter 1 - Overview

1.1 The Maximum Clique Problem

A *graph* G = (V, E) is an ordered pair consisting of a finite set of *vertices* and a set of unordered pairs (u, v) of distinct vertices, called *edges*. The *cardinality* of the graph |V| is equal to the number of vertices. Given an undirected graph G = (V, E), a *clique* is a subset of vertices $S \subseteq V$ such that $(u, v) \in E$ for all $u, v \in S$. A clique is a *maximal clique* if it is not the subset of a larger clique. A clique that has the largest cardinality among all cliques of G is called a *maximum clique*. In general, a graph may have more than one maximum clique.

The problem of finding a maximum clique in a graph G is known as the *maximum clique problem*. Although this problem is known to be NP-complete [7], many algorithms which give a slightly less than optimal solution have been found that perform well in practice.

The maximum clique problem has several important applications, including computer vision, information retrieval, cluster analysis, classification theory, and signal transmission [4].

One of the earliest papers on practical clique algorithms is from 1970, by Augustson and Minker [1]. Most recent maximum clique algorithms are based on the branch and bound

algorithm CACM457, from Bron and Kerbosch in 1973 [5]. The most important of these were by Balas and Yu [3] and Babel [2]. The algorithm of Balas and Yu uses a greedy coloring together with the concept of a maximally triangulated subgraph(MTIS). Their algorithm was among the fastest until Babel's algorithm in 1990, which made use of the Dsatur coloring method due to Brelaz [4]. In 1998, Myrvold, Prsa and Walker [10] introduced a dynamic programming approach for timing and designing clique algorithms on very large graphs.

In order to find the maximum clique in a graph G, we present a backtracking algorithm in Section 1.2 to exhaustively search through all possible solutions in the state space tree. To reduce the number of nodes which we must process in the state space tree it is often useful to apply a *bounding function*, which we present in Section 1.3. One of the bounding functions is based on a coloring of G. The related coloring problem is discussed in Section 1.4.

1.2 Basic Definitions

Given a graph $G = (V, E)$, two vertices u and v are *adjacent* or *neighbours* if $(u,v) \in E$. The *neighbourhood* of a vertex v is $N(v) = \{ u \in V \mid (u, v) \in E\}$. The *degree* of a vertex v is $\deg(v) = |N(v)|$. For $X \subseteq V$, we define $\deg_X(v) = |N(v) \cap X|$. If $e = (u, v) \in E$, then vertex v is *incident* to edge e. Two edges are *adjacent* if they share a common vertex. For $W \subseteq V$, the *induced subgraph* G[W] has vertex set W and edge set $E' = \{(u,v) \mid (u,v) \in E, u \in W, v \in W\}$.

A *vertex coloring* of a graph G is an assignment of colors to the vertices of G such that no two adjacent vertices are assigned the same color. More formally, given a graph $G = (V, E)$, a *coloring* of G is a function *color*: $V \rightarrow M \subseteq \{0, 1, 2,\ldots\}$ such that $\text{color}(u) \neq \text{color}(v)$ for all $(u, v) \in E$. If color is onto and $M = \{0, 1, 2, \ldots, k\text{-}1\}$ the color is called a *k-coloring* of G and denoted by color(G). A *partial vertex coloring* of G is a coloring of a subgraph of G. Given a partial vertex coloring of G, let cdeg(v) denote the number of colored neighbours of any vertex v, and the *saturation degree*, satdeg(v), the number of different colors adjacent to v.

The general algorithm we use for finding maximal cliques is a type of *backtracking algorithm*. A backtracking algorithm is a recursive method for generating *feasible* solutions to a combinatorial optimization problem one step at a time. A solution is called *feasible* if it satisfies the constraints of the problem, for example, that it forms a clique. A

backtracking algorithm is *exhaustive*, in that all possible solutions are considered, so that eventually the optimal solution will be found.

Recursive calls to the backtracking algorithm produce a tree of possible solutions which is called the *state space tree*. Each node of the tree is a partial solution to the optimization problem. A leaf of the state space tree is a full (but not necessarily optimal) solution. A node at the k^{th} level of the tree is a k-tuple $X = [x_0, x_1, ...,x_k]$. In the case of the maximum clique algorithm, at the k^{th} level of the tree, the current partial solution consists of a list $X = [x_0, x_1, ...,x_k]$ where each x_i is a vertex.

If we have a partial solution $X = [x_0, x_1, ..., x_{\ell-1}]$, the requirement that X be a clique will restrict the possible values for x_ℓ to a subset $C_\ell \subseteq V$ which we refer to as a *choice set*. The computation of the choice set C_ℓ is referred to as *pruning*, since if a vertex y is in $V \backslash C_\ell$, then nodes in the subtree with root node $[x_0, x_1, ..., x_{\ell-1}, y]$ will not be considered by the backtracking algorithm. We say that this subtree has been *pruned* from the original state space tree.

Algorithm 1, from Kreher and Stinson [8, page 139] is an implementation of the maximum clique algorithm, using bounding functions. C_ℓ is the choice set at a particular step in the program. Each time the function is called, we first check whether the current clique (partial solution) $X = [x_0, x_1,..., x_{\ell-1}]$ is larger than OptSize. If so, OptSize is replaced by ℓ and OptClique is replaced by $[x_0, x_1,..., x_{\ell-1}]$.

Next, the choice set C_ℓ is computed recursively as follows. Let $A_v = \{u \in V \mid (u, v) \in E\}$ be the set of vertices which are adjacent to vertex v. Let $B_v = \{u \in V \mid u > v\}$ be the set of vertices which are higher than v in our ordering of the vertices. If we are at level 0, the choice set $C_\ell = V$, the set of all vertices in our graph G. Otherwise we define $C_\ell = C_{\ell-1} \cap A_{x_{\ell-1}} \cap B_{x_{\ell-1}}$. The first term $C_{\ell-1}$ requires that any vertex in the choice set must be adjacent to all of the vertices in the current clique $X = [x_0, x_1, \ldots, x_{\ell-1}]$. The second term $A_{x_{\ell-1}}$ requires that any vertex in the choice set C_ℓ be adjacent to the current vertex $x_{\ell-1}$. The third term $B_{x_{\ell-1}}$ guarantees that we only consider vertices which come after x_ℓ in our vertex ordering. This last condition prevents us from reconsidering previously found cliques. For example, if the node [0 3] at level two leads to a clique [0 3 6], then the vertex 3 will not be part of the choice set for the node [0 6].

Next, the bounding function B is applied to the current clique, and the result is stored in variable M. For each vertex x in the choice set C_ℓ we first check whether the bound M is less than or equal to OptSize. If so, we exit the function, since the current branch of the state space tree cannot possibly lead to a maximum clique. Otherwise, we set x_ℓ equal to x, and recursively call the function Maxclique2(ℓ-1). In the main program at the bottom, we initialize OptSize to 0, and then call the function MaxClique2(0), which indicates that we start at level 0.

Algorithm 1: Maximum Clique Algorithm (with bounding function): MaxClique2(ℓ)

global C_ℓ ($\ell = 0, 1, \ldots, n-1$)

if $\ell >$ OptSize **then**

{

OptSize $\leftarrow \ell$

OptClique $\leftarrow [x_0, x_1, \ldots, x_{\ell-1}]$

}

if $\ell = 0$ **then** $C_\ell \leftarrow V$

else $C_\ell = C_{\ell-1} \cap Ax_{\ell-1} \cap Bx_{\ell-1}$

$M \leftarrow B([x_0, x_1, \ldots, x_{\ell-1}])$

for each x in C_ℓ **do**

{

if $M \leq$ OptSize **then return**

$x_\ell \leftarrow x$

MaxClique2(ℓ-1)

}

main

OptSize $\leftarrow 0$

MaxClique2(0)

Output (OptClique)

1.3 Bounding Functions

A *bounding function* is a function that allows us to prune the size of the state space tree by not processing branches that have no chance of improving the optimal solution found so far. For any partial solution (i.e. clique) $X = [x_0, x_1, ..., x_{\ell-1}]$ let MC(X) be the size of a maximum clique of any descendent of X in the state space tree. In general, a bounding function is a real-valued function B defined on the set of nodes in the state space tree such that for any partial solution X, $B(X) \geq MC(X)$. Thus, B(X) gives an upper bound on the size of any feasible solution that is a descendent of X in the state space tree.

We can use a bounding function as follows. Suppose at some stage of the backtracking algorithm, we have a current partial solution $X = [x_0, x_1, ..., x_{\ell-1}]$, and an *OptSize* which is the size of the maximum clique found so far. If it is the case that $MC(X) \leq B(X) \leq$ OptSize, then no descendents of X in the state space tree can improve the current maximum clique. This allows us to prune the entire subtree below X.

It helps if our bounding function B(X) is both easy to compute, and close to the actual value of MC(X). These two properties work against each other. (For example, MC(X) is itself a bounding function, but difficult to calculate.)

Let $X = [x_0, x_1, ..., x_{\ell-1}]$ be a partial solution (clique) at some point in the maximum clique algorithm. Suppose that $X' = [x_0, x_1, ..., x_j]$ is a clique which extends the partial solution X, where $j \geq \ell - 1$. Then $\{x_1, ..., x_j\}$ must be a clique in the induced subgraph $G[C_\ell]$. Thus

we can obtain a bounding function by finding an upper bound on the size of a maximum clique in $G[C_\ell]$.

With respect to the maximum clique problem, the simplest bounding function to use is the *size bound*. Let $MC(\ell)$ denote the size of a maximum clique in $G[C_\ell]$. We obtain a size bound by observing that $MC(\ell) \leq |C_\ell|$. In other words, if the choice set C_ℓ contains r vertices, then the maximum clique in its induced subgraph cannot contain more than r vertices. This is a rough bounding function, but it has the advantage that it is easy to compute.

Other bounding functions can be derived from vertex colorings of the graph. Every vertex in a clique is adjacent to every other vertex of that clique. Hence, in a colored graph, all vertices of a clique must have different colors. Therefore, if a graph G has a vertex k-coloring, then the maximum clique in G has size at most k. Thus, vertex coloring gives rise to a useful bounding function for the maximum clique problem.

To calculate the *sampling bound*, we first obtain an approximate vertex coloring of the entire graph, which uses, say, k colors. Let C_ℓ be the choice set, at any particular point in the backtracking algorithm for finding the maximum clique. If the approximate vertex k-coloring is restricted to the *induced subgraph* $G[C_\ell]$, it gives a coloring of $G[C_\ell]$ (using possibly less than k colors), which is an upper bound on the size of a maximum clique in $G[C_\ell]$. In particular, the upper bound calculated is $(\ell + |\{color[x] : x \in C_\ell\}|)$

Another way to use vertex coloring as a bounding function is to apply the coloring algorithm to the induced subgraph $G[C_\ell]$ each time a new choice set is used. We call this a *greedy bound*. The upper bound computed this way is $(\ell + |color(G[C_\ell])|)$, which is usually better than the sampling bound (i.e. less than or equal to the sampling bound).

Note the difference between the sampling bound and the greedy bound. With the sampling bound, the coloring algorithm is applied just once to the entire graph G at the beginning of the program. Bounds are obtained by restricting this coloring to the induced subgraph $G[C_\ell]$ each time a new choice set used. With the greedy bound, a fresh coloring is obtained for each new choice set by applying the coloring algorithm to the induced subgraph $G[C_\ell]$. In the next section we discuss the coloring problem in more detail.

1.4 The Coloring Problem

A well-known problem in combinatorial analysis is to color the vertices of a graph G such that no two adjacent vertices are assigned the same color. We call this the *coloring problem.* The *chromatic number* of G is defined to be the smallest k such that a vertex k-coloring exists.

The graph coloring problem is related to the problem of scheduling final exams at a university, so that no student has two exams at the same time [12]. Let each course be represented by a vertex, with an edge between a pair of vertices if a student is in the two courses represented by the vertices. Each time slot for a final exam is represented by a different color. A scheduling of the set of exams corresponds to a coloring of the associated graph. If the chromatic number of a graph is k, then at least k time slots are required to schedule the exams.

Another application of graph coloring is in the channel assignment problem [12]. In North America, television channels 2 through 13 are assigned to stations such that no pair of stations within 150 miles of each other can be assigned the same channel. We can associate this problem to graph coloring as follows. Assign a vertex to each station. Two vertices are adjacent if the stations they represent are within 150 miles of each other. Each channel is represented by a different color. An assignment of channels corresponds to a coloring of the graph.

Yet another application of graph coloring is in the area of compiler design [12]. To speed up execution time for loops, it is helpful to store frequently used variables in index registers, rather than in regular memory. Graph coloring can be applied to compute the number of index registers required for a given loop. Represent each variable in the loop with a vertex. Connect two vertices with an edge if the variables they represent must be stored in index registers at the same time during execution of the loop. Thus, if two vertices are adjacent in the graph, the variables they represent must be assigned different registers. The chromatic number of the graph gives the number of index registers needed.

The coloring problem is relevant to the maximum clique problem. As explained in Section 1.3, vertex coloring can be used as a bounding function for the maximum clique problem. One problem with this approach is that it is NP-hard to find an optimal coloring of a graph [7]. However, it is easier to find colorings which may not be optimal. Yet, non-optimal colorings are still useful as bounding functions. Two possible non-optimal coloring methods are the greedy coloring algorithm and the Dsatur algorithm, each of which will be discussed in Chapter 2.

1.5 Summary

In this project we look at some results in finding maximum cliques in various graphs. In Chapter 2, we present the greedy and Dsatur coloring algorithms, and use them as bounding functions with the maximum clique backtracking algorithm.

In Chapter 3, we compare the size of the state space trees between greedy and Dsatur coloring algorithms. First we look at the results on some random graphs, then we generate some random k-partite results, and compare results on these graphs.

In Chapter 4, we look more closely at the coloring problem, and compare the number or colors required by the greedy versus the Dsatur algorithm on some random graphs, and random k-partite graphs.

In Chapter 5, we present a bichromatic interchange scheme within the coloring algorithms, in an attempt to reduce the total number of colors required to color a graph. We investigate the performance of the coloring algorithms with the bichromatic interchange.

In Chapter 6 we give a brief conclusion to this report.

Chapter 2 - Coloring Algorithms

2.1 Greedy Coloring Algorithm

Although it is NP-hard [7] to find a vertex k-coloring in which k is minimized, it is often possible to find an approximate coloring in a reasonable amount of time. One way to do this is to color the vertices sequentially using a greedy coloring algorithm. This is not to be confused with the greedy bound, which a type of bounding function, as discussed in the previous chapter. With a greedy strategy, the vertices are colored in order, each vertex receiving the lowest available color. The greedy coloring algorithm has the advantage that it does not require a large amount of calculation, yet still yields graph colorings that are reasonably close to being optimal.

Algorithm 2, from Kreher and Stinson [8, page 137] is an implementation of the greedy coloring algorithm. V is the set of n vertices in the graph, in some predefined order 0, 1, 2, ..., n-1. A is the adjacency matrix, and color is an array, with color[i] equal to the color of the i^{th} vertex. ColorClass is a list of sets, with ColorClass[h] containing the set of vertices colored using color h. This Greedy Coloring algorithm returns the number of colors used to color the graph, which is represented by the variable k. Initially, k is zero. For each vertex i in the graph, we attempt to color i with the smallest possible color h such that no vertex adjacent to i is already colored using color h. If a new color is required, then we update the number of colors used k. When the entire graph has been colored, we return k, the total number of colors used.

Algorithm 2 [8]: GreedyColoring Algorithm: GreedyColor(G = (V, E))

global color

comments: V = {0, 1,..., n - 1}

A_i = set of vertices adjacent to vertex i

color[i] holds the color of the i^{th} vertex

ColorClass[h] contains the set of vertices colored using color h

```
k ← 0
for i ← 0 to n - 1 do
{
    h ← 0
    while h < k and (A_i ∩ ColorClass[h]) ≠ ∅ do h ← h + 1
    if h = k then
    {
        k ← k + 1
        ColorClass[h] ← ∅
    }
    ColorClass[h] ← ColorClass[h] ∪ {i}
    color[i] ← h
}
return (k)
```

2.2 Dsatur Coloring Algorithm

Another sequential coloring algorithm, due to Brelaz [4], is called the Dsatur algorithm, because it uses saturation degrees. Recall that given a simple graph G which has been partially colored, the saturation degree of a vertex w is defined to be the number of different colors that w is adjacent to. As with the greedy algorithm, the vertices are colored sequentially with the smallest possible color. The difference is that rather than using a pre-defined order, the order is determined by choosing at each step a vertex of maximal saturation degree.

Dsatur coloring algorithm

1) Arrange the vertices by decreasing order of degrees.
2) Color a vertex of maximal degree with color 1.
3) Choose a vertex of maximal saturation degree.
4) Color the chosen vertex with the least possible color.
5) If all vertices are colored, stop. Otherwise, return to step 3.

In case of a tie among several vertices at step 3, there are several possible ways of selecting the next vertex to be colored. Brelaz [4] suggests choosing any vertex of maximal degree in the uncolored subgraph. Babel [2] suggests the following. Let X be the set of all vertices of maximal saturation degree. To choose among several vertices of maximal saturation degree either

a) Choose a vertex $v \in X$ such that $\text{cdeg}(v) \geq \text{cdeg}_X(w)$ for all $w \in X$, or

b) Choose a vertex $v \in X$ such that $\deg_X(v) \geq \deg_X(w)$ for all $w \in X$.

By successively coloring the vertices of maximum saturation degree with the least possible color we begin by the coloration of a clique. This provides a lower bound on the chromatic number of the graph. In addition, because the Dsatur algorithm makes use of the graph structure, it often finds cliques of almost maximal dimension [4]. The Dsatur algorithm can be extended without the increasing order of complexity to find the connected components of the graph and at the same time compute upper and lower bounds for the clique number of each component [2].

Chapter 3 – State Space Tree Size Comparisons

In this chapter we compare the size of the state space trees in finding the maximum cliques in a graph using either the sampling bound or the greedy bound, in combination with either the Dsatur coloring or the greedy coloring algorithm. Thus, there are four possible combinations of bounding function/coloring algorithm.

3.1 Random Graphs (Greedy vs. Dsatur Algorithm)

In Table 1 and Table 2 we list the number of nodes in the state space trees for some random graphs. The bounding function used is either the sampling bound or the greedy bound, which are each used with either the Dsatur coloring algorithm or the greedy coloring algorithm. The results with the greedy coloring algorithm come from Kreher and Stinson [8]. We have added the additional rows containing the results when the Dsatur coloring algorithm is used.

In Table 1, the random graphs have an edge density of 0.5, with the number of vertices ranging from 50 to 250. When the sampling bound is used, the Dsatur algorithm requires slightly fewer nodes to find the maximum clique for all five graphs, although the difference is not large. For example, for a graph of 250 vertices the difference is less than 1%.

When a greedy bound is used, the Dsatur coloring algorithm performs significantly better than the greedy coloring algorithm, usually requiring less than half the number of nodes in the state space tree.

In Table 2, the random graphs have an edge density of 0.75, with the number of vertices ranging from 25 to 125. The results are similar to the results in Table 1. When using the sampling bound, there is little to choose between the greedy coloring algorithm and Dsatur algorithm. The Dsatur coloring algorithm requires slightly fewer nodes for smaller graphs, but for graphs with more vertices, the greedy coloring algorithm requires slightly fewer nodes. Overall, neither algorithm appears to perform significantly better than the other.

When the greedy bound is used, the Dsatur coloring algorithm again gives a much smaller state space tree than the greedy coloring algorithm, with the number of nodes from two to five times less.

Based on these results, it appears that for random graphs, when the sampling bound is used, there is no advantage in using the Dsatur coloring algorithm instead of the simpler greedy coloring algorithm. However, when the greedy bound is used, the Dsatur coloring algorithm does result in a state space tree with significantly fewer nodes than the greedy coloring algorithm.

Table 1

Size of state space trees for random graphs with edge density 0.5

Number of vertices	50	100	150	200	250
Number of edges	607	2535	5602	9925	15566
Size of maximum clique	7	9	10	11	11
Bounding Function					
Sampling Bound/Greedy Coloring	2268	44072	266246	1182514	4093535
Sampling Bound/Dsatur Coloring	2055	40529	251373	1147409	4072615
Greedy Bound/Greedy Coloring	430	5734	22599	91671	290788
Greedy Bound/Dsatur Coloring	238	2229	8542	37329	110899

Table 2

Size of state space trees for random graphs with edge density 0.75

Number of vertices	25	50	75	100	125
Number of edges	236	959	2045	3720	5780
Size of maximum clique	11	14	15	17	18
Bounding Function					
Sampling Bound/Greedy Coloring	794	37218	195567	2225982	15615755
Sampling Bound/Dsatur Coloring	497	34180	210336	2264085	17769063
Greedy Bound/Greedy Coloring	91	2843	10476	70404	413421
Greedy Bound/Dsatur Coloring	46	1083	2142	19583	105695

3.2 k-partite Graphs (Greedy vs. Dsatur Algorithm)

A graph G is called *k-partite* if its vertex set V can by partitioned into k disjoint nonempty sets $A_1, A_2, \ldots, A_k$, called *components*, such that every edge in the graph connects a vertex in A_i with a vertex in A_j for $i \neq j$, (and no edge in the graph connects two vertices in the same component A_i.) k-partite graphs are of interest in combinatorial analysis because they arise naturally from block design searches.

We generated random k-partite graphs of edge density *p* as follows. For simplicity, we limited our graphs to the special case where k divides n, where n is the total number of vertices in the graph. For each possible edge between a pair of vertices in two different components of the graph, we generate a random number *r* in the interval (0, 1). If r is between 0 and p, then we add an edge to the graph (by placing a 1 in the corresponding position of the adjacency matrix of the graph). Otherwise, if r is between p and 1, then no edge connects the two vertices. This scheme allows us to generate k-partite graphs, where k divides n, with a particular edge density p, with $0 \leq p \leq 1$.

In this section we apply the maximum clique algorithm to randomly generated k-partite graphs. In the best case, we would find a clique of size k, but of course a k-clique does not always exist. Note that the chromatic number for the graph is at most k.

Although the graphs were randomly generated, the vertex ordering was such that all vertices in the same partition were together. Such an ordering guarantees that the greedy

algorithm will not use more than k colors. For example, vertices in the first partition can each be colored using color 1, since no two vertices in the same partition are adjacent to each other. In the worst case, each vertex in the second partition can now be colored using color 2, but in general we will be able to use a mixture of color 1 and color 2 in this partition. Similarly, in the third partition we can always color each vertex using color 3 in the worst case, but in general we will be able to color some of these vertices with colors 1 or 2. Continuing in this way it is clear that for a vertex ordering in which all vertices in the same partition are together, the greedy algorithm will never require more than k colors on a k-partite graph. In fact, for graphs of low density, the greedy coloring algorithm will frequently require less than k colors when this vertex ordering is used.

In Table 3 we apply the maximum clique algorithm to some randomly generated k-partite graphs, comparing the use of the greedy and Dsatur algorithms as bounding functions. In each graph, we have 100 vertices, with the edge density ranging from 0.5 to 0.75, and the number of partitions ranging from 5 to 20.

Table 3

Size of state space trees for random k-partite graphs

Number of vertices (n)	100	100	100	100	100
Number of partitions (k)	5	20	10	20	20
Edge Density(p)	0.6	0.5	0.6	0.6	0.75
Size of maximum clique	5	9	10	11	15
Bounding Function/Coloring Algorithm					
Sampling Bound/Greedy Coloring	6	25381	2139	91178	238467
Sampling Bound/Dsatur Coloring	6	29866	27318	130546	1198148
Greedy Bound/Greedy Coloring	6	3099	972	8186	41361
Greedy Bound/Dsatur Coloring	6	1459	1730	4215	21898

When the sampling bound is used, the greedy coloring algorithm performs better than the Dsatur coloring algorithm. For the graph with 100 vertices, 10 partitions, and edge density 0.6, this difference is more than a factor of 10. The fact that the greedy coloring algorithm requires significantly fewer nodes in all (but one) cases suggests that it requires fewer colors to color the graph. This translates into a better bounding function for the clique algorithm. We will see in the next chapter that this is generally true for k-partite graphs with "high" edge densities.

When the greedy bound is used, the Dsatur algorithm performs significantly better, usually requiring only about half as many nodes to be searched. Recall that with the greedy bound, the bounding function is called once during each step of the clique algorithm. The superior performance of the Dsatur algorithm suggests that it requires fewer colors when coloring subgraphs of the full graph, when the number of vertices is smaller.

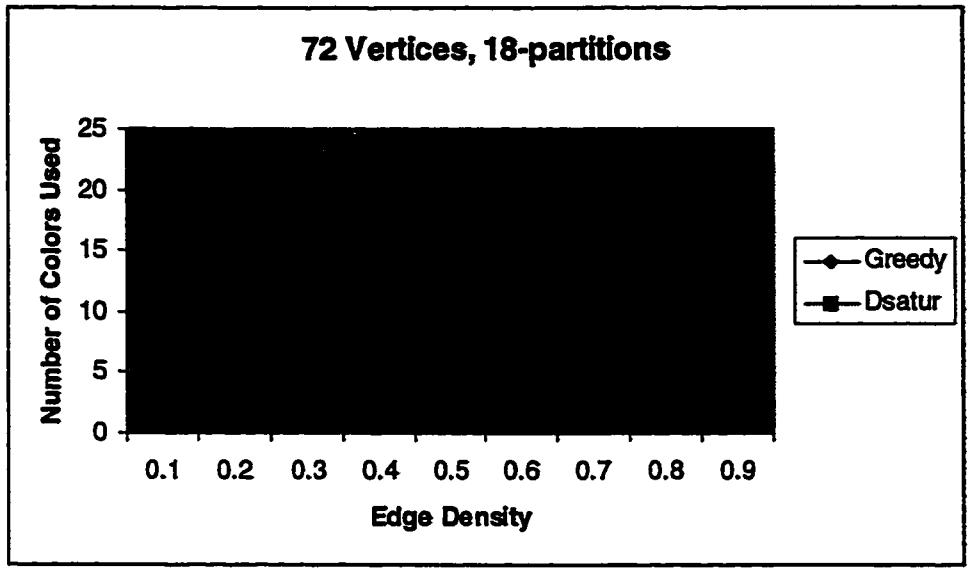
Table 3 gives a brief comparison of the greedy coloring algorithm and the Dsatur coloring algorithm, when they are used in conjunction with either the sampling bound or the greedy bound, on some k-partite graphs. It is in no way comprehensive in considering all possible combinations of the number of vertices, the number of partitions, and the graph density. The table compares the size of state space trees, which of course are influenced by the number of colors used by the respective coloring algorithms. It is not immediately clear from the results under which combinations of n, k, and p would one coloring algorithm perform better than the other. This is what we will investigate in more detail in the next chapter.

Chapter 4 – Coloring Random k-partite Graphs

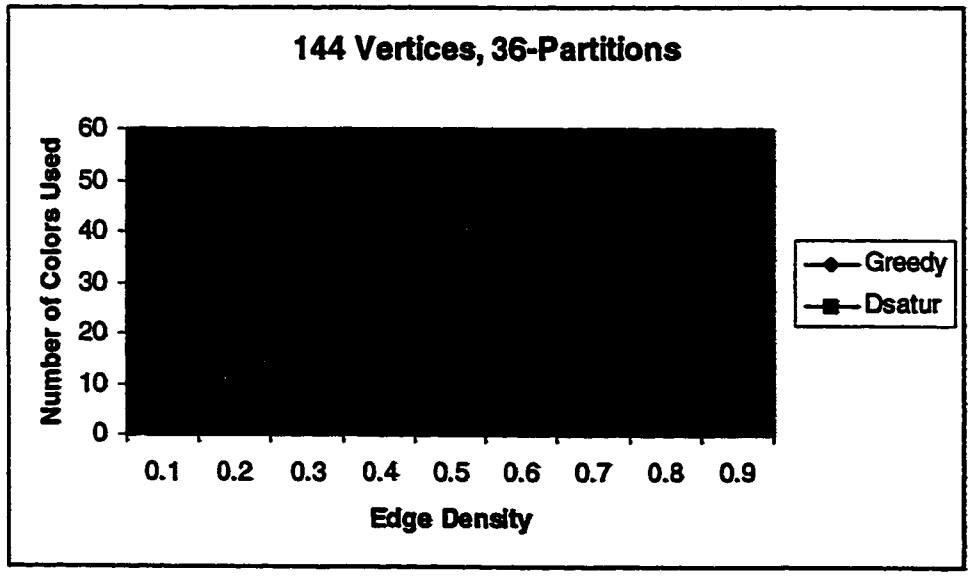
4.1 Greedy vs. Dsatur Coloring Algorithm

In this section we compare the greedy and Dsatur algorithms with respect to the number of colors required for some randomly generated k-partite graphs. In the first set of experiments, we keep the ratio n / k (number of vertices / number of partitions) fixed, while varying the edge density p. As mentioned in the previous chapter, a vertex ordering in which vertices in the same partition are consecutive guarantees that the greedy algorithm will never require more than k colors for such graphs.

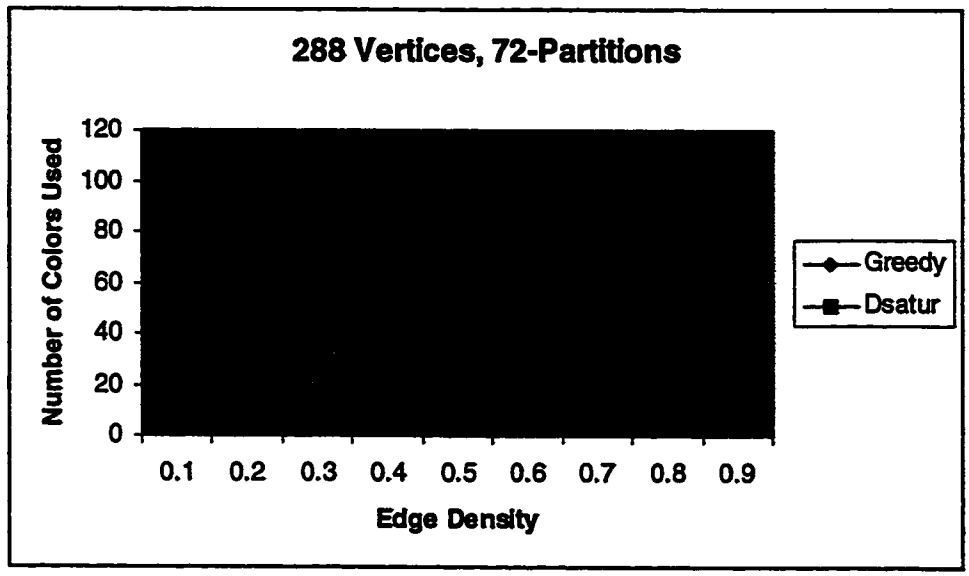
In Figure 4.1, we have n = 72, k = 18, while the edge density p varies from 0.1 to 0.9. When p is small, the Dsatur algorithm used fewer colors, while for larger values of p, the greedy algorithm is superior. The crossover point appears to be about 0.6. Note that the greedy coloring algorithm "flattens out" when it reaches 18 colors, whereas the Dsatur algorithm uses more than 18 colors for an edge density above 0.6. Of course this is less than optimal, since any k-partite graph can be colored using at most k colors by coloring all of the vertices in each of the k partitions A_i with the same color i.



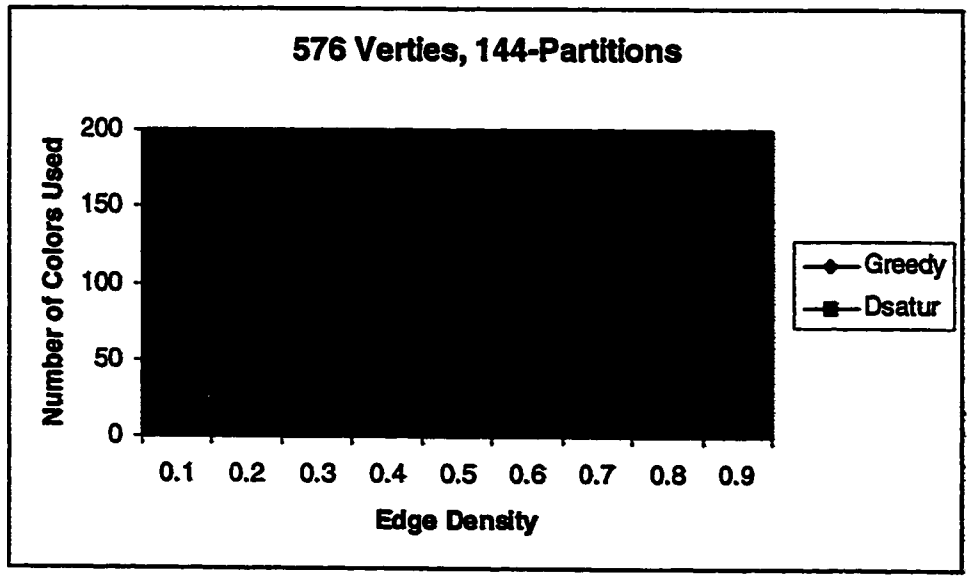
In Figure 4.2 we consider graphs with 144 vertices and 36 partitions, again varying p. As before, the Dsatur algorithm requires fewer colors for graphs with low edge density, but above a certain point the greedy algorithm performs better. In this case, the crossover point occurs near p = 0.7.



In Figure 4.3 we fix n = 288, k = 72 and vary p. The Dsatur coloring algorithm performs better for graphs of low density. In this case the crossover point is near p = 0.75.



In Figure 4.4, with n = 576, k = 144, the crossover point occurs at p = 0.8.

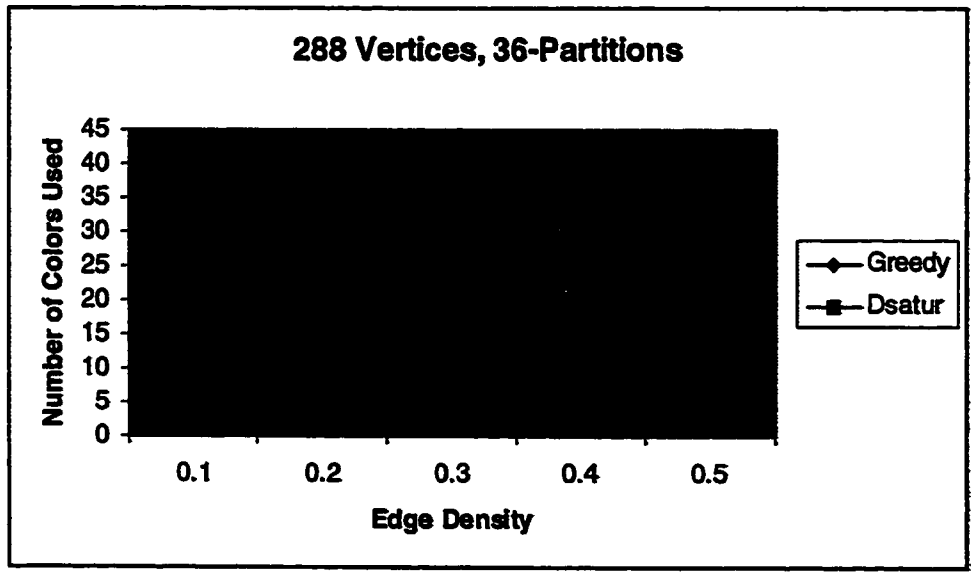


Each of the above graphs demonstrates that for random k-partite graphs with a fixed ratio n / k, (so the number of vertices in each partition is n / k), the Dsatur algorithm requires

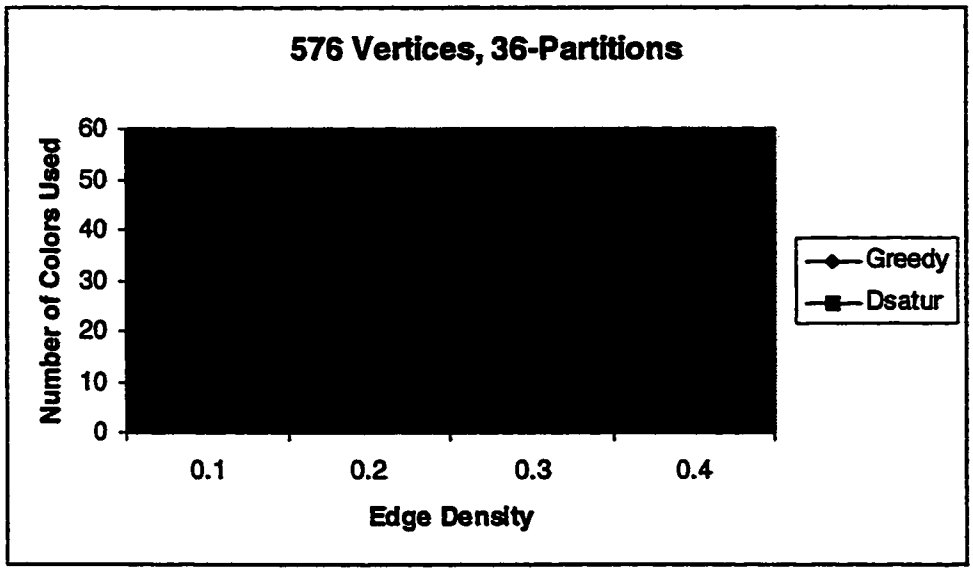
fewer colors for graphs of low density, while beyond a certain point the greedy algorithm performs better. Overall, the four graphs suggest that as the size of the graph increases, so does the crossover point between the Dsatur algorithm and greedy algorithm.

In the next set of experiments, we fix the number of partitions k, but vary the number of vertices in the graph, starting from n = 288. For each value of n, we vary the edge density p, starting from p = 0.1, until we find a crossover point between the greedy and Dsatur algorithms.

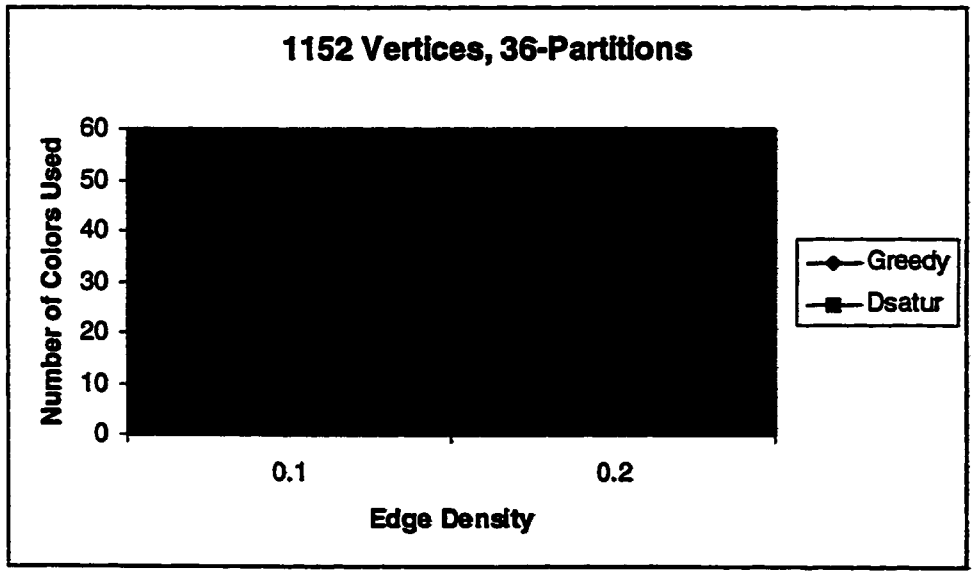
In the first example, we have 288 vertices. As before, the Dsatur algorithm performs better on graphs of low density. Beyond an edge density of about 0.45 the greedy algorithm is superior.



In the next example, with 576 vertices, the Dsatur algorithm performs better for graphs of edge density below 0.25. Beyond that value, the greedy algorithm is superior.



With a graph of 1152 vertices, the greedy algorithm is superior for graphs with an edge density larger than about 0.13.



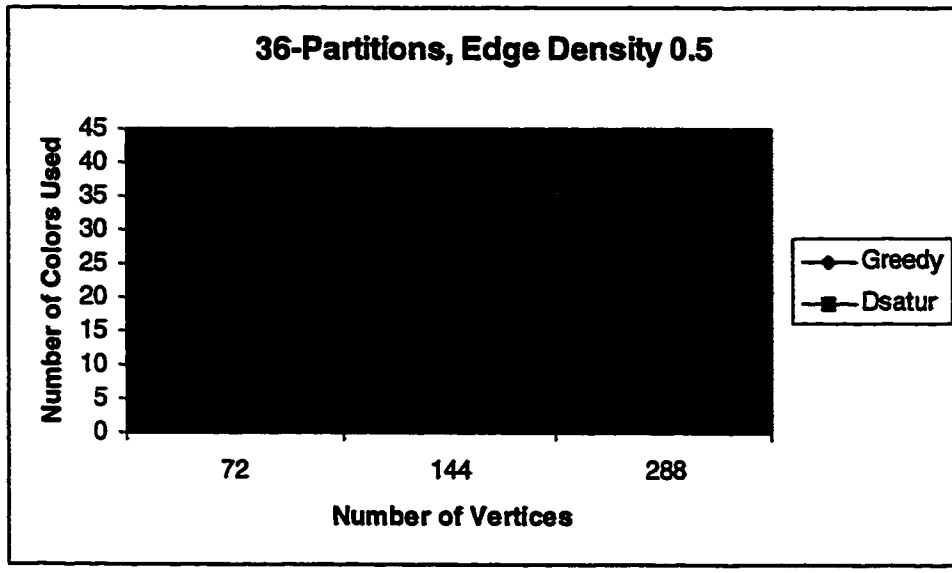
Each of the above graphs confirms that for random k-partite graphs, the Dsatur algorithm needs fewer colors for graphs of low density. Taken together, the graphs show that for a fixed number of partitions k, the crossover point between greedy and Dsatur algorithms decreases as n increases. In other words, as n increases, the range of edge densities for

which the Dsatur algorithm outperforms the greedy algorithm becomes ever more narrow.

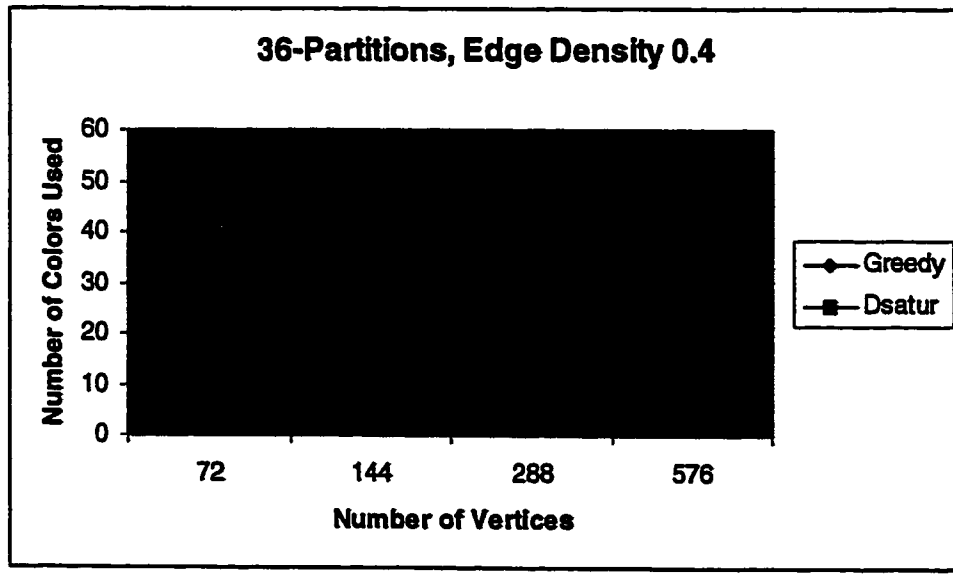
Also, as the number of vertices increases, the range of edge densities for which the greedy algorithm is able to improve upon its worst case situation of using k colors decreases. In other words, since we know a priori that a 36-partite graph can be colored using at most 36 colors, unless there is a chance that a particular coloring algorithm can improve on this number, it makes no sense to use up computing time by calling the coloring algorithm.

In our third set of experiments, we fix the number of partitions k (at 36) and the edge density p, and compare the performance of the greedy and Dsatur coloring algorithms while varying the number of vertices n. We repeat the experiment for values of p ranging from 0.1 to 0.5.

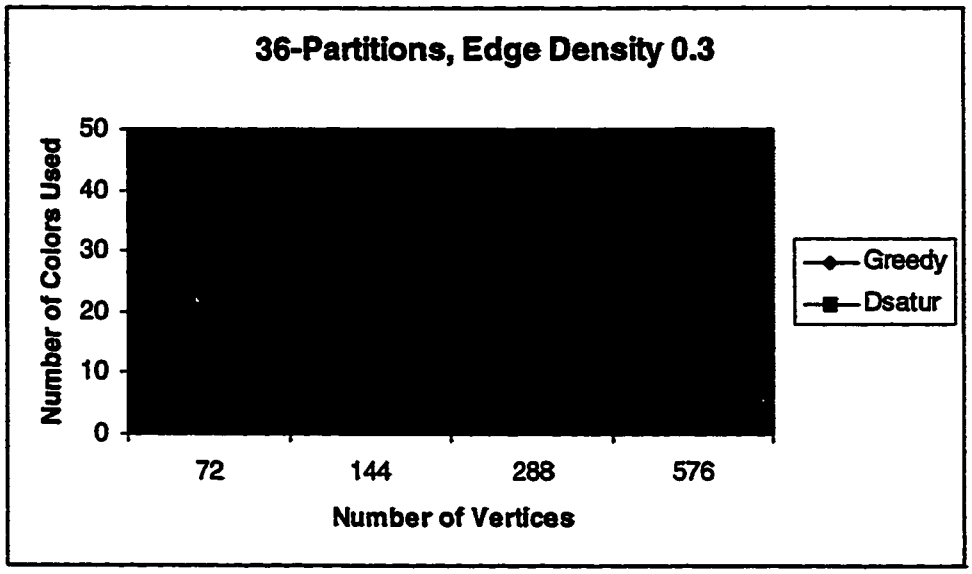
With an edge density of 0.5, the Dsatur algorithm performs better for small graphs, with a crossover point of about 150.



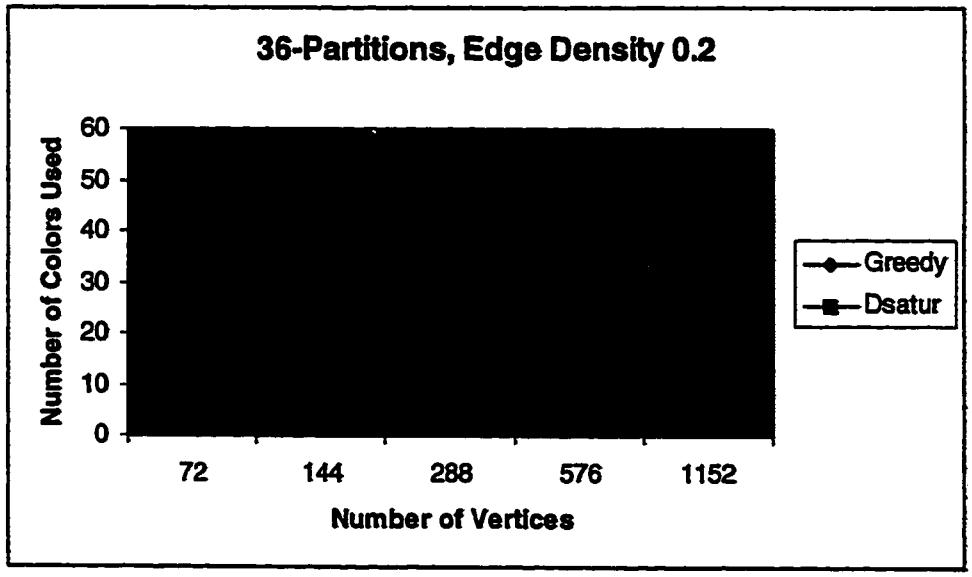
With an edge density of 0.4, the crossover point is about 300.



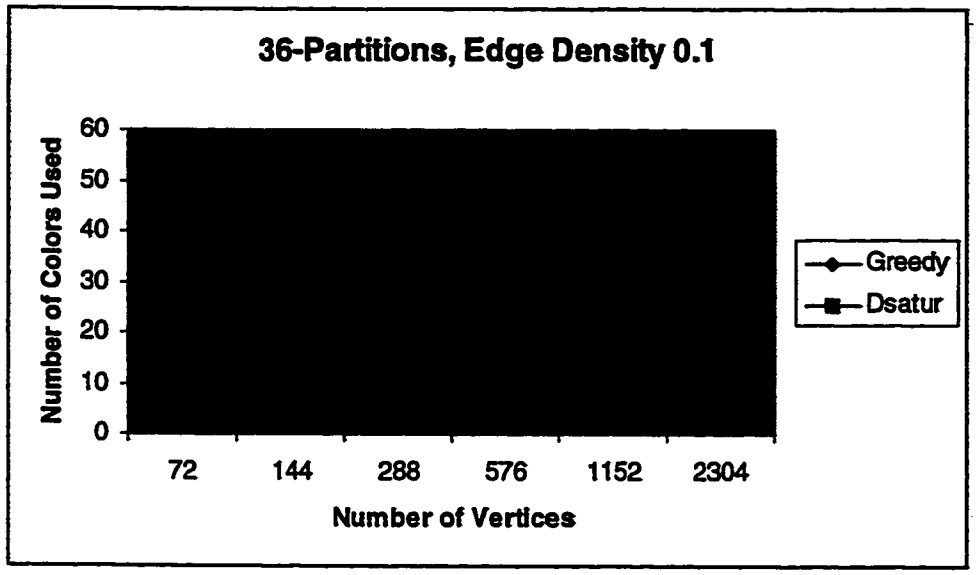
With an edge density of 0.3, the crossover point is about 400.



With an edge density of 0.2, the crossover point is about 700.



With an edge density of 0.1, the crossover point is about 1400.



Each of the above five graphs suggest that for fixed values of k and p, the Dsatur coloring algorithm performs better for smaller graphs, but as n increases, the greedy algorithm requires fewer colors beyond a certain point.

Taken together, the results show that as the graph density p increases, the range of values for which the Dsatur algorithm outperforms the greedy algorithm narrows. For example, when $p = 0.5$ the Dsatur algorithm is superior only for graphs of less than 150 vertices, while for graphs of edge density 0.1, the Dsatur algorithm needs fewer colors for graphs of up to 1400 vertices.

4.2 Application to the Maximum Clique Problem

As mentioned in the previous section, as the number of vertices n increases, the range of edge densities for which the greedy coloring algorithm is able to improve upon its worst case situation of using k colors decreases. We can use this last fact as follows. Suppose we know the crossover point N, where N is the largest number of vertices for which the greedy coloring algorithm requires fewer than k colors on a particular k-partite graph. Suppose that we are using the *greedy bound* as our bounding function. In that case, if we are at a point in the maximum clique algorithm for which the choice set C[L] has N or more vertices, then there is nothing to be gained by calling the greedy coloring algorithm, since above the crossover point the greedy algorithm usually requires the worst case of k colors. Thus if we only call the coloring algorithm if the condition C[L] < N is satisfied, we can reduce the time required to find a maximum clique.

In Table 4 we see that this is indeed the case. We are using the greedy bound, with greedy coloring algorithm, but not calling the coloring algorithm if the number of vertices in the choice set C[L] is larger than the crossover value N. In the first column, with 720 nodes, we have a time reduction from 215 seconds to 188 seconds. In the last column, there is still a modest time saving, from 12 seconds to 11 seconds.

Note that the number of nodes in the search tree is not affected by this optimization procedure. That is because we are not calling the greedy bound function only in cases in which we know that the greedy coloring algorithm is unlikely to improve on a worst case

scenario of using k colors on a k-partite graph. Calling the greedy bound function in such cases would not eliminate any nodes from the state space tree, so the number of nodes in our state space tree is not reduced.

Table 4 **Optimization with Greedy Bound**

Number of vertices (n)	720	576	540	360
Number of partitions (k)	18	18	18	18
Edge Density	0.3	0.2	0.3	0.3
Maximum Clique Size	9	6	8	7
Nodes	420971	37908	120390	43450
Crossover Point(N)	180	250	180	180
Execution Time(seconds)	215	17	51	12
Time with Optimization(seconds)	188	15	50	11

Chapter 5 - Bichromatic Interchange Algorithm

5.1 Description

We attempted to improve the performance of the coloring algorithms by implementing the *bichromatic interchange* algorithm as described in a paper by Matula [9]. This method seeks to make an interchange of colors in a bipartite subgraph each time a new color has to be introduced. If this interchange is successful, the new vertex can be colored using an existing color rather than a new color. This interchange method can be used for any algorithm with coloration of the vertices by the smallest possible color in a given order. It is hoped that by introducing the bichromatic interchange scheme into a sequential coloring scheme such as the greedy algorithm or Dsatur coloring algorithm, the total number of colors used on the graph will be reduced. Of course this will be at a cost of increased computing time.

Here is a more detailed description of the bichromatic interchange. Suppose in a graph G that the subgraph $G[v_0, v_1, \ldots, v_{i-2}]$ has already been colored using (k-1) colors. A k^{th} color is needed for the next vertex only if vertex v_{i-1} is adjacent to vertices with colors 0,1, 2, ..., k-2.

If the neighbours of v_i (among the set $\{v_0, v_1, \ldots, v_{i-2}\}$) contains a complete subgraph with k-1 vertices, then certainly the chromatic number $\chi(G[v_0, v_1, \ldots, v_{i-1}]) = k$, and a new color is undoubtedly necessary. If not, it may be possible, before coloring v_i, to change

the colors on some of the neighbours of v_i in a way that the coloring of $G[v_0, v_2, \ldots, v_{i-1}]$ still uses (k-1) colors, and leaves at most (k-2) colors adjacent to v_i. This would free a previous color for v_i, avoiding the necessity of introducing a new color.

With the bichromatic interchange, each time in a sequential coloring algorithm that a new vertex has to be colored, we first check to see if such a color interchange is possible on the previously colored vertices. It is hoped that performing a sequential coloring with interchange, the total number of colors required for the entire graph is reduced.

The steps of a sequential graph coloring with bichromatic color interchange are as follows:

Let $v_0, v_1, \ldots, v_{n-1}$ be an ordering of the vertices of G. (This ordering may be in descending order of degrees, or it may be random.)

1) Assign color 0 to vertex v_0

2) If $G[v_0, v_1, \ldots, v_{i-2}]$ has been j-colored (colored with exactly j colors) using each of the colors 0, 1, 2, ..., j-1, and if m is the minimum positive integer not occurring on vertices $v_0, v_1, \ldots, v_{i-2}$ adjacent to v_i in G, then

a) If $m < j$ then an interchange is not necessary. We assign v_{i-1} color m, while each of $v_0, v_1, \ldots, v_{i-2}$ retain their original colors. This gives a j-coloring of the subgraph $G[v_0, v_1, \ldots, v_{i-1}]$.

b) If $m = j$, then, before assigning a new color to v_i, we check to see if a bichromatic interchange is possible.

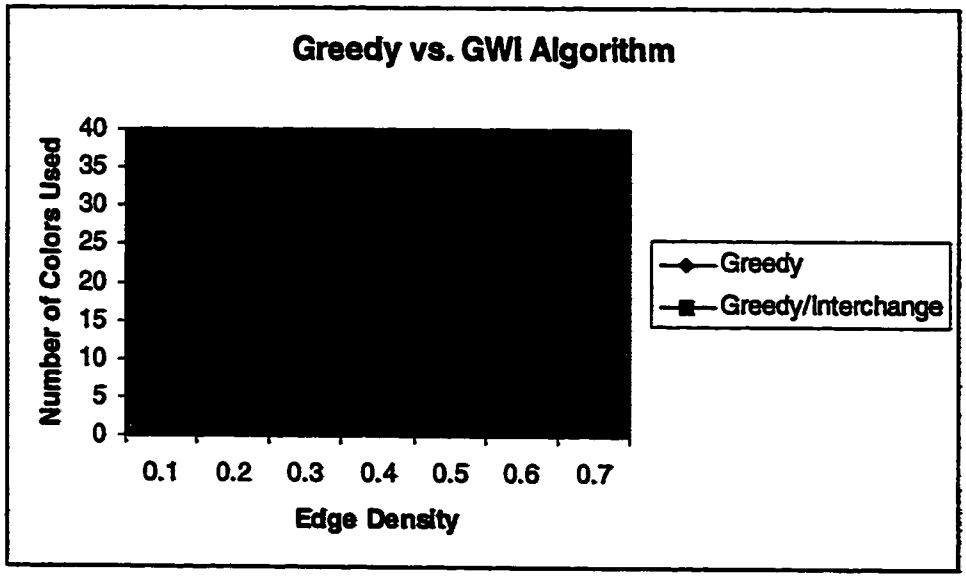
First, we find the set $K \subseteq \{0,1, 2, \ldots, j-1\}$ which is the set of colors such that $\alpha \in K \Rightarrow$ exactly one vertex adjacent to v_i in $G[v_0, v_1, \ldots, v_{i-1}]$ has color α. Suppose that we can find two colors $\alpha, \beta \in K$, $\alpha \neq \beta$. Let A_α and A_β be the sets of vertices which have been colored with α and β respectively. We define an α, β component of $G[v_0, v_1, \ldots, v_{i-1}]$ to be a connected component of the subgraph $G[A_\alpha \cup A_\beta]$.

If there exists an α, β component of $G[v_0, v_1, \ldots, v_{i-1}]$ which has only one vertex w adjacent to v_i , then on that component we interchange the colors α and β. If this changes the color of vertex w from α to β, then we can now assign color α to v_i. If the interchange causes vertex w to have its color switched from β to α, we can now assign color β to v_i. In either case, a j-coloring of $G[v_0, v_1, \ldots, v_{i-1}]$ is obtained.

If no such bichromatic interchange is possible for any α, β component of $G[v_0, v_1, \ldots, v_{i-2}]$ then we must assign color j to v_i , obtaining a (j+1)-coloring of $G[v_0, v_1, \ldots, v_{i-1}]$.

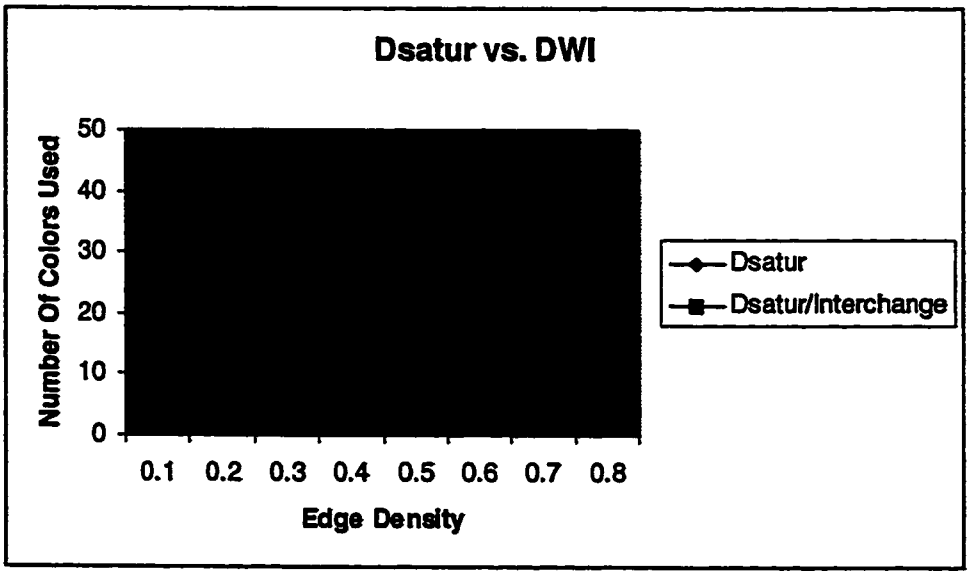
5.2 Comparison with Previous Results

In this section we compare the number of colors used by the greedy coloring algorithm and Dsatur coloring algorithm with and without the bichromatic interchange, when used by a sampling bound. When coloring random k-partite graphs, it appears that the interchange scheme generally results in a small improvement when combined with the greedy coloring algorithm. That is, with the interchange the number of colors required is usually reduced by one or two. With the Dsatur algorithm however, the results are less consistent, with the interchange scheme sometimes reducing the number of colors, and sometimes increasing the number of colors, but always by a small amount.

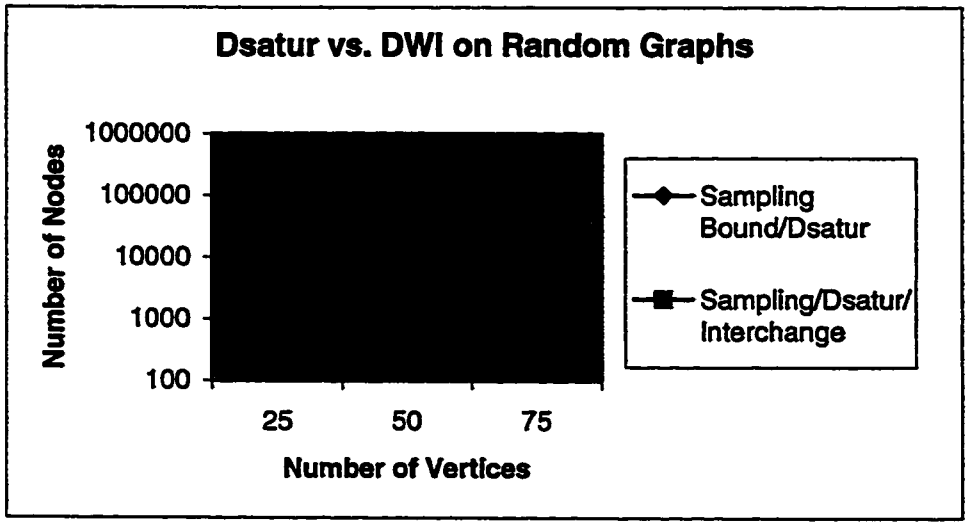


In the first graph we compare the performance of the greedy coloring algorithm with and without the interchange, for a 36-partite graph with 144 vertices. We see that applying the interchange provides a slight reduction in the number of colors used.

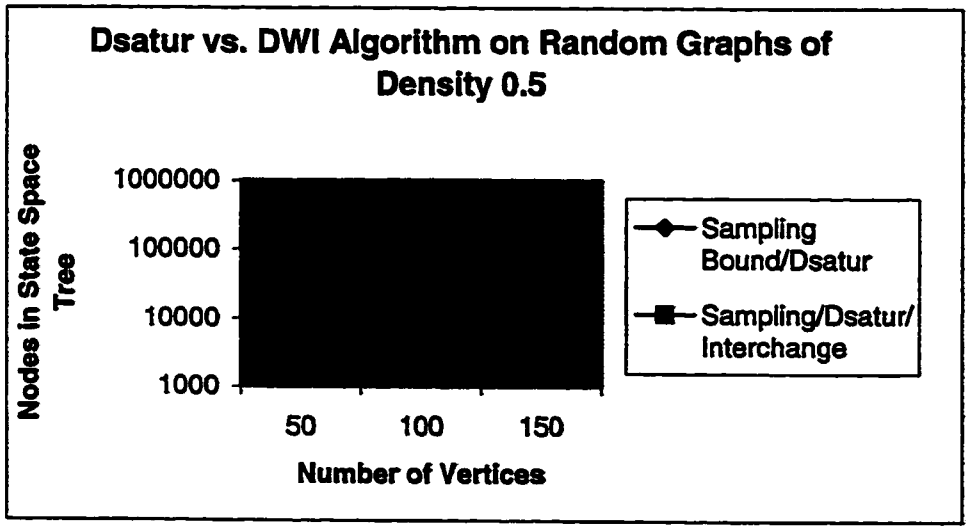
In the second graph we compare the Dsatur Algorithm, with and without the interchange, for the same 36-partite graph of 144 vertices. In this case, there does not appear to be any significant difference. If anything, with the interchange the algorithm seems to perform slightly worse.



In the next example we compare the Dsatur algorithm with and without interchange on random graphs of density 0.75. In this case there does seem to be a clear improvement when the interchange scheme is introduced. (Note that the y-axis uses a logarithmic scale in the graph.)



However, in the final example below we see that on random graphs with edge density 0.5 there does not appear to be any significant improvement when the interchange scheme is used with the Dsatur algorithm.



These results suggest that for k-partite graphs, when the sampling bound is used, the interchange does improve performance of the greedy coloring algorithm, but not the Dsatur coloring algorithm.

For random graphs, the interchange did improve the performance of the Dsatur algorithm for graphs of density 0.75. However, for a graph density of 0.5 the interchange made little difference.

Chapter 6 - Conclusion

In this project we have implemented a Dsatur coloring algorithm, and applied it towards coloring random graphs and random k-partite graphs. We have compared performance with a greedy coloring algorithm by looking at the number of colors used to color the graphs.

We have also used these coloring algorithms within a maximum clique algorithm, in combination with either a sampling bound or a greedy bound, and compared the number of nodes in the corresponding state space trees.

Finally, we implemented an attempted improvement to the coloring algorithms known as a bichromatic interchange, and compared results both with and without the interchange.

In searching for maximum cliques in random graphs, we found that when a greedy bound was used, the Dsatur coloring algorithm resulted in fewer nodes in the state space tree than did the greedy coloring algorithm. However, when a sampling bound was used, there was little if any improvement.

When we compared the two Dsatur coloring algorithm with the greedy coloring algorithm on random k-partite graphs, we found that the results are sensitive to the number of vertices, the number of partitions k, and the edge density p.

Such graphs can always be colored with at most k colors, by coloring the vertices in each partition by the same color. Hence, if the coloring algorithm requires more than k colors, it is not very useful. The greedy coloring algorithm never requires more than k colors for such graphs for a vertex ordering such that vertices in the same partition are consecutive. However, for graphs of "large" density, the Dsatur algorithm does require more than k colors.

It is always the case that for graphs of low density, the Dsatur coloring algorithm requires fewer colors than the greedy coloring algorithm. The question then becomes for which range of edge densities does the Dsatur coloring algorithm improve upon the greedy coloring algorithm. We call this value a crossover point.

For graphs with a constant number of vertices in each partition (n / k fixed), the value of the crossover point increases as the number of vertices in the graph increases. For graphs with a constant number of vertices in each partition (k fixed), the crossover point decreases as the number of vertices in the graph increases. Finally, for graphs with a fixed number of partitions, the range of graph sizes n for which the Dsatur coloring algorithm decreases as the graph density increases.

We note that knowing such crossover points can be useful in reducing the computation time (but not the number of nodes) in finding maximum cliques using a greedy bound. By not invoking the coloring algorithm in cases where we know in advance that it will not help in pruning the state space tree, we avoid some irrelevant calculations.

Finally, we implemented a bichromatic interchange scheme to be used with the coloring algorithms and a sampling bound. We found that for k-partite graphs, the interchange did result in a slight improvement for the greedy coloring algorithm, but made little difference to the Dsatur algorithm. For random graphs, the interchange did significantly improve performance of the Dsatur algorithm for graphs of high density (0.75), but made little difference for graphs of moderate density (0.5).

Further study in this area could involve applying the results to specific rather than random graphs. Other coloring algorithms could be investigated. One might also try different vertex orderings when performing the graph colorings, for example, ordering the vertices in reverse order of degrees, or perhaps randomly. Work could also be done to optimize the implementations of the existing algorithms.

References

[1] J.G. Auguston and J. Minker. An analysis of some graph theoretical cluster techniques. Journal of the ACM, 17(4):571-588, 1970.

[2] L. Babel. Finding maximum cliques in arbitrary and special graphs. Computing, 15:321-341, 1991.

[3] E. Balas and C. S. Yu. Finding a maximum clique in an arbitrary graph. SIAM J. Computing, 15(4):1054-1068, 1986.

[4] Daniel Brelaz. New methods to color the vertices of a graph. Communications of the ACM, 22:251-256, 1979.

[5] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph H. Comm. ACM, 16(9):575-577, 1973.

[6] N. Deo. Graph Theory with applications to engineering and computer science. Prentice Hall, 1974.

[7] M. Garey and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco, 1979.

[8] D.L. Kreher and D. R. Stinson. Combinatorial Algorithms, Generation, Enumeration, and Search. CRC Press, 1999.

[9] David W. Matula, George Marble, & Joel D. Isaacson. Graph Coloring Algorithms. In Graph Theory and Computing, Academic Press, New York, 1972, pp. 109-122.

[10] W. Myrvold, T. Prsa and N. Walker. A dynamic programming approach for testing clique algorithms, preprint, 1997.

[11] I. Pohl, A. Kelly. A Book on C, 4th edition. Addison Wesley, 1998.

[12] K. Rosen. Discrete Mathematics and Its Applications, 4th edition. McGraw-Hill, 1999.

Appendices

Here we provide a listing of the C language implementations of the algorithms. These are based on programs made available by Kreher and Stinson [8].

Appendix 1 defs.h

This header file lists some symbolic constants used in the program.

```
#define WORDSIZE 32
#define false 0
#define true 1
#define BETTER 1
#define EQUAL  0
#define WORSE -1
#define BLANK ' '
#define COMMA ','
#define TAB ''
#define NEWLINE '\n'
#define SEMICOLON ';'
#define RBRACE '}'
#define LBRACE '{'
#define ComputeWords(n) (WORDS=n/WORDSIZE+1)
#define decreasing -1
#define increasing  1
#define Min(x,y) ((x<y)?x:y)
#define red 'R'
#define black 'B'
```

Appendix 2 setlib0.h

This header file gives the interface for the set operations.

```
typedef unsigned int UINT;
typedef UINT * set;
typedef set* setlist;

extern int WORDS;
extern set V;
extern set _AuxSet1;

//  SetInit(n) sets the universe V to be V={0,1,2,...,n-1}.  It
//  must be executed at least once before other operations are
//  executed.
extern void SetInit(int);
extern void ClearSet(int);

//  The following operations do allocation/deallocation of sets and setlists.
extern set NewSet(void);
extern void FreeSet(set *);
extern setlist NewSetList(int);
extern void FreeSetList(int,setlist *);
extern void free_and_null (char **);

//  Input/Output
extern void ReadSet(FILE *, set *);
extern void OutSet(FILE *,int, set);
extern void OutSetByRank(FILE *, set);

//  Initialization
extern void GetEmptySet(set);
extern void GetFullSet(set);

//  Operations
extern void SetInsert(int,set);
extern void SetDelete(int,set);
extern void Intersection(set,set,set);
extern void Union(set A,set B,set C);
extern void GetSet(set,set);
extern void SetMinus(set,set,set);
extern int SetOrder(set);
extern int FindLargestElement(set);
extern void Complement(set,set);

//  tests
extern int MemberOfSet(int,set);
extern int IntersectTest(set,set);
extern int Empty(set);
extern int CompareSets(set,set);

//  timing
extern void start_time(void);
extern double prn_time(void);
```

Appendix 3 setlib0.c

This file gives the implementations for the set operations.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "defs.h"
#include "setlib0.h"
#include <time.h>

#define MAXSTRING 100

 int WORDS;
 set V=NULL;
 set _AuxSet1=NULL;

// Allocation/Deallocation

void free_and_null (char **ptr)
{
 if (*ptr != NULL)
 {
  free (*ptr);
  *ptr = NULL;
 }
} /* free_and_null */

set NewSet(void)
// Allocate storage for set.
{
 set S;
 S=(set)calloc(WORDS,sizeof(UINT));
 return(S);
}

void FreeSet(set *S)
// Deallocate storage for set.
{
 free_and_null((char **) S);
}

setlist NewSetList(int m)

// Allocate storage for a list of m sets.
{
 int i;
 setlist L;
 L=(setlist )calloc(m,sizeof(set));
 for(i=0;i<m;i++) L[i]=NewSet();
 return(L);
}

void FreeSetList( int m, setlist *S)
// Deallocate storage for a list of m sets.
```

```
{
 int i;
 if((*S)==NULL) return;
 for(i=0;i<m;i++)
 {
  FreeSet(&((*S)[i]));
 }
 free_and_null((char **) S);
}

void SetInit( int n)
//  Initialize environment with universe V={0,1,2,...,n-1}
{
 int i;
 ComputeWords(n);
 V=NewSet();
 for(i=0;i<n;i++) SetInsert(i,V);
  _AuxSet1=NewSet();

}

void ClearSet(int n)
{
 FreeSet(&V);
 FreeSet(&_AuxSet1);
}

/*
** Input\Output
*/

void OutSetByRank( FILE* F,set S)
//  Write to file F the rank of the set with SusetLexrank S.
{
 int i;
 for(i=0;i<WORDS;i=i+1) fprintf(F," %x",S[i]);
}

void OutSet( FILE* F,int n, set S)
//  Write to file F the set with SusetLexrank S.
{
 int u,v;
 fprintf(F,"{");
 u=0; while((u<n) && !MemberOfSet(u,S)) u=u+1;
 if(u!=(n))
 {
  fprintf(F,"%d",u);
  for(v=u+1;v<n;v=v+1)
   if ( MemberOfSet(v,S) ) fprintf(F,",%d",v);
 }
 fprintf(F,"}");
}

void ReportBadSet( char * X )
{
 fprintf(stderr,"Error: Read Bad Set: %s\n",X);
 exit(1);
```

```
}

void ReadSet( FILE *F, set *S)
//  Read a set from the file F and return its SubsetLexRank in S.
{
 int i,L,u;
 char X[1000];
 GetEmptySet(*S);
 fgets(X,1000,F); if (X[0]=='\n') fgets(X,1000,F);
 L=strlen(X);
 i=0;
 while(i<L && X[i]!=LBRACE) i=i+1;
 if (i==L) ReportBadSet(X);
 while(i<L && X[i]!=RBRACE)
 {
  if (i==L) ReportBadSet(X);
  i=i+1;
  u=atoi(X+i);
  SetInsert(u,*S);
  while((i<L) && !(X[i]==COMMA || X[i]==BLANK || X[i]==RBRACE) ) i=i+1;
  }
}

// Operations
/*****************************************************/
 int look[256]=
    {
     0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
     1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
     1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
     2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
     1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
     2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
     2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
     3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8
    };

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

int SetOrder(set S)
//  Algorithm 1.8 Return the number of ones in the set with //
//  SubsetLexRank S.
{
 int ans,i;
 UINT x;
 ans = 0;
 for(i=0;i<WORDS;i++)
 {
  x=S[i];
  while (x)
  {
   ans += look[x&(UINT)0377];
   x >>= (UINT)8;
  }
 }
 return(ans);
}
```

```
void SetInsert(int u, set A)
/*
**  Algorithm 1.3 Replaces A with the SubsetLexRank of
**   SubsetLexUnrank(A) union {u}.
*/
{
 int i;
 UINT j;
 j=WORDSIZE-1-(u % WORDSIZE );
 i=u/WORDSIZE;
 A[i]=A[i]|((UINT)1<<j);
}

void SetDelete(int u, set A)
/*
**  Algorithm 1.4
**  Replaces A with the SubsetLexRank of SubsetLexUnrank(A)\{u}.
*/
{
 int i;
 UINT j;
 j=WORDSIZE-1-(u % WORDSIZE );
 i=u/WORDSIZE;
 A[i]=A[i]&~((UINT)1<<j);
}

int MemberOfSet(int u, set A)
/*
**  Algorithm 1.5
** Returns true if u is in the set with SubsetLexRank S.
*/
{
 int  i;
 UINT  j;
 j=WORDSIZE-1-(u % WORDSIZE );
 i=u/WORDSIZE;
 if( A[i]&((UINT)1<<j) )
 {
  return(true);
 }
 else
 {
  return(false);
 }
}

void Intersection(set A,set B,set C)
//   Algorithm 1.7   C gets A intersect B
{
 int i;
 for(i=0;i<WORDS;i++) _AuxSet1[i]=A[i]&B[i];
 for(i=0;i<WORDS;i++) C[i]=_AuxSet1[i];
}

void Union(set A,set B,set C)
//  Algorithm 1.6  C gets A union B
```

```
{
 int i;
 for(i=0;i<WORDS;i++) _AuxSet1[i] = A[i] | B[i];
 for(i=0;i<WORDS;i++) C[i]=_AuxSet1[i];
}


int IntersectTest(set A,set B)
/*
**  Returns true if A intersect B
**  is nonempty; otherwise false is returned.
*/
{
 int i;
 for(i=0;i<WORDS;i++) if(A[i]&B[i]) return(true);
 return(false);
}

void SetMinus(set A,set B,set C)
/*
**  C gets A \ B
*/
{
 int i;
 for(i=0;i<WORDS;i++) _AuxSet1[i]=A[i]&~B[i];
 for(i=0;i<WORDS;i++) C[i]=_AuxSet1[i];
}

void GetEmptySet(set A)
/*
**  A gets {}
*/
{
 int i;
 for(i=0;i<WORDS;i++) A[i]=0;
}

void GetFullSet(set A)
/*
**  A gets the universe set,
**  i.e. A gets V.
*/
{
 int i;
 for(i=0;i<WORDS;i++) A[i]=V[i];
}


void GetSet(set A, set B)
/*
**  A gets B.
*/
{
 int i;
 for(i=0;i<WORDS;i++) A[i]=B[i];
}
```

```
int Empty(set x)
/*
**  Returns true if x is {}; otherwise false is returned.
*/
{
 int i;
 for(i=0;i<WORDS;i++) if(x[i]!=0) return(false);
 return(true);
}

int CompareSets(set A,set B)
/*
**  Returns -1, 0, or 1 if the SubsetLexRank(A) is
**         less than, equal to, or greater than  the
**  SubsetLexRank(B) respectively.
*/
{
 int i;
 i=WORDS-1;
 while((A[i]==B[i]) && (i>0)) i=i-1;
 if((i==0) && (A[0]==B[0])) return(0);
 if(A[i]>B[i]) return(1);
 return(-1);
}

int FindLargestElement(set S)
/*
**   Returns the largest  element u in the set S
*/
{
 int i,j;
 UINT a,d;
 i=WORDS-1; while(!S[i]) i=i-1; a=S[i];
 d=(UINT)1;
 j=0;
 while(!(d&a))
 {
  d=(d<<1);
  j=j+1;
 }
 return( (WORDSIZE-1-j)+(i*WORDSIZE) );
}

void Complement(set A, set B)
//  B gets the complement of A.
{
  int i;
  for(i=0;i<WORDS;i++) B[i]=V[i]&-A[i];
}

// code to time program, from [12]
typedef struct
{
 clock_t begin_clock, save_clock;
 time_t begin_time, save_time;
}
 time_keeper;
```

```
static time_keeper tk; /* known only to this file */

void start_time()
{
 tk.begin_clock = tk.save_clock = clock();
 tk.begin_time = tk.save_time = time(NULL);
}

double prn_time()
{
 char s1[MAXSTRING], s2[MAXSTRING];
 int field_width, n1, n2;
 double clocks_per_second = (double) CLOCKS_PER_SEC,
   user_time, real_time;
 user_time = (clock() - tk.save_clock) / clocks_per_second;
 real_time = difftime(time(NULL), tk.save_time);
 tk.save_clock = clock();
 tk.save_time = time(NULL);

 /* print the values found, and do it neatly */

 n1 = sprintf(s1, "%.1f", user_time);
 n2 = sprintf(s2, "%.1f", real_time);
 field_width = (n1 > n2) ? n1 : n2;
  printf("%s%*.1f%s\n%s%*.1f%s\n\n",
    "User time : ", field_width, user_time, " seconds",
    "Real time : ", field_width, real_time, " seconds");
 return user_time;
}
```

Appendix 4 maxc_int.c

This file contains C language implementations of the algorithms discussed in this report.

```
// maxc_int.c

// library files
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h> // for floor
#include <time.h> // for time
#include "defs.h"
#include "setlib0.h"

#define false 0
#define true 1

// a graph is an array of sets
// actually an array of adjacency lists
typedef set * graph;

FILE *f;   // input file

// global variables
int NODES;  // number of nodes in graph
int n;    // number of vertices in graph
int kp = 1;     // number of partitions in graph
double p = 0.5;// density of graph
int NumColors; // number of colors used
int OptSize; // size of largest clique found so far
int cross_over = 216; // lowest value where greedy beats dsatur coloring for
fixed p, k

typedef int BOOL;
// flags used for debugging purposes
BOOL debug1 = false;
BOOL debug2 = false;
BOOL debug3 = false;
BOOL debug4 = false;
BOOL interchange = false; // set to true to implement interchange
BOOL color_change = false;

char bounding_function;
char WBOUND[80]; // name of bounding technique used

graph A;   // contains the adjacency matrix for the graph

// lists of sets
setlist C, B;
setlist ColorClass; // contains list of vertices with the same color
setlist Component; // contains list of vertices in same component

// arrays
int * X; // contains list of vertices in optimal clique so far
int * OptClique; // maximum clique
int * Color;   // array of colors
int * Degree;  // degree of ith vertex
int * Mark;    // component number that vertex is in

// function pointer to bounding function for pruning search tree
```

```
int (*BOUND)();

// sets
set Uncolored;  // set of uncolored vertices
set used;   // set of colors used
// set of colors with one vertex of that color adjacent to v_new
set Adj_Colors;

// find the maximum degree of any vertex in the set S from a to n
int find_max_degree(int a, int n, set S, int* Degree)
{
 int i, max_degree = 0;
 for (i = a; i < n; i++)
 {
  if (MemberOfSet(i, S))
  {
   if (Degree[i] > max_degree)
    max_degree = Degree[i];
  }
 }
 return max_degree;
}

BOOL Valid_Coloring()
{
 int i, j;
 // for each pair of vertices in the graph
 // if i and j have the same color, then
 // if they are also adjacent, coloring is invalid
 for (i = 0; i < n; i++)
  for (j = i + 1; j < n; j++)
   if ((Color[i] != -1) && (Color[i] == Color[j])
      && (MemberOfSet(j, A[i])))
   {
    printf("Error: vertex %d and vertex %d are adjacent, and \n are both
colored using color %d\n", i, j, Color[i]);
    return false;
   }
 printf("Coloring is valid\n");
 return true;
}

// print values of an array in set S from a to n
void print_array(int a,int n,int *A, set S)
{
 int i;
 printf("[ ");
 for (i = a; i < n; i++)
  if (MemberOfSet(i, S))
   printf("%d ", A[i] );
  else
   printf(" - " );
 printf("]");
 printf("\n");
}

void WaitForUser( )
{
 int x = 0;;
 printf("Enter any integer to continue\n");
 scanf("%d", &x);
}
```

```
// print values of a set from a to n
void print_set(int a, int n, set S)
{
 int i;
 printf("[ ");
 for (i = a; i < n; i++)
  if (MemberOfSet(i, S))
   printf("%d ", i);
 printf("]");
 printf("\n");
}

// choose next vertex from set S of maximum degree from a to n
int choose_vertex(int a, int n, set S, int max_deg, int* Degree)
{
 typedef int BOOL;
 BOOL found = false;
 int i = a, v_new = a;
 while ((i < n ) && (!found))
 {
  if (MemberOfSet(i, S) && (Degree[i] == max_deg))
  {
   v_new = i;
   found = true;
   return v_new;
  }
  else
   i++;
 }
 if (!found)
 {
  printf("Error: vertex not found in set S\n");
  return -1;
 }
 return v_new;
}// end if

// uses a depth-first search to find the connected
// components in a graph
void DFS_component(int v, int comp_count, set S)
{
 int i;
 Mark[v] = comp_count; // vertex v is marked  with comp_count
 // vertex v inserted into Component set
 SetInsert(v, Component[comp_count]);

 for (i = 0; i < n; i++)
 {
 if (MemberOfSet(i, S) && MemberOfSet(i, A[v]))
  if (Mark[i] == 0) // vertex i not in any previous component
   DFS_component(i, comp_count, S);
 }
}

// finds the component in a set S containing vertex
void Find_Component(int vertex, set S, int n)
{
 int v; // vertex loop counter
 int comp_count = 1; // number of components in the graph
 GetEmptySet(Component[1]);
 for (v = 0; v < n ; v++)
 {
  if (MemberOfSet(v, S))
```

```
    Mark[v] = 0;  //vertices start in no component
  }
  comp_count = 1;// counter for current component

  // find component of vertex
  DFS_component(vertex, comp_count, S);
}

// implements the bichromatic interchange
int Interchange(int max_color_used, int colour, int v_new)
{
  int i, j, v;
  int num_components = 0; // number of components in graph
  int x;  // vertex

  set temp_set;
  set temp_set2;
  // subset of vertices colored using exactly two colors
  set Bi_Colored;

  setlist UA; // uniquely adjacent to v_new
  UA = NewSetList(n);

  temp_set = NewSet();
  temp_set2 = NewSet();
  Bi_Colored = NewSet();

  GetEmptySet(Adj_Colors);

  // first step: find the set of colors Adj_Colors which are
  // uniquely adjacent to the next vertex v_new
  for (i = 0; i <= max_color_used; i++)
  {

    Intersection(ColorClass[i], A[v_new], temp_set);
    if (SetOrder(temp_set) == 1)
    {
      SetInsert(i, Adj_Colors);
      Union(UA[v_new], temp_set, UA[v_new]);
    }
  }

  //find connected <i,j> components of subgraph which are
  // colored using only the colors in the set Adj_Colors
  if (SetOrder(Adj_Colors) >= 2)
  {
    // find an <i,j> component of vertices colored so far
    // such that exactly one vertex in the component is
    // adjacent to v_new
    for (i = 0; i < colour; i++)
    {
      if (MemberOfSet(i, Adj_Colors))
      {
        for (j = i + 1; j < colour; j++)
        {
          if (MemberOfSet(j, Adj_Colors))
          {
            Union(ColorClass[i], ColorClass[j], Bi_Colored);
            // Bi_Colored is the set of vertices colored
            // with either i or j

            // test if we can do an i,j interchange
            // on some i,j component
```

```
      // first, find the vertex of color i adjacent to v_new
         Intersection(ColorClass[i], A[v_new],temp_set);
         for (v = 0; v < n; v++)
         {
          // true only for one vertex in the set
          if (MemberOfSet(v, temp_set))
          {
           Find_Component(v, Bi_Colored, n);
           Intersection(Component[1], A[v_new], temp_set2);
      //Intersection(ColorClass[j], temp_set2, temp_set2);
           if (SetOrder(temp_set2) == 1) // if true, we interchange
           {
      // exchange colors i and j on all vertices in Component[1]
            for (x = 0; x < n; x++)
            {
             if (MemberOfSet(x, Component[1]))
             {
              if (MemberOfSet(x, ColorClass[i]))
              {
               Color[x] = j;
               SetInsert(x, ColorClass[j]);
               SetDelete(x, ColorClass[i]);
              }
              else if (MemberOfSet(x, ColorClass[j]))
              {
               Color[x] = i;
               SetInsert(x, ColorClass[i]);
               SetDelete(x, ColorClass[j]);
              }
              else
              {
               printf("Error:Bi_Colored set should contain only i and j\n");
               exit(1);
              }
             }
            }// end for
            return i;
           }// end if

          }// end if
         }// end for
        }// end if
       }// end for
      }// end if
     }// end for
    }// end if

    // de-allocate memory
    FreeSet(&Bi_Colored);
    FreeSet(&temp_set);
    FreeSet(&temp_set2);
    FreeSetList(n, &UA);
    return -1; // if we got this far, no interchange was possible
}

int NoBound( )
{
 // no bounding function used
 return(n);
}

int SizeBound(int L)
```

```
//  Algorithm 4.15
{
  int k;
  k = SetOrder(C[L]);
  return (k+L);
}


int GreedyColor( set U, int a )
/*
   Algorithm 4.16
 color the vertices of the graph using a greedy algorithm
 use the result in a bounding function
*/
{
 int h;   // color used on current vertex
 int i;   // counter in for loop
 int k = 0; // number of colors used for graph is initialized to zero

 // initialize colors
  for (i = 0; i < n; i++)
  Color[i] = -1;

 for (i = a; i < n; i++) // for each vertex in the graph, color it in turn
 {
   if( MemberOfSet(i,U) ) // Color vertex i, if i is in the set U
   {
    // try to color with lowest available color
      h = 0;
    // keep trying until lowest possible color is found for vertex
      while ( (h < k) && IntersectTest(A[i],ColorClass[h]))
        h = h + 1;  // increment color h as long as necessary
      if(h == k) // if a new color is necessary
      {
          k = k + 1; // number of colors used so far is incremented
          GetEmptySet(ColorClass[h]); // new color class is initially empty
      }
      // add vertex i to set of vertices colored using color h
      SetInsert(i, ColorClass[h]);
      // color vertex i with color h
      Color[i] = h;
   }
 }
 return (k); // return the number of colors needed to color the graph
}

int GreedyColor_Int( set U, int a )
/*
   Algorithm 4.16
 color the vertices of the graph using a greedy algorithm
 use the result in a bounding function
*/
{
 int h;   // color used on current vertex
 int i;   // counter in for loop
 int k = 0; // number of colors used so far is initialized to zero
 int new_color = 0;

 for (i = 0; i < n; i++)
  Color[i] = -1;
 // for each vertex in the graph, color it in turn
 for (i = a; i < n; i++)
 {
```

```
   // Color vertex i, if it has not yet been colored
  if( MemberOfSet(i,U) )
  {
     // try to color with lowest available color
     h = 0;
   // keep trying until lowest possible color is found for vertex
     while ( (h < k) && IntersectTest(A[i],ColorClass[h]))
       h = h + 1;  // increment color h as long as necessary

   // interchange if possible
   if ((h == k) && (k > 0)) // new color necessary
   {
    new_color = Interchange(k - 1 , h, i);

   // if interchange occurred, v_new is colored using value
   // returned by Interchange function
    if (new_color >= 0)
      h = new_color;
   }
     if (h == k) // if a new color really is necessary
     {
      k = k + 1; // number of colors used so far is incremented
     // new color class is initially empty
      GetEmptySet(ColorClass[h]);
     }
     // add vertex i to set of vertices colored using color h
     SetInsert(i, ColorClass[h]);
     // color vertex i with color h
     Color[i] = h;
  }
  }
  Valid_Coloring(); // checks if coloring obtained is valid
  return (k); // return the number of colors needed to color the graph
}


int SamplingBound( int L , int a)
// Algorithm 4.17
{
   int u, k;
   if(L == 0)
    return NumColors;
   GetEmptySet(used);
   for(u = a; u < n; u++)
    if(MemberOfSet(u,C[L]))
     SetInsert(Color[u],used);
   k = SetOrder(used);
   return (k + L);
}

/*****************************************************/
int DSatur(set U, int a)
//  DSatur Algorithm to color graph vertices
{
  // dsatur algorithm defs
  // variables
  BOOL found = false;

  // true if maximum clique found, else false
  int clmax = false;
  int i, j, ii;
  int max_sat_deg;
  int max_degree;
```

```
int v_new;
int max_col_neighbors;
int colour = 0;
int new_color;
int max_color_used = 0;

// dynamic arrays
int *cdeg;    // number of colored neighbours
int *satdeg;   // saturation degree
int *Degree_x;  // degree of ith vertex in set XX

// sets of vertices
set neighbour;
set temp_set;
set W;    // set of uncolored vertices
set XX;    // set of uncolored vertices of maximum degree
      // from which new vertex is selected
// list of sets
setlist neighbcol;  // set of colors in neighbourhood of vertex

// allocate memory for dsatur algorithm structures

// arrays of degrees of vertex set
 cdeg = (int *)calloc(n,sizeof(int));
 satdeg = (int *)calloc(n,sizeof(int));
 Degree_x = (int *)calloc(n,sizeof(int));

// sets of vertices
neighbour =  NewSet();
temp_set = NewSet();
W = NewSet();
XX = NewSet();

// lists of sets
neighbcol = NewSetList(n);

// initialize colors
for (i = 0; i < n; i++)
{
 Color[i] = -1;
 GetEmptySet(ColorClass[i]);
}

// step 1 - initialize arrays
for (i = a; i < n; i++)
{
 cdeg[i] = 0;  // number of colored neighbors of vertex i
 satdeg[i] = 0; // number of different colored neighbors of vertex i
 Degree_x[i] = 0;  // degree of vertex i in set XX
 GetEmptySet(neighbcol[i]);
}

clmax = false;  // true when max clique is found
GetSet(W, U);  // assign set U to set W

while (!Empty(W)) // continue until all vertices have been colored
{
// XX is a set containing the uncolored vertices (in W)
// of maximum saturation degree
// first find maximum saturation degree
 max_sat_deg = 0;
 max_sat_deg = find_max_degree(a, n, W, satdeg);
```

```
// next insert any vertices of maximum saturation
// degree into the set XX
for (i = a; i < n; i++)
{
 if (MemberOfSet(i, W))
 {
  if (satdeg[i] == max_sat_deg)
   SetInsert(i, XX);
 }
}
// the above has defined the set XX of vertices of
// maximum saturation degree in W
// next we must choose the new vertex to be colored
// from the set XX

if (!clmax) // clmax == false choose v_new with max degree in XX
{
 // first find the degree of each vertex within set XX
 for (i = a; i < n; i++)
 {
  if (MemberOfSet(i, XX))
  {
   Intersection(A[i], XX, temp_set);
   Degree_x[i] = SetOrder(temp_set);
  } // end if
 } // end for
 // choose new vertex from set XX with maximum degree
 // first find the maximum degree of colored neighbours
 max_degree = 0;
 max_degree = find_max_degree(a, n, XX, Degree_x);
 // next choose a vertex from XX with maximum degree
 v_new = choose_vertex(a, n, XX, max_degree, Degree_x);
}// end if
else
{
 // choose new vertex from set XX with maximum number of colored neighbours
 // first find the maximum number of colored neighbours
 max_col_neighbors = 0;
 max_col_neighbors = find_max_degree(a, n, XX, cdeg);
 // next choose a vertex from XX with max number of colored neighbours
 v_new = choose_vertex(a, n, XX, max_col_neighbors, cdeg);
}// end else
GetEmptySet(XX);    // re-initialize set XX
// color vertex v_new with lowest available color
colour = 0; // start with lowest possible color
// increment color until it is not equal to
// the color of a neighbour of v_new
while (IntersectTest(A[v_new], ColorClass[colour]))
   colour ++;

if (interchange)
{
 // interchange if necessary
 if (colour > max_color_used)
 {
  if (debug3)
  {
   printf("\nCalling Interchange: Coloring used so far: ");
   print_array(a, n, Color, U);
   for (ii= 0; ii < colour; ii++)
   {
    printf("Vertices colored with color %d = ", ii);
    print_set(0, n, ColorClass[ii]);
```

```
      }
      printf("Saturation degrees = ");
      print_array(0, n, satdeg, U);
     }
    }
    new_color = Interchange(max_color_used, colour, v_new);

    if (new_color >= 0)
    {
     // if interchange occurred, v_new is colored using the
     // value returned by Interchange function
     colour = new_color;
     new_color = -1; // reset to -1
     color_change = true;
     if (debug3)
     {
      printf("\n Interchange did take place: ");
      printf("Next vertex to be colored = %d\n", v_new);
     }
    }
   }

   Color[v_new] = colour;
   //GetEmptySet(ColorClass[colour]);
   SetInsert(v_new, ColorClass[colour]);
   if (colour > max_color_used)
    max_color_used = colour;
   // remove vertex from list of uncolored vertices
   SetDelete(v_new, W);

   // update saturation degrees of each vertex i
   for (i = 0; i < n; i++) // for each vertex i
   {
    cdeg[i] = 0;
    satdeg[i] = 0; // reset to zero
    GetEmptySet(neighbcol[i]);
    for (j = 0; j <= max_color_used; j++) // for each color used
    {
     // update arrays after coloring vertex
     // if vertex i has a neighbour with color j,
     // then increment saturation degree of i
     if (IntersectTest(A[i], ColorClass[j]))
     {
      Intersection(A[i], ColorClass[j], temp_set);
      cdeg[i] = cdeg[i] + SetOrder(temp_set);
      satdeg[i] ++;
      SetInsert(j, neighbcol[i]);
     }
    }
   }
} // end while

// Finally, de-allocate memory used by DSatur
// function in reverse order of allocation

// First: lists of sets
FreeSetList(n, &neighbcol);

// sets of vertices
FreeSet(&XX);
FreeSet(&W);
FreeSet(&temp_set);
FreeSet(&neighbour);
```

```
 // arrays
 free(Degree_x);
 free(satdeg);
 free(cdeg);

 printf("\n");
 return (max_color_used + 1);
}
/*****************************************************/

int GreedyBound( int L , int a)
/*
**  Algorithm 4.18
*/
{
  int bound = 0;
  bound = (GreedyColor(C[L], a) + L);
  return bound;
}

int GreedyBound_DSatur( int L , int a)
//  Algorithm 4.18
{
    int bound = 0;
  bound = (DSatur(C[L], a) + L);
    return bound;
}

void MaxClique2(int L )
//  Algorithm 4.19
{
    int start,i,x;
  NODES++;
  // L = depth of current node in search tree = length of list X
  if (L == 0)
  {
  GetSet(C[L], V);
  start = 0;
  }
  else
  {
  // WORDS = ( n / 32 ) + 1, defined in "defs.h"
  for(i = 0; i < WORDS; i++)
   C[L][i] = A[X[L-1]][i] & B[X[L-1]][i] & C[L-1][i];
  start = X[L-1] + 1;
  }

  for(x = start; x < n; x++)
   if(MemberOfSet(x, C[L]))
   {
   if (SetOrder(C[L]) < cross_over)
   {
     if ( BOUND(L, start) <= OptSize)
     {
    return;
     }
   }// end if cross over
   {
      X[L] = x;
      if ((L + 1) > OptSize)
      {
      OptSize = L + 1;
```

```
        for(i = 0; i <= L; i++)
         OptClique[i] = X[i];
        } // end if
        MaxClique2(L + 1);
     }// end
     } // end if
}// end MaxClique2
/*****************************************************/

graph NewGraph(int n)
//  Allocate storage for a graph on order n
{
    int i;
    graph A;
    A = (graph)calloc(n, sizeof(set));
    for(i = 0; i < n; i++)
    {
   A[i] = NewSet();
    }
    return(A);
}

void FreeGraph(int n, graph A)
//  Allocate storage for a graph on order n
{
    int i;
    for(i = 0; i < n; i++)
     FreeSet(&A[i]);
    free(A);
}

void ReadGraphEdges(FILE *F, graph *A, int  *n)
/*
   Read and allocate storage for a graph from the file
   F which contains a list of edges. The order of the
   graph is returned in n.
*/
{
  int i,u,v,m;
  // input number of edges (m) and vertices (n)
  fscanf(F,"%d %d", &m, n);
  ComputeWords(*n);
  *A = NewGraph(*n);

  // set up symmetric adjacency matrix for graph
  for(i = 0; i < m;i++)
  {
     fscanf(F," %d %d",&u,&v);
     SetInsert(u,(*A)[v]); // vertex u is adjacent to vertex v
     SetInsert(v,(*A)[u]); // vertex v is adjacent to vertex u
  }
}

void RandomGraph(graph *A, int m, double p, int k)
/*
   Create a random k-partite graph with n vertices,
   k partitions, and probability p that any two
   vertices in different partitions are adjacent
*/
{
  double random;
  int u, v; // loop counters
  int start;
```

```
  int size; // size of each of the k partitions

  // Seed the random number geneator
  // int seed = time (0); // variable seed value
  int seed = 100; // use a fixed seed to compare results
  srand(seed); // seed the random number generator

  n = m;
  // assume for simplicity that n is a multiple of k
  size = n / k;

  // allocate memory space for adjacency matrix A
  ComputeWords(n);
  *A = NewGraph(n);

  // set up symmetric adjacency matrix for graph
  for(u = 0; u < n - size; u++)
  {
     start = size + (int)floor(u / size) * size;
   for (v = start; v < n; v++)
   {
    // generate a random number p between 0 and 1
   random = rand() * (1.0 / RAND_MAX);
   if (random > 0.0 && random < p)
   {
    SetInsert(u,(*A)[v]); // vertex u is adjacent to vertex v
    SetInsert(v,(*A)[u]); // vertex v is adjacent to vertex u
   }
   }
  }
}

/*********************************************************/
int main( )
{
 int v;
 int i, j;
 interchange = false;

 // select which graph to color
// f = fopen("a:\\clique\\graphs\\small_graph_7_5.txt", "r");
   f = fopen("a:\\clique\\graphs\\sample", "r");
// f = fopen("a:\\clique\\graphs\\small_graph_10_7.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand50_5.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand100_5.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand150_5.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand200_5.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand250_5.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand25_75.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand50_75.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand75_75.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand100_75.txt", "r");
// f = fopen("a:\\clique\\graphs\\rand125_75.txt", "r");

 ReadGraphEdges(f, &A, &n); // read input file
 fclose(f);       // close input file
  OptSize = 0;  // size of largest clique found so far

// n = 100;
  SetInit(n); // initializes the set V to {0, 1, 2, ..., n-1}

  // allocate memory for dynamic arrays, sets, setlists
```

```
 // set lists
 B = NewSetList(n);
 C = NewSetList(n);
 Component = NewSetList(n);
 ColorClass = NewSetList(n);

 // sets
 Uncolored = NewSet();
 used = NewSet();
 Adj_Colors = NewSet();

 // arrays
 X =         (int *) calloc(n,sizeof(int));
 OptClique = (int *) calloc(n,sizeof(int));
 Color =      (int *) calloc(n,sizeof(int));
 Degree = (int *) calloc(n,sizeof(int));
 Mark =     (int *) calloc(n,sizeof(int));

// p = 0.5;
// kp = 20;  // number of partitions
// RandomGraph(&A, n, p, kp);

 // set debugging flags
 interchange = false;
 debug1 = false;
 debug2 = false;
 debug3 = false;
 debug4 = false;
 if (debug3)
   for (i = 0; i < n; i++)
   print_set(0, n, A[i]); // display adjacency matrix

 // initialize degrees of vertices O(n squared)
 for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
   if (MemberOfSet(i, A[j]))
    Degree[i]++;

 // prompt user for bounding function to be used
 printf("Please enter the type of bounding function you wish to use\n");
 printf("N for no bound\n");
 printf("Z for size bound\n");
 printf("G for greedy bound with greedy coloring algorithm \n");
 printf("R for greedy bound with DSatur coloring algorithm\n");
 printf("S for sampling bound with greedy coloring algorithm\n");
 printf("I for sampling bound with greedy coloring algorithm with
interchange\n");
 printf("D for sampling bound with DSatur coloring algorithm\n");

 //read in user's choice of bounding function
 scanf("%c", &bounding_function);
 start_time(); // start timing of program

switch(bounding_function)
{
  case 'N':
  case 'n':
    sprintf(WBOUND,"%s","None          ");
    BOUND = NoBound;
  break;
  case 'Z':
  case 'z':
    sprintf(WBOUND,"%s","Size          ");
```

```
    BOUND = SizeBound;
  break;
  case 'G':
  case 'g':
    sprintf(WBOUND,"%s","Greedy        ");
    BOUND = GreedyBound;
  break;
  case 'R':
  case 'r':
    sprintf(WBOUND,"%s","Greedy using DSatur Coloring Algorithm    ");
    BOUND = GreedyBound_DSatur;
  break;
  case 'S':
  case 's':
    sprintf(WBOUND,"%s","Sampling using Greedy Coloring Algorithm  ");
    NumColors = GreedyColor(V,0);
    BOUND = SamplingBound;
  break;
  case 'I':
  case 'i':
    sprintf(WBOUND,"%s","Sampling using Greedy Coloring Algorithm with
Interchange  ");
    NumColors = GreedyColor_Int(V,0);
    BOUND = SamplingBound;
  break;
  case 'D':
  case 'd':
    sprintf(WBOUND,"%s","Sampling using DSatur Coloring Algorithm   ");
   NumColors = DSatur(V,0);
    BOUND = SamplingBound;
  break;
  default:
    printf("Illegal sort method\n");
    exit(1);
  break;
}

 // initialize set B[v] to the set of vertices numbered higher than v
 GetSet(B[0], V);
 SetDelete(0, B[0]);

 for (v = 1; v < n; v++)
 {
   GetSet(B[v], B[v-1]);
   SetDelete(v, B[v]);
 }

 NODES = 0;  // number of nodes in search tree initialized to zero
 MaxClique2(0); // call maxclique function
 printf("Number of vertices in the graph = %11d   \n", n);
 if (kp > 1)
 {
 printf("Graph density = %29.2f   \n", p);
   printf("Number of partitions            = %11d   \n", kp);
 }
 printf("Type of bound used = %35s   \n",WBOUND);
 printf("Maximum clique size found = %16d \n",OptSize);
 printf("Number of nodes in search tree = %11d\n",NODES);

 // print the vertices in the maximum clique
 printf("Max. Clique = ");
 print_array(0, OptSize, OptClique, V);
 prn_time();
```

```
 printf("---------------------------------\n");

 // de-allocate memory in reverse order of allocation

 // lists of sets
 FreeSetList(n, &B);
 FreeSetList(n, &C);
 FreeSetList(n, &Component);
 FreeSetList(n, &ColorClass);

 // sets of vertices
 FreeSet(&Adj_Colors);
 FreeSet(&used);
 FreeSet(&Uncolored);

 // arrays
 free(Mark);
 free(Degree);
 free(Color);
 free(OptClique);
 free(X);

 return (0);
}
```