SIAM J. SCI. COMPUT.
Vol. 29, No. 3, pp. 1042–1072

NEW ACYCLIC AND STAR COLORING ALGORITHMS WITH APPLICATION TO COMPUTING HESSIANS*

ASSEFAW H. GEBREMEDHIN†, ARIJIT TARAFDAR†, FREDRIK MANNE‡, AND ALEX POTHEN†

Abstract. Acyclic and star coloring problems are specialized vertex coloring problems that arise in the efficient computation of Hessians using automatic differentiation or finite differencing, when both sparsity and symmetry are exploited. We present an algorithmic paradigm for finding heuristic solutions for these two NP-hard problems. The underlying common technique is the exploitation of the structure of two-colored induced subgraphs. For a graph G on n vertices and m edges, the time complexity of our star coloring algorithm is $O(n\overline{d}_2)$, where $\overline{d}_k$, a generalization of vertex degree, denotes the average number of distinct paths of length at most k edges starting at a vertex in G. The time complexity of our acyclic coloring algorithm is larger by a multiplicative factor involving the inverse of Ackermann's function. The space complexity of both algorithms is $O(m)$. To the best of our knowledge, our work is the first practical algorithm for the acyclic coloring problem. For the star coloring problem, our algorithm uses fewer colors and is considerably faster than a previously known $O(n\overline{d}_3)$-time algorithm. Computational results from experiments on various large-size test graphs demonstrate that the algorithms are fast and produce highly effective solutions. The use of these algorithms in Hessian computation is expected to reduce overall runtime drastically.

Key words. acyclic coloring, star coloring, computation of Hessians, automatic differentiation, finite differences

AMS subject classifications. 05C15, 05C85, 68R10

DOI. 10.1137/050639879

1. Introduction.

The cost involved in computing a Jacobian or a Hessian matrix using automatic differentiation or finite differencing is proportional to the number of columns in the matrix. When the target derivative matrix A is sparse, huge savings in runtime can be attained by first computing a *compressed* matrix B, where each column of B is a sum of several structurally "independent" columns of A, and then recovering the nonzero elements in A from B. Thus, an important component of the efficient computation of a sparse derivative matrix, whose sparsity pattern is known a priori, is the problem of partitioning the columns of the matrix into the fewest groups, each consisting of structurally independent columns.

The appropriate notion of structural independence—and the corresponding partitioning problem—depends on whether the derivative matrix to be computed is *symmetric* or *nonsymmetric*, and on whether the nonzero entries are to be recovered from the compressed matrix *directly* or via *substitution*. Conceptually, direct recovery corresponds to solving a diagonal system of equations, and recovery via substitution corresponds to solving a set of simple triangular systems of equations. Several previous studies have shown that these matrix partitioning problems can be modeled and effectively solved using specialized graph coloring problems and their algorithms [9, 12, 13, 14, 22, 27]. See our earlier work [17] for a comprehensive review.

*Received by the editors September 8, 2005; accepted for publication (in revised form) November 7, 2006; published electronically May 10, 2007. This work was supported by the U.S. National Science Foundation through grants ACI 0203722 and CCF 0515218, by DOE grant DE-FC02-01ER25476 and by subcontract B 533520 from the Lawrence Livermore National Laboratory.

http://www.siam.org/journals/sisc/29-3/63987.html

†Department of Computer Science and Center for Computational Science, Old Dominion University, Norfolk, VA 23529-0162 (assefaw@cs.odu.edu, tarafdar@cs.odu.edu, pothen@cs.odu.edu).

‡Department of Informatics, University of Bergen, N-5020 Bergen, Norway (fredrikm@ii.uib.no).

This paper is concerned with the design and implementation of effective algorithms for *star* and *acyclic* coloring, models for efficient computation of Hessians. In a star coloring, adjacent vertices receive distinct colors (distance-1 coloring), and every *path on four vertices* (P_4) uses at least three colors. In an acyclic coloring, adjacent vertices receive distinct colors, and every *cycle* uses at least three colors. Coleman and Moré [13] showed that star coloring models the partitioning problem that occurs in the computation of a Hessian via a direct method; Coleman and Cai [9] showed that the corresponding model in a substitution-based computation is acyclic coloring. Both problems, whose objectives are to minimize the number of colors used, are known to be NP-hard [9, 13]. As we show in section 2, these problems are also hard to approximate.

Star coloring as a model for direct Hessian computation was preceded by two less accurate models: the *distance*-2 coloring model due to McCormick [27] and the *restricted star* coloring model due to Powell and Toint [28]. Similarly, acyclic coloring as a model for Hessian computation via substitution was preceded by a less accurate model, *triangular* coloring, due to Coleman and Moré [13]. These models and the interrelationships amongst them are discussed in detail in section 5.

In a previous work [17], we suggested a simple heuristic algorithm for the star coloring problem. In each step of that algorithm, the color of a vertex is determined by examining every P_4 starting at the vertex. The time complexity of the algorithm for a graph on n vertices is $O(n\bar{d}_3)$, where $\bar{d}_k$ denotes the average number of paths of length at most k edges starting at a vertex. A similar $O(n\bar{d}_2)$-time algorithm for restricted star coloring, originally due to Powell and Toint [28], exists. However, when used for solving the star coloring problem, the latter algorithm uses significantly more colors than the $O(n\bar{d}_3)$-time algorithm mentioned earlier.

Although Coleman and Cai [9] identified acyclic coloring as the model for computing a Hessian via a substitution method, the algorithm they suggested is an algorithm for triangular coloring. In their approach, first a graph G' is constructed by adding edges to the input graph G, and then a distance-1 coloring algorithm is applied on the graph G'. A similar approach for triangular coloring was taken earlier by Coleman and Moré [13]. An optimal solution for the triangular coloring problem is not necessarily an optimal solution for the acyclic coloring problem. Moreover, working with a denser graph G' is undesirable for large-scale problems, since the approach may require excessive storage space and lead to memory overflow. As for the acyclic coloring problem, we are not aware of any previous practically relevant algorithm.

In this paper, we present new algorithms for both the acyclic and the star coloring problems. The common paradigm underlying the two algorithms is the exploitation of the structure of *two-colored induced subgraphs*. In the acyclic coloring case, every subgraph induced by any two color classes is a collection of *trees*; the corresponding structure in the star coloring case is a collection of *stars*. In both cases, the two-colored induced subgraphs *partition* the edge set of the input graph. Our algorithms are based on these observations.

The algorithms presented in this paper are *greedy*—they progressively extend a partial coloring by processing one vertex at a time. In each step, a vertex is assigned the *smallest* (assuming colors are positive integers) allowable color with respect to the current partial coloring, and the color assigned to the vertex remains unchanged during the course of the algorithm. Greedy coloring heuristics stand in contrast to *iterative, local improvement* heuristics, where the color assigned to a vertex may change several times during the course of an algorithm in an attempt to ultimately

reduce the number of colors used. Greedy coloring heuristics are in general fast, which makes them attractive approaches for solving subproblems in scientific computing applications, such as the case in this paper. An important issue in greedy coloring algorithms is the order in which vertices are processed. We consider several effective ordering techniques in this paper.

A key component of our new algorithms is the efficient management of two-colored induced subgraphs. For the acyclic coloring case, the disjoint-set data structure is used for this purpose; for the star coloring case, a simpler data structure is shown to be sufficient. In addition, a few intricate techniques which yet use simple data structures are employed to avoid two-colored cycles in the acyclic coloring case, and two-colored P_4's in the star coloring case. The time complexity of the star coloring algorithm is $O(n\bar{d}_2)$, and that of the acyclic coloring algorithm is larger by a multiplicative factor involving the inverse of Ackermann's function. The space complexity of both algorithms is $O(m)$, where m denotes the number of edges in a graph. The acyclic coloring algorithm will be presented (in section 3) before the star coloring algorithm (in section 4) to reflect that the former in an algorithmic sense corresponds to the more general case.

We have implemented the new acyclic and star coloring algorithms presented in this paper as well as several previously known algorithms for related coloring problems; the latter algorithms will be discussed in section 6. Results from experiments carried out on various large-size test graphs show that the new algorithms are highly effective, both in terms of the number of colors used and in practical runtime. On the test graphs used, the number of colors used by the acyclic coloring algorithm is observed to be very close to (within a factor of two) the *coloring number* of a graph—the number of colors used by a greedy distance-1 coloring algorithm that employs a *smallest last* vertex ordering. The coloring number is an important graph parameter with close relationship to graph notions such as degeneracy, maximal core, and arboricity. The runtime of the acyclic coloring algorithm is observed to be nearly the same as that of a greedy distance-2 coloring algorithm. In comparison with our implementation of an algorithm for triangular coloring—the previously known approach for Hessian computation via a substitution method—the acyclic coloring algorithm runs faster, uses fewer colors, and requires less memory and storage space. In agreement with time complexity analyses, our new star coloring algorithm is observed to be significantly faster than our previous $O(n\bar{d}_3)$-time algorithm—it offers a relative speedup as high as ten on the test graphs used. Moreover, the new algorithm uses slightly fewer colors. In comparison with the restricted star coloring algorithm of Powell and Toint [28], which is of the same time complexity as the new star coloring algorithm proposed here, our algorithm significantly reduces the number of colors used. In terms of observed runtime, the new star coloring algorithm is slower than the restricted star coloring algorithm by only a small factor, typically two.

2. Background.

2.1. Preliminary concepts and notations. Throughout this paper, we consider simple, undirected graphs without loops or parallel edges. Two classes of (sub)graphs that we have need for are trees and stars. A *tree* is a connected graph without any cycle, and a *forest* is a collection of trees. A *star* is a complete bipartite graph in which one of the vertex sets consists of a single vertex. In a star with at least two edges, we call the vertex at which the edges meet the *hub* of the star.

We denote an edge whose endpoints are the vertices v and w by vw. Two distinct vertices v and w are *distance-k neighbors* if a shortest path connecting them consists

TABLE 2.1
Characterization of subgraphs induced by a pair of color classes. The coloring variants are listed in an increasing order of restrictiveness; correspondingly, the two-colored induced subgraphs get increasingly less connected as one goes from top to bottom in the table.

Coloring variant	A two-colored induced subgraph
Distance-1 coloring	A bipartite graph
Acyclic coloring	A collection of trees
Star coloring	A collection of stars
Distance-2 coloring	A distance-2 matching

of at most k edges. We denote the set of distance-k neighbors of a vertex v, excluding the vertex v itself, by $N_k(v)$. As usual, the *degree* of a vertex is the number of edges incident on it. We extend this notion and define the *degree-k* of a vertex v, denoted by $d_k(v)$, to be the number of distinct paths of length at most k edges starting at v. (Two paths are *distinct* if they differ in at least one edge.) We denote the average degree-k in a graph by $\overline{d}_k$. Note that degree-1 is synonymous with degree, and that the degree-2 of a vertex v is equal to the number of edges that are incident either on v or on vertices adjacent to v. Note also that in general $d_k(v) \geq |N_k(v)|$.

A *distance-k coloring* of a graph $G = (V, E)$ is a mapping $\phi : V \to \{1, 2, \ldots, p\}$ such that whenever vertices v and w are distance-k neighbors, $\phi(v) \neq \phi(w)$. By definition, two distance-k neighboring vertices are distance-k' neighbors for $k' > k$, and a distance-k coloring is a distance-k'' coloring for $k'' < k$. The *kth power* of a graph $G = (V, E)$ is the graph $G^k = (V, F)$, where $vw \in F$ whenever vertices v and w are distance-k neighbors in G. A mapping ϕ is a distance-k coloring of a graph G if and only if ϕ is a distance-1 coloring of the graph G^k. A *star coloring* of a graph is a distance-1 coloring with the additional requirement that *every path on four vertices uses at least three colors.* An *acyclic coloring* of a graph is a distance-1 coloring with the further restriction that *every cycle uses at least three colors.* In a distance-k, a star, an acyclic, or any other coloring variant on a graph, a *color class* is a set of vertices that have received the same color.

The names star and acyclic coloring are due to the structures of subgraphs induced by any two color classes. In an acyclically colored graph, a subgraph induced by any two color classes is clearly a collection of trees, and thus acyclic. Similarly, in a star-colored graph, a subgraph induced by any two color classes is a collection of stars. It is easy to see that a two-colored connected induced subgraph here ought to be a star, for otherwise there would exist a two-colored path on four vertices, violating a condition of star coloring. These structures are used in the design of the acyclic and star coloring algorithms to be presented in the next two sections. Although not so useful for algorithm design, the structures of two-colored induced subgraphs in the cases of distance-1 and distance-2 coloring are also interesting. A coloring of a graph is a distance-1 coloring if and only if every two-colored induced subgraph is a bipartite graph. In the case of a distance-2 coloring, every two-colored induced subgraph is a set of edges in which a pair of edges is apart by a path of length at least two edges in the original graph, a structure appropriately called a *distance-2 matching.* These observations on two-colored induced subgraphs are summarized in Table 2.1.

2.2. Complexity results. The objective of an optimization problem associated with each of the coloring variants we have introduced is to minimize the number of colors used. Lin and Skiena [26] proved that the distance-k graph coloring problem is NP-hard for every fixed integer $k \geq 1$. Coleman and Moré [13] showed that the

star coloring problem is NP-hard even if the graph is bipartite. The acyclic coloring problem was proven to be NP-hard by Coleman and Cai [9].

The distance-1 coloring problem is also known to be hard to approximate: for a graph on n vertices, there exists $\delta > 0$ such that approximating distance-1 coloring within $O(n^\delta)$ is NP-hard (Corollary 10.1 in [5]). We use this to prove similar inapproximability results for acyclic and star coloring. We first introduce some notation. The *distance-k chromatic number* of a graph G, denoted by $\chi_k(G)$, is the least number of colors required to distance-k color G. The acyclic and star chromatic numbers are defined analogously, and are denoted by χ_a and χ_s, respectively.

THEOREM 2.1. *There exists fixed $\epsilon > 0$ such that it is NP-hard to approximate the acyclic chromatic number $\chi_a(G)$ of a graph G within $O(n^\epsilon)$, where n is the number of vertices in G.*

Proof. The reduction is from distance-1 coloring, and it is a small modification of the reduction for proving the NP-completeness of acyclic coloring in Coleman and Cai [9].

Assume that we are given a graph $G = (V, E)$ which we are required to distance-1 color; from G we create a new graph $H = (W, F)$ for acyclic coloring as follows. Let Δ denote the maximum degree of a vertex in G. Each edge $uv \in E$ is replaced by a subgraph (a gadget) in H consisting of vertices u', v', and Δ new vertices w_1, w_2, $\ldots$, w_Δ; and edges join vertices u' and v' to every one of the Δ vertices w_i.

We claim that G has a distance-1 coloring using p colors if and only if H has an acyclic coloring using p colors, where $3 \leq p \leq \Delta$.

Suppose G has a distance-1 coloring with p colors. In any edge $uv \in E$, the vertices u and v receive distinct colors. In the graph H, assign u' the same color as u, and v' the same color as v. Since each vertex w_i is adjacent only to the pair of vertices u' and v', it can be assigned any one of the $p-2$ colors distinct from the two colors that have been used for u' and v'; note that the quantity $p-2$ is positive (there exists a color for the vertex w_i) since p is assumed to be at least three. The coloring of H obtained by "processing" every edge in G in this fashion is acyclic. To see this, notice that every cycle in H contains one u' vertex, one v' vertex, and at least one w_i vertex from the gadget we created for some edge $uv \in E$, and these three vertices receive distinct colors in H. Hence every cycle in H uses at least three colors, and one direction of the claim follows.

Now consider an acyclic coloring of H with p colors, again where $3 \leq p \leq \Delta$.

We claim that the vertices u' and v' corresponding to an edge $uv \in E$ receive distinct colors. Suppose not. Then every pair of vertices w_i and w_j in this gadget would have to be colored with distinct colors, or else there would be a cycle u', w_i, v', w_j, u' which would be colored with two colors. But then this subgraph of H would need $\Delta + 1$ colors, contradicting the choice of p.

Thus in every gadget the vertices u' and v' receive distinct colors in the acyclic coloring, and these same colors can be assigned to the vertices u and v to obtain a distance-1 coloring of G using at most p colors.

There exists $\delta > 0$ such that approximating distance-1 coloring within $O(n^\delta)$ is NP-hard [5], where $n = |V|$, the number of vertices in G. Let N denote the number of vertices in H. Then $N = n + \Delta|E| = O(n + n^3)$. Thus it follows that there exists an $\epsilon > 0$ such that approximating acyclic coloring within $O(N^\epsilon)$ is NP-hard. ∎

Under stronger complexity conjectures ($NP \neq ZPP$, the class of problems with polynomial time zero-error randomized algorithms), we can prove that it is hard to approximate acyclic coloring within $O(n^{1/3-\epsilon})$, using a technique similar to that in

Agnarsson and Halldórsson [3].

We can also prove inapproximability results for star coloring, using the same gadget used in the acyclic coloring proof. We omit the details.

THEOREM 2.2. *There exists fixed $\epsilon > 0$ such that it is NP-hard to approximate the star chromatic number $\chi_s(G)$ of a graph G within $O(n^\epsilon)$.*

2.3. Related work. Some of the coloring problems introduced in section 2.1 have been studied by other authors, from an application or a pure graph-theoretic point of view. Krumke, Marathe, and Ravi [24] studied the role of distance-2 coloring in channel assignment problems in radio networks. Kumar et al. [25] have undertaken a related study on scheduling problems in wireless networks. In these two studies, several approximation and distributed algorithms for distance-2 coloring of disk and other special classes of graphs have been suggested. Balakrishnan et al. [6] showed that the problem of maximizing concurrent transmissions in certain ad-hoc wireless networks can be modeled as a maximum cardinality distance-2 matching problem, a problem related to distance-2 coloring as discussed at the end of section 2.1. The distance-2 matching problem was first studied and shown to be NP-hard by Stockmeyer and Vazirani [29].

Agnarsson and Halldórsson [2] have studied distance-k coloring of planar graphs, and a similar study of chordal graphs is available in Agnarsson, Greenlaw, and Halldórsson [1]. Acyclic colorings were first defined and studied by Grünbaum [21], who asked whether the acyclic chromatic number $\chi_a(G)$ of a graph G satisfies the inequality $\chi_a(G) \leq \Delta + 1$. A negative answer to this question was given by Erdös, who proved probabilistically the existence of graphs where $\chi_a(G) \geq \Delta^{4/3-\epsilon}$ [23]. Borodin [7] proved that every planar graph can be acyclically colored using at most five colors. Star coloring has recently been studied by Albertson et al. [4], who showed that every acyclic coloring that uses q colors can be refined to a star coloring with at most $2q^2 - q$ colors. These authors also prove that planar graphs have star colorings with at most 20 colors, and they exhibit an example in which 10 colors are required. For more pointers to theoretical works on distance-k, acyclic, and star coloring, see the monograph by Jensen and Toft [23] and our earlier work [17]. The focus of this paper is on practical, fast, suboptimal algorithms for acyclic and star coloring.

3. The new acyclic coloring algorithm.

3.1. Overview. Suppose a subset $U \subset V$ of the vertices of a graph $G = (V, E)$ has been colored satisfying the two requirements of an acyclic coloring. As discussed in section 2.1, every subgraph of G induced by a set of vertices assigned any two colors is a forest. Furthermore, every edge in G both of whose endpoints are in the set U of colored vertices belongs to a unique tree (in a two-colored forest). Therefore, the union of all possible two-colored forests induced by the vertices in the set U is a collection of edge-disjoint trees. Our algorithm maintains such a collection of trees efficiently and extends the partial coloring by processing one vertex at a time.

We use the illustration in Figure 3.1 to provide an overview of the algorithm. The graph shown at the left is the input graph G which has been partially colored, and v is the only vertex that remains to be colored. The right half of Figure 3.1 shows two-colored forests induced by the rest of the vertices of the graph. The input graph currently uses four colors, red, blue, green, and yellow (to ease black-and-white viewing, each colored vertex in the figure has been labeled with a letter reflecting the color used). Thus there could exist at most six two-colored induced forests. Out of these, the subgraph induced by green and yellow vertices is a graph whose edge set

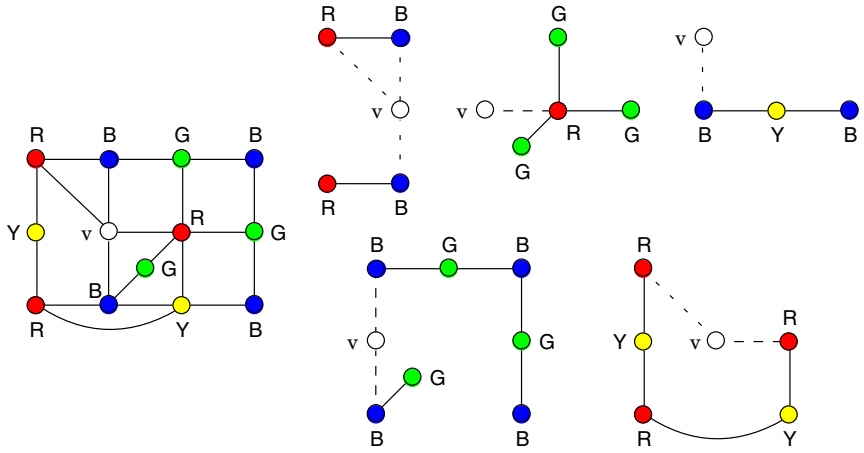

FIG. 3.1. *Left: A partial acyclic coloring of a graph; vertex v is the only vertex that remains to be colored. Every other vertex has been assigned one of the four colors red, blue, green, or yellow; the label on each vertex reflects the color used. Right: All of the nonempty, two-colored induced subgraphs; each subgraph is a forest.*

is empty. The remaining five forests are depicted in the figure. Each of the red-blue and blue-green forests consists of two trees, while each of the other forests consists of a single tree. In the example, all of the two-colored trees are incident on the vertex v.

The task of choosing a valid color for the vertex v can be divided into two natural parts. The first part involves excluding the colors of each of the distance-1 neighbors of v from its set of allowable colors. In our illustration, this results in the exclusion of the colors red and blue from the allowable set of colors for v. The second part involves ensuring that a color chosen for the vertex v does not lead to a two-colored cycle in the graph G. Since the vertex v (and, in general, any vertex in a graph) could be part of many, arbitrarily long cycles, the second part is more difficult than the first, which is confined to the distance-1 neighborhood of v in G.

The key point in our algorithm is the detection of a potential two-colored cycle that involves the vertex v in an efficient way. In particular, the cycle detection is done efficiently by examining two-colored trees incident on v, a task accomplished by visiting the distance-2 (and not any further) neighborhood of v once. Due to the distance-1 coloring requirement in an acyclic coloring, an odd cycle would necessarily use at least three colors. Thus in avoiding two-colored cycles, we will be concerned only with even cycles.

A two-colored cycle that involves the vertex v can arise only if the following two conditions are satisfied:

1. the vertex v is adjacent to *two* vertices in a two-colored tree, and
2. these adjacent vertices are of the *same* color.

Note that condition 2 restricts the concern to only even cycles, for if the vertex v is adjacent to two vertices with different colors in a two-colored tree, the corresponding cycle involving v is of odd length and thus uses three colors, posing no concern. For example, in Figure 3.1, though vertex v is adjacent to two vertices in the upper red-blue tree, v will not induce a cycle in that tree since the adjacent vertices have distinct colors.

Now let F_v denote a collection of two-colored trees that are incident on the vertex

v and satisfy the aforementioned two conditions. For each tree $T_v \in F_v$, let c_1 denote the color of the vertices adjacent to the vertex v, and let c_2 be the other color used in the tree T_v. Then, for each tree T_v, we exclude the color c_2 from the allowable set of colors for the vertex v, for otherwise v might be assigned the color c_2, creating a two-colored cycle involving v. In Figure 3.1 this process would lead to the exclusion of the color yellow from the allowable set of colors for the vertex v, for otherwise a two-colored cycle could arise in the red-yellow tree.

When both the first and the second tasks discussed above are completed for the vertex v, we know exactly the colors that cannot be assigned to v, and thus we can choose the smallest allowable color for v.

To complete the overview of the algorithm, we now need to *update* the collection of two-colored trees to reflect the fact that the vertex v has been colored. The update can be performed in two stages. In the first stage, edges vw that connect the vertex v and an already colored vertex w are grouped into two-colored *stars*. In the second stage, these stars are merged with existing two-colored trees as needed. A point that should be noted here is that the second stage of the update may cause previously disconnected trees in a two-colored forest to be connected. For example, if the vertex v in Figure 3.1 receives the color green, then the two (currently disconnected) trees in the green-blue forest need to be merged. Just as the task of choosing a color for the vertex v, we will show that the task of updating the collection of two-colored trees after v is assigned a color can be accomplished by visiting the distance-2 neighborhood of v once.

3.2. Detailed description. For an efficient implementation of the algorithm sketched in section 3.1, we use the *disjoint-set* data structure to maintain the collection of two-colored trees. For a discussion of the disjoint-set data structure, see, e.g., [15]. A set in our context corresponds to a two-colored tree, and an element of a set corresponds to an edge. The disjoint-set data structure supports three operations: MAKESET, FIND, and UNION. Operation MAKESET(e) creates a new set consisting of the element e. Operation FIND(e) returns the *representative* element of the set to which element e belongs, and operation UNION(e_1, e_2) merges the two sets $S(e_1)$ and $S(e_2)$ to which elements e_1 and e_2 belong.

Further, in our implementation of the sketched algorithm, we use the two arrays color and forbiddenColors for coloring purposes. The vertex-indexed array color is used to store the colors assigned to vertices. In particular, color$[u] = c$ indicates that the vertex u has been assigned the color c, a value that once set remains unchanged during the course of the algorithm. The color-indexed array forbiddenColors is used to mark the colors that are forbidden to a specific vertex. In particular, forbiddenColors$[c] = u$ indicates that the color c is impermissible for the vertex u; otherwise the color c is a candidate color for the vertex u. Unlike the array color, the value stored at a particular index in the array forbiddenColors varies during the course of the algorithm, an issue that will be clear in a moment.

3.2.1. The main routine. The driver routine in our acyclic coloring algorithm is outlined in Algorithm 3.1 and the current section is devoted to its discussion. The subroutines called by Algorithm 3.1 will be presented in section 3.2.2.

Leaving the initialization of various data structures in line 2 aside for a moment, the body of the outermost for-loop of Algorithm 3.1 consists of two parts. The first part (lines 4–10) is concerned with finding a color for the vertex v, the current vertex in the iteration over the vertex set V. The second part (lines 11–16) deals with updating the collection of two-colored trees incident on the vertex v.

ALGORITHM 3.1. Input: a graph G. Effect: finds an acyclic coloring of G.

```
1: procedure AcyclicColoring(G = (V, E))
2:     Initialize data structures
3:     for each v ∈ V do
4:         for each colored w ∈ N1(v) do
5:             forbiddenColors[color[w]] ← v
6:         for each colored w ∈ N1(v) do
7:             for each colored x ∈ N1(w) do
8:                 if forbiddenColors[color[x]] ≠ v then
9:                     preventCycle(v, w, x)
10:        color[v] ← min{i > 0, forbiddenColors[i] ≠ v}
11:        for each colored w ∈ N1(v) do      ▷ grow two-colored stars around the vertex v
12:            growStar(v, w)
13:        for each colored w ∈ N1(v) do
14:            for each colored x ∈ N1(w), x ≠ v do
15:                if color[x] = color[v] then
16:                    mergeTrees(v, w, x)   ▷ merge trees T1 ∋ vw and T2 ∋ wx if T1 ≠ T2
```

Finding a color. Pursuant to the first requirement of acyclic coloring, the for-loop in lines 4–5 marks the colors of the distance-1 neighbors of the vertex v as forbidden colors to v. The subsequent for-loop in lines 6–9 forbids additional colors that can result in two-colored cycles involving the vertex v, to enforce the second requirement of acyclic coloring. Here, a subset of the two-colored trees incident on the vertex v—through edges wx, where w is an already colored vertex adjacent to v and x is an already colored vertex adjacent to w—is examined. The routine preventCycle (to be discussed in section 3.2.2) checks whether the vertex v is connected to the two-colored tree containing the edge wx via two edges. If this turns out to be the case, the routine forbids the color of the vertex x from being a candidate color for the vertex v by updating the global array forbiddenColors. The if-test in line 8 ensures that a two-colored tree incident on the vertex v is investigated only if there is a potential for a two-colored cycle to arise in the same tree; recall that the upper red-blue tree in Figure 3.1 need not be examined while coloring the vertex v since it cannot lead to a two-colored cycle involving v. The if-test also avoids an unnecessary computation: once a color c in a two-colored tree T is forbidden to the vertex v to prevent a potential two-colored cycle C (that involves T and v), then the tree T need not be investigated any further in the determination of a color for the vertex v, for the existence of any other cycle C' (that also involves T and v) would only result in the exclusion of the same color c.

Once all the colors that are not allowed for the vertex v are marked in the array forbiddenColors, the array is scanned sequentially in search of the first index in which a value other than v is stored. This corresponds to the smallest allowable color for the vertex v and is chosen as the color for v (line 10). The array forbiddenColors is initialized at the beginning of the algorithm with some value outside the set V. This array can be used without being reinitialized while coloring another vertex, since in each iteration a unique vertex is colored (i.e., the identity of a vertex serves as a "time-stamp").

Updating the collection of two-colored trees. The remainder of Algorithm 3.1 is concerned with creating and updating two-colored trees incident on the vertex v. This

task is done in two stages, reflected by the two for-loops in lines 11–12 and lines 13–16. In the first stage (the first for-loop), two-colored trees consisting *only* of edges incident on the vertex v are grown. More precisely, the structures grown are two-colored stars, and the hub of each star is the vertex v. To this end, for every edge vw that connects the vertex v with an already colored vertex w, (1) a new two-colored star consisting only of the edge vw is created, and (2) if there is already a similarly two-colored star S with one or more edges of the form vw' (hence $\mathsf{color}[w] = \mathsf{color}[w']$), then the newly created star (edge vw) is merged with S. Both of these items are achieved by the call to the routine growStar in line 12; the details of this routine will be discussed in section 3.2.2.

At the end of the for-loop in lines 11–12, every edge vw connecting v and an already colored vertex w has been placed in a star in the right two-colored forest. There may now be a need for another level of merges in a forest, constituting the second stage of the data structure update. Such merges involve stars grown at the current iteration, and trees grown in iterations prior to the coloring of v. The piece of code in lines 13–16 determines whether there is a need for such merges, and if so, updates the disjoint-set data structure appropriately. In particular, for each colored vertex w adjacent to v, and each colored vertex x adjacent to w with $\mathsf{color}[x] = \mathsf{color}[v]$, the routine mergeTrees is called in line 16 with the arguments v, w, and x. The routine checks whether the tree that contains the edge vw is the same as the tree that contains the edge wx; if the two trees turn out to be different, then they are merged.

To illustrate the points discussed in the previous two paragraphs without revealing the details of the routines growStar and mergeTrees, consider the example in Figure 3.1. Suppose the vertex v has been assigned the color green. At the end of the for-loop in lines 11–12, two stars are grown around the vertex v. These are the star (path) B–v–B and the star R–v–R, where B and R stand for a blue and a red neighbor of the vertex v, respectively. In the subsequent for-loop, the star R–v–R is merged with the only tree in the current red-green forest. In the same loop, the star B–v–B is first merged with one of the two disconnected trees in the current blue-green forest, and then the resulting tree is further merged with the other tree. As a result, the blue-green forest now becomes a single tree (to be more precise, a path on eight vertices).

3.2.2. The subroutines. Let us now take an inside look at the routines preventCycle, growStar, and mergeTrees. In the discussion below, we will refer to the actions performed on the vertex v in lines 4–16 of Algorithm 3.1 as *processing vertex* v.

Avoiding two-colored cycles. The routine preventCycle (see Algorithm 3.2) takes three vertices v, w, and x as input parameters: v is the vertex currently being processed, w is a colored vertex adjacent to v, and x is a colored vertex adjacent to w. The primary task of the routine is to forbid the color of the vertex x from being chosen as a color for the vertex v, if it could lead to a two-colored cycle in the input graph. This would happen if the vertex v is incident on the two-colored tree to which the edge wx belongs via two edges. An auxiliary task of the routine is to update an associated data structure.

To determine whether or not a vertex v is connected to a two-colored tree via two edges in an efficient way, we *mark* each two-colored tree containing an edge wx incident on the vertex v. While doing so, if a tree is marked twice, then we know that the vertex v is incident on the tree via two edges. We use an edge-indexed array called $\mathsf{firstVisitToTree}$ for this purpose. An index of the array $\mathsf{firstVisitToTree}$ is the

ALGORITHM 3.2. Input: the vertex v being colored, a colored neighbor w of v, and a colored neighbor x of w. Associated data structure: $\mathsf{firstVisitToTree}[e] = (p, q)$ indicates that $\mathsf{firstVisitToTree}[e]$ was last set when processing vertex p and at that time the tree represented by the edge e was *first* visited through the edge pq. Effect: (1) updates the array firstVisitToTree, or (2) excludes $\mathsf{color}[x]$ from the allowable set of colors for v if it could otherwise lead to a two-colored cycle involving v.

```
1: procedure PREVENTCYCLE(v, w, x)
2:     e ← FIND(wx)        ▷ edge wx belongs to the 2-colored tree T represented by edge e
3:     (p, q) ← firstVisitToTree[e]
4:     if p ≠ v then                ▷ T is being visited from vertex v for the first time
5:         firstVisitToTree[e] ← (v, w)
6:     else if q ≠ w then           ▷ T is connected to vertex v via at least two edges
7:         forbiddenColors[color[x]] ← v
```

ID of the representative edge of a two-colored tree. A vertex pair (p, q) is stored at index e of the array to indicate that $\mathsf{firstVisitToTree}[e]$ was last set when processing the vertex p, and at that time the two-colored tree whose representative edge is e was *first* visited through the edge pq. In firstVisitToTree and other similar data structures used elsewhere in this paper, the word *first* refers to the context in which the neighborhood of the vertex p being processed is visited. The array firstVisitToTree is globally declared and initialized at the beginning of Algorithm 3.1 with a pair of values not corresponding to any vertex. As in the array forbiddenColors, the "marker" vertex p in the array firstVisitToTree serves as a time-stamp. Hence, this array need not be reinitialized while processing another vertex.

The routine PREVENTCYCLE begins by getting the two-colored tree T to which the edge wx belongs. Let e be the representative edge of T returned by the FIND operation. In line 3, the vertex pair (p, q) stored at index e of the array firstVisitToTree is retrieved. If $p \neq v$, then the implication is that the tree T is being visited from the vertex v for the first time. To reflect this, the vertex pair (v, w) is stored at $\mathsf{firstVisitToTree}[e]$. If, on the other hand, $p = v$, then there are two possibilities, corresponding to the cases $q = w$ and $q \neq w$. The case $q = w$ implies that the tree T has previously been visited from the vertex v, but the visit was through the same edge vw. This occurs if the vertex w is adjacent to at least two vertices (besides the vertex v) in the tree T, and does not cause the color of the vertex x to be a forbidden color for the vertex v. The second case, $q \neq w$, implies that the tree T has previously been visited from the vertex v through an edge other than vw. In this case, setting $\mathsf{color}[v] = \mathsf{color}[x]$ would create a two-colored cycle, and therefore $\mathsf{color}[x]$ is forbidden from being a color for the vertex v.

Updating the collection of two-colored trees. As discussed earlier, the first stage of updating the collection of two-colored trees incident on the vertex v being processed involves growing two-colored stars around the vertex v. This is achieved by the calls GROWSTAR(v, w) made in line 12 of Algorithm 3.1 for each colored vertex w adjacent to the vertex v.

To achieve the desired merging of edges into two-colored stars in an efficient manner, we maintain a color-indexed array, firstNeighbor, in which we store pairs of vertices. The array is used to determine whether or not a particular color is encountered for the first time while visiting the distance-1 neighborhood of a particular vertex. Specifically, we let $\mathsf{firstNeighbor}[c] = (p, q)$ to indicate that while processing the vertex p, the vertex q was the first-to-be-encountered neighbor having the color

ALGORITHM 3.3. Input: the vertex v being processed and a colored neighbor w of v. Associated data structure: $\mathsf{firstNeighbor}[c] = (p, q)$ indicates that the vertex q is the first-to-be-encountered neighbor of the vertex p having the color c. Effect: (1) creates a new two-colored tree T_{vw} consisting only of the edge vw, and (2) either updates the array firstNeighbor or merges the tree T_{vw} with a two-colored star being grown around v.

```
1: procedure GROWSTAR(v, w)
2:     MAKESET(vw)                          ▷ create a new tree T_vw consisting only of edge vw
3:     (p, q) ← firstNeighbor[color[w]]
4:     if p ≠ v then          ▷ a neighbor of v with color color[w] encountered for 1st time
5:         firstNeighbor[color[w]] ← (v, w)
6:     else                                 ▷ merge T_vw with a two-colored star being grown around v
7:         e1 ← FIND(vw); e2 ← FIND(pq)
8:         UNION(e1, e2)
```

ALGORITHM 3.4. Input: the vertex v being processed, a colored neighbor w of v, and a neighbor x of w having the same color as v. Effect: merges the tree containing the edge vw with the tree containing the edge wx, if the two trees are distinct.

```
1: procedure MERGETREES(v, w, x)
2:     e1 ← FIND(vw); e2 ← FIND(wx)
3:     if e1 ≠ e2 then
4:         UNION(e1, e2)
```

c. As the array firstVisitToTree discussed earlier, the array firstNeighbor is globally declared and initialized in Algorithm 3.1, and can be used without reinitialization while iterating through the vertex set V.

The routine GROWSTAR (Algorithm 3.3) begins by creating a two-colored tree (star) consisting only of the edge vw, a task accomplished by the call MAKESET(vw). The routine then retrieves the vertex pair (p, q) stored at the index $\mathsf{color}[w]$ of the array firstNeighbor. If $p \neq v$, then the implication is that the color used by the vertex w is encountered for the first time while visiting the vertices adjacent to the vertex v. To signify this, the vertex pair (v, w) is stored at the index $\mathsf{color}[w]$ of the array firstNeighbor. Otherwise, if $p = v$, then the implication is that a vertex q (distinct from w) that has the same color as the vertex w has already been encountered while visiting the neighbors of the vertex v. In this case, the edge vw—the two-colored star just created—is merged with the (growing) two-colored star containing the edge pq.

The second stage of the update involves merging the two-colored stars just grown with existing two-colored trees as needed. This is achieved by the calls to MERGETREES made in line 16 of Algorithm 3.1. Each such call is made with the arguments v, w, and x, where v is the vertex currently being processed, w is a colored vertex adjacent to v, and x is a colored vertex adjacent to w with $\mathsf{color}[x] = \mathsf{color}[v]$. The routine MERGETREES (Algorithm 3.4) first queries for the representative edges e_1 and e_2 of the two-colored trees T_1 and T_2 to which the edges vw and wx, respectively, belong. If $e_1 \neq e_2$, then it means that the two trees T_1 and T_2 are distinct, and therefore they need to be merged. Otherwise no further action needs to be taken.

3.3. Complexity. The outermost for-loop in Algorithm 3.1 consists of $n = |V|$ iterations. In each iteration, the two visits to the distance-2 neighborhood of the vertex v being processed, i.e., the for-loops in lines 6–9 and 13–16, constitute the dominant part of the computation. On each visited neighbor, a constant number

of operations is performed. Thus, disregarding the time spent on updating and querying the disjoint-set data structure, the time complexity of the algorithm is $O(\sum_{v \in V} d_2(v)) = O(n\overline{d}_2)$.

The disjoint-set data structure in our context contains at most m distinct elements, the edges of G. Thus there can be at most $m-1$ UNION calls and m MAKESET calls. Further, there can be at most $m' = 2n \cdot \overline{d}_1 + 3n \cdot \overline{d}_2 \leq 5n \cdot \overline{d}_2$ FIND calls, since the routine FIND is called once in the procedure PREVENTCYCLE and twice in each of the procedures GROWSTAR and MERGETREES. In our implementation of the disjoint-set data structure, we use *path compression* for the FIND operation and *union-by-size* for the UNION operation to achieve the best runtime performance. When these techniques are used, the average time spent on each such operation is at most $\alpha(m', m)$, where α is the inverse of Ackermann's function [15]. Thus the overall time complexity of our acyclic coloring algorithm is $O(n\overline{d}_2 \cdot \alpha(m', m))$. Clearly, the space complexity of the algorithm is $O(m)$.

4. The new star coloring algorithm.

4.1. Overview. Our star coloring algorithm is based on the same idea as the acyclic coloring algorithm presented in section 3. Instead of managing a collection of edge-disjoint two-colored trees, we now manage a collection of edge-disjoint two-colored stars. Recall that the *hub* of a star is the vertex at which the edges meet. The key difference between the acyclic and the star coloring algorithms stems from the following fact: An edge may be added to a tree at any vertex of the tree, whereas the only way an edge may be added to a star is at the hub of the star. A consequence of this fact is that the addition of an edge to a star does not lead to the merging of two existing stars. Hence, a simpler data structure than disjoint-set suffices. We use a one-dimensional array that maps edges to stars, and a similar array that maps stars to hubs, to manage the collection of two-colored stars.

Figure 4.1 provides an overview of the algorithm. Part (a) of the figure shows the input graph G where every vertex except v has been colored consistent with the requirements of a star coloring. The illustration uses the six colors red, blue, green, yellow, pink, and black (Bk). Parts (b) through (f) show a subset of the two-colored stars with nonempty edge sets that are incident on v.

Since a star coloring is also a distance-1 coloring, the set of colors used by vertices adjacent to the vertex v is impermissible for v. Enforcing this requirement is straightforward. In addition, any color that leads to a two-colored path involving v and three other vertices is impermissible for v. There are exactly two cases that need to be considered to identify and forbid a color of this type.

- *Case* 1. If vertex v has two adjacent vertices w and x of the *same* color, then the color of every vertex adjacent to w, and the color of every vertex adjacent to x should be excluded from the allowable set of colors for v.
- *Case* 2. If vertex v has exactly one adjacent vertex w of color c, and w belongs to a star S containing at least two edges but is not the hub of S, then the color used by the hub of S should be disallowed from being a candidate color for v.

Parts (b) and (c) of Figure 4.1 illustrate Case 1; neither the color blue nor the color green can be assigned to the vertex v. Similarly, parts (d) and (e) illustrate Case 2; the colors green and pink used by the hubs of the respective stars are not allowed for the vertex v. As illustrated in part (f), if the star S in a situation similar to Case 2 has only one edge, then it does not impose any additional color restriction on the vertex v. We say that the hub of such a star consisting of a single edge is *undefined*.

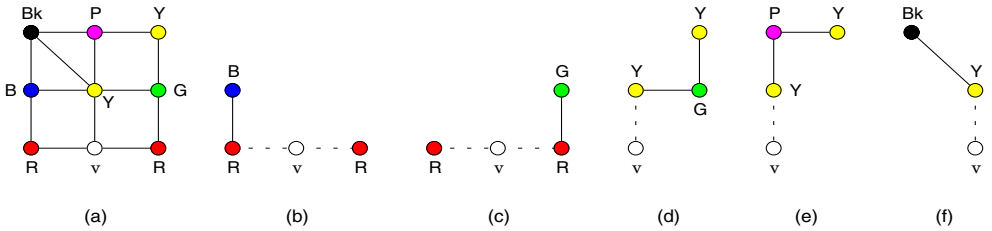

FIG. 4.1. (a): *A partial star coloring of a graph; vertex v is the only vertex that remains to be colored.* (b)–(f): *A subset of the two-colored induced subgraphs incident on the vertex v; each subgraph is a collection of stars. The illustration uses the six colors red, blue, green, yellow, pink, and black (Bk).*

The hub is defined later if and when another edge joins the star, in which case the point of incidence between the two edges becomes the hub.

Like the acyclic coloring algorithm, our star coloring algorithm iterates over the vertex set V in some order, coloring one vertex at a time. In the step where a vertex v is colored, the color used by each adjacent vertex and every additional color identified by checking Cases 1 and 2 is excluded from the allowable set of colors for v. Then the smallest permissible color is chosen and assigned to the vertex v. After this, every edge vw connecting the vertex v and an already colored vertex w is placed in the appropriate two-colored star. In particular, if an edge vw is incident on some (color$[v]$,color$[w]$)-star S, then the edge vw joins the star S; otherwise, the edge vw forms a star on its own with an undefined hub. Unlike the acyclic coloring algorithm in which the addition of an edge to a two-colored tree may result in the merging of two existing trees, the addition of an edge to a two-colored star does not lead to the merging of two existing stars.

4.2. Detailed description. To maintain the collection of two-colored stars effectively, we use an edge-indexed array, star, that maps edges to stars, and a star-indexed array, hub, that maps stars to hubs. In particular, star$[vw] = s$ signifies that the edge vw belongs to the star s, and hub$[s] = u$ indicates that the hub of the star s is the vertex u. We work with the two separate arrays, star and hub, in order to distinguish between stars consisting of at least two edges (having defined hubs) and stars consisting of a single edge (having undefined hubs). In addition, we make use of the two arrays color and forbiddenColors in just the same way as in section 3.

Our star coloring algorithm is outlined in Algorithm 4.1. The body of the outer for-loop has two parts. The first part, the piece of code in lines 4–16, is concerned with finding a color for the vertex v, the current vertex in the iteration over the vertex set. The second part deals with updating the collection of two-colored stars incident on the vertex v, and is achieved by the call to UPDATESTARS in line 17. Hereafter, we shall refer to the actions performed on the vertex v in lines 4–17 as *processing the vertex v*.

Finding a color. The for-loop in lines 4–15 of Algorithm 4.1 determines the colors that cannot be assigned to the vertex v. In line 5, the color used by a vertex w adjacent to the vertex v is marked as a forbidden color for v, since a star coloring is also a distance-1 coloring. To determine and forbid additional colors that lead to two-colored paths involving v and three other vertices, Cases 1 and 2 outlined earlier need to be checked. To detect Case 1 in an efficient manner, we use the color-indexed array firstNeighbor which stores pairs of vertices: firstNeighbor$[c] = (p, q)$ indicates

ALGORITHM 4.1. Input: a graph G. Associated data structure: $\mathsf{firstNeighbor}[c] = (p, q)$ indicates that the vertex q is the first-to-be-encountered neighbor of the vertex p having the color c; $\mathsf{treated}[y] = x$ indicates that the color of every vertex adjacent to the vertex y is already forbidden from being a color for the vertex x. Effect: finds a star coloring of G.

```
1:  procedure StarColoring(G = (V, E))
2:      Initialize data structures
3:      for each v ∈ V do
4:          for each colored w ∈ N1(v) do
5:              forbiddenColors[color[w]] ← v
6:              (p, q) ← firstNeighbor[color[w]]
7:              if p = v then                                   ▷ Case 1
8:                  if treated[q] ≠ v then
9:                      TREAT(v, q)                  ▷ forbid colors of neighbors of q
10:                     TREAT(v, w)                  ▷ forbid colors of neighbors of w
11:             else
12:                 firstNeighbor[color[w]] ← (v, w)
13:                 for each colored x ∈ N1(w), x ≠ v do
14:                     if x = hub[star[wx]] then                ▷ potential Case 2
15:                         forbiddenColors[color[x]] ← v
16:     color[v] ← min{i > 0, forbiddenColors[i] ≠ v}
17:     updateStars(v)
```

ALGORITHM 4.2. Input: the vertex v being processed and a colored vertex w adjacent to v. Effect: excludes the colors used by vertices adjacent to w from the allowable set of colors for v, and updates the array treated to reflect this.

```
1:  procedure TREAT(v, w)
2:      for each colored x ∈ N1(w) do
3:          forbiddenColors[color[x]] ← v
4:      treated[w] ← v
```

that when processing the vertex p, the vertex q was the first neighbor with color c to be encountered. (An array with the same name was used for a similar purpose in the acyclic coloring algorithm discussed in the previous section.) In line 6, a lookup in the table firstNeighbor at the index color$[w]$ is made and the stored value is retrieved in the vertex pair (p, q).

If $p = v$, then the implication is that a vertex which is distinct from, but has the same color as, the vertex w—the vertex q—has already been encountered while visiting the distance-1 neighbors of the vertex v. This clearly corresponds to Case 1. The appropriate action to be taken is to forbid the colors used by the vertices adjacent to q and the colors used by the vertices adjacent to w. The calls to the subroutine TREAT (outlined in Algorithm 4.2) in lines 9 and 10 do precisely that. In connection with this case, we use the vertex-indexed array treated in which we store vertices; treated$[y] = x$ indicates that the color of every vertex adjacent to the vertex y has already been marked as a forbidden color for the vertex x. The if-test in line 8 is performed to avoid unnecessary computation in the case where color$[w]$ is encountered for a third time or later.

If, on the other hand, $p \neq v$, then the implication is that the color used by the vertex w is encountered for the first time while visiting the distance-1 neighborhood

ALGORITHM 4.3. Input: the vertex v being processed. Effect: adds edges vw, where w is colored, to appropriate two-colored stars.

```
1:  procedure updateStars(v)
2:      for each colored w ∈ N1(v) do
3:          if ∃x ∈ N1(w) where x ≠ v and color[x] = color[v] then      ▷ vw, wx ∈ E
4:              hub[star[wx]] ← w                                       ▷ this may already be true
5:              star[vw] ← star[wx]
6:          else
7:              (p, q) ← firstNeighbor[color[w]]
8:              if (p = v) and (q ≠ w) then                             ▷ vw, vq ∈ E ∧ color[w] = color[q]
9:                  hub[star[vq]] ← v                                   ▷ this may already be true
10:                 star[vw] ← star[vq]
11:             else                                                    ▷ vw forms a new star
12:                 starID ← starID + 1
13:                 star[vw] ← starID
```

of the vertex v. This information is recorded in line 12 for future use. The piece of code in the subsequent for-loop takes a proactive measure in anticipation of Case 2. In particular, the loop iterates over edges wx that belong to existing two-colored stars. For each star to which an edge wx belongs, if the star has a defined hub and the hub is the vertex x, then the color of x is forbidden from being a color for the vertex v (line 14). Potentially, this corresponds to Case 2; if the vertex v is later found to have a neighbor vertex $w' \neq w$ with the same color as w, then it reduces to Case 1. Recall that only stars containing at least two edges have defined hubs; for a star with only one edge the test in line 14 will fail and therefore will not impose an additional color restriction (cf. Figure 4.1 (f)).

At the end of the for-loop in lines 4–15, every color that is not allowed to be a color for the vertex v is marked in the array forbiddenColors. The smallest permissible color is then chosen and assigned to the vertex v in line 16.

As with the arrays used in section 3, in the arrays firstNeighbor and treated the marker vertex p serves as a time-stamp; it is used to determine whether the information stored in the data structures concerns the vertex currently being processed. These data structures are initialized at the beginning of Algorithm 4.1, and no reinitialization is required in the course of iteration over the vertex set.

Updating the collection of two-colored stars. The call to the routine updateStars (outlined in Algorithm 4.3) is made at line 17 to update the collection of two-colored stars to reflect the fact that the vertex v has been assigned a color. In Algorithm 4.3, each edge vw that joins the vertex v with an already colored vertex w is placed in an existing or a new two-colored star depending on the colors of the vertex v and its distance-2 neighbors. The edge vw joins an existing star if (1) there exists an edge wx and color$[v]$ = color$[x]$, or (2) there exists an edge vq and color$[w]$ = color$[q]$. The first case can be detected by looking at the distance-1 neighborhood of the vertex w; for the second case, the information in the array firstNeighbor, which is set in Algorithm 4.1, is utilized. Clearly, in the first case the vertex w becomes the hub of the resulting star (if it was not already so), and in the second case the vertex v becomes the hub (again, if it was not already so). If neither of cases (1) or (2) turns out to be positive, then the edge vw forms a new two-colored star on its own with an undefined hub. Notice that the definitions of the hubs in lines 4 and 9 are necessary, since the edges wx and

vq could be single-edge-stars with undefined hubs at the time the routine is called.

4.3. Complexity. Given a graph on n vertices and m edges, the computational work involved in each of the n iterations of the outer for-loop in Algorithm 4.1 is proportional to $d_2(v)$. Hence the overall time complexity of our star coloring algorithm is $O(n\bar{d}_2)$. Its space complexity is clearly $O(m)$.

5. Coloring models in Hessian computation. We have experimentally compared the new algorithms for acyclic and star coloring presented in this paper with previously suggested algorithms for these and other related coloring problems that occur in the computation of sparse Hessians. In this section, we briefly review such coloring models and discuss their interrelationships. In section 6, we will discuss corresponding greedy algorithms.

One step in an efficient computation of a sparse derivative matrix A using automatic differentiation (or finite differencing) is to use structural information about A to partition its n columns into p disjoint groups, with p as small as possible. This step can be posed as a problem of seeking an $n \times p$ *seed* matrix S whose (j, k) entry is defined as follows:

$$s_{jk} = \begin{cases} 1 & \text{if column } a_j \text{ belongs to group } k, \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

Since the matrix S specified by (5.1) corresponds to a partitioning of the columns of the matrix A, in every row r of the matrix S there exists exactly one column c in which the entry s_{rc} is equal to one. In the automatic differentiation literature, there exist approaches that use a seed matrix where a row-sum is not necessarily equal to one [19]. Such approaches are not partitioning-based as they allow a column of A to be placed in more than one group. In this paper we consider only seed matrices that define a partition.

The specific criteria used to define a seed matrix S depend on whether the entries of the matrix A are to be recovered from the *compressed* representation AS *directly* (by conceptually solving a diagonal system of equations) or via *substitution* (by conceptually solving a set of triangular systems of equations). The criteria also depend on whether the derivative matrix A is Jacobian (nonsymmetric) or Hessian (symmetric). In general, a seed matrix S suitable for a direct method requires a larger p compared to one suitable for a substitution method. On the other hand, the recovery of entries via substitution incurs an additional, but affordable, computational cost.

5.1. Models for direct methods. Curtis, Powell, and Reid [16] observed that a *structurally orthogonal* partition of a Jacobian matrix A—a partition of the columns of A in which no two columns in a group share a nonzero at the same row index—gives a seed matrix S where the entries of A can be directly recovered from the compressed representation AS. Obviously, a structurally orthogonal partition could also be used in the context of computing a Hessian if one were to settle for not exploiting the available symmetry. Following this approach, McCormick [27] showed that a structurally orthogonal partition of a Hessian is equivalent to a *distance*-2 *coloring* of its adjacency graph.

The *adjacency graph* $G(A)$ of a Hessian A has a vertex for each column, and an edge joins column vertices a_i and a_j whenever the entry a_{ij}, $i \neq j$, is nonzero; the diagonal entries in A are assumed to be nonzero, and they are not explicitly represented by edges in $G(A)$. In the distance-2 coloring model for structurally orthogonal

partition as well as in other models discussed in this section, a set of vertices having the same color corresponds to a set of columns that belong to the same group in the induced partition. In other words, each column of the seed matrix corresponds to a color class.

Powell and Toint [28] were the first to suggest symmetry-exploiting partitions for both direct and substitution-based methods for computing Hessians. When translated to a coloring ϕ of the adjacency graph, the partition Powell and Toint suggested for a direct Hessian computation requires that (1) ϕ be a distance-1 coloring, and (2) in every path on three vertices $P_3 = v, w, x$, the vertices v and x receive different colors whenever $\phi(w) > \min\{\phi(v), \phi(x)\}$. We call a coloring that satisfies these two requirements a *restricted star coloring*. Notice that, unlike in a distance-2 coloring, in a restricted star coloring, the terminal vertices v and x in a path v, w, x are allowed to have the same color as long as the color of the middle vertex w is lower in value.

Coleman and Moré [13] generalized the approach of Powell and Toint and showed that a symmetrically orthogonal partition of a Hessian suffices for a direct recovery of its entries from a compressed representation. A partition of the columns of a Hessian matrix A is *symmetrically orthogonal* if for every nonzero element a_{ij}, either (1) the group containing column a_j has no other column with a nonzero in row i, or (2) the group containing column a_i has no other column with a nonzero in row j. Coleman and Moré established that a symmetrically orthogonal partition of a Hessian is equivalent to a *star coloring* of its adjacency graph.

5.2. Models for substitution methods. Based on the work of Powell and Toint [28], Coleman and Moré [13] found a coloring model for a partitioning of a Hessian matrix A that defines a seed matrix S such that the nonzero entries of A can be recovered from the compressed representation AS via a (restricted) substitution method. The recovery of the entries of the matrix A from the matrix AS requires solving a triangular system of equations, a system that heavily relies on the structure of A. Appropriately, Coleman and Moré called the model triangular coloring. Given a graph $G = (V, E)$, a mapping $\phi : V \to \{1, 2, \ldots, p\}$ is a *triangular coloring* if there exists a vertex ordering π such that (1) ϕ is a distance-1 coloring, and (2) in every $P_3 = v, w, x$, the vertices v and x receive different colors whenever $\pi(w) > \max\{\pi(v), \pi(x)\}$. A vertex ordering π that is well suited for triangular coloring exists and will be discussed in section 5.3.

Triangular coloring exploits symmetry only to a limited extent. This limitation was later addressed by Coleman and Cai [9], who introduced a generalized notion of substitutable partitions for which they provided a simple graph-theoretic characterization. In particular, they proved that an *acyclic coloring* of the adjacency graph of a Hessian induces a substitutable partition of its columns. They also showed that, using an acyclic coloring, the nonzeros of the Hessian can be recovered from its compressed representation by considering *a pair of color classes* (groups of columns) at a time. The key difference between a partition induced by a triangular coloring and a partition induced by an acyclic coloring is the following. Triangular coloring induces a partition that defines one large, coupled triangular system of equations, whereas acyclic coloring induces a partition that defines several smaller, decoupled triangular systems of equations, each defined by a pair of color classes.

Table 5.1 summarizes the coloring models in Hessian computation discussed thus far. In the table, the coloring variants are listed in an increasing order of restrictiveness, with the least constrained variant at the top and the most constrained variant at the bottom. In the following subsection, we present results that make the interre-

TABLE 5.1

*Overview of coloring models used in Hessian computation. In each case, the adjacency graph $G = (V, E)$ of the underlying Hessian is used, and the mapping $\phi : V \rightarrow \{1, 2, \ldots, p\}$ defines a coloring. In addition to the input graph G, the definition of triangular coloring assumes the existence of a vertex ordering π. *Powell and Toint [28] stated these models using their equivalent matrix partitioning counterparts; the translations into graph colorings are due to Coleman and Moré [13].*

Name	Coloring conditions	Computation modeled
Acyclic	1. $vw \in E \Rightarrow \phi(v) \neq \phi(w)$	substitution
	2. every cycle uses ≥ 3 colors	Coleman and Cai [9]
Triangular	1. $vw \in E \Rightarrow \phi(v) \neq \phi(w)$	restricted substitution
	2. in every $P_3 = v, w, x$:	Powell and Toint [28]*
	$\pi(w) > \max\{\pi(v), \pi(x)\} \Rightarrow \phi(v) \neq \phi(x)$	Coleman and Moré [13]
Star	1. $vw \in E \Rightarrow \phi(v) \neq \phi(w)$	direct
	2. every P_4 uses ≥ 3 colors	Coleman and Moré [13]
Restricted	1. $vw \in E \Rightarrow \phi(v) \neq \phi(w)$	restricted direct
star	2. in every $P_3 = v, w, x$:	Powell and Toint [28]*
	$\phi(w) > \min\{\phi(v), \phi(x)\} \Rightarrow \phi(v) \neq \phi(x)$	Coleman and Moré [13]
Distance-2	1. $vw \in E \Rightarrow \phi(v) \neq \phi(w)$	direct (ignores symm.)
	2. $P_3 = v, w, x \in G \Rightarrow \phi(v) \neq \phi(x)$	McCormick [27]

lationships among these variations more precise.

5.3. Interrelationships. Recall that we denote specialized chromatic numbers by χ with an appropriate subscript. For example, $\chi_a(G)$ denotes the acyclic chromatic number of G. In Lemma 5.1, we relate the acyclic, star, and restricted star chromatic numbers of a graph. Similarly, in Lemma 5.2 we show the relationship among the acyclic, triangular, and restricted star chromatic numbers of a graph.

LEMMA 5.1. *For every graph G, $\chi_a(G) \leq \chi_s(G) \leq \chi_{rs}(G)$.*

Proof. Let ϕ be a star coloring of a graph G. Observe that a cycle in G on three vertices uses three colors, since ϕ is also a distance-1 coloring of G. Moreover, a cycle on at least four vertices contains a P_4 and therefore uses at least three colors. Hence ϕ is an acyclic coloring of G, and the first inequality follows.

For the second inequality, we show that if ϕ is a restricted star coloring of G, then ϕ is also a star coloring of G. Let $P = v, w, x$ be a P_3 in G. Since ϕ is a restricted star coloring, $\phi(v) \neq \phi(w)$ and $\phi(w) \neq \phi(x)$. Furthermore, either $\phi(v) \neq \phi(x)$ or $\phi(v) = \phi(x) > \phi(w)$ holds. In the former case, in any P_4 in G that contains the path P, adjacent vertices have different colors, and the P_4 uses at least three colors, satisfying the requirements of a star coloring. To see that the latter case also fulfills the requirements of a star coloring, consider any $P_4 = v, w, x, y$ in G that extends the path P in one of its ends. (A similar argument would hold for a path u, v, w, x that extends P in the other end.) Again, since ϕ is a restricted star coloring, $\phi(y) \neq \phi(x)$. Moreover, the relationship $\phi(y) \neq \phi(w)$ holds, since otherwise the relationship $\phi(x) > \phi(w) = \phi(y)$ in the path w, x, y would hold, contradicting the fact that ϕ is a restricted star coloring. Hence, in the path v, w, x, y, adjacent vertices have different colors, and at least three colors are used. Therefore ϕ is a star coloring of G. ∎

LEMMA 5.2. *For every graph G, $\chi_a(G) \leq \chi_t(G) \leq \chi_{rs}(G)$.*

Proof. Suppose ϕ is a triangular coloring of a graph G for a vertex ordering π. We show that ϕ is then an acyclic coloring of G. Consider any cycle C in G, and let

z be the vertex in C with the *highest* index in the ordering π. Since C is a cycle, there exists a path $P = u, z, v$ in C for some vertices u and v. Further, since ϕ is a triangular coloring, $\phi(u) \neq \phi(v)$ by choice of z. Since ϕ is also a distance-1 coloring, both of these colors are distinct from $\phi(z)$. Hence, the cycle C uses at least three colors, and therefore ϕ is an acyclic coloring of G.

We now show the second inequality. Suppose ϕ is a restricted star coloring of a graph G on n vertices. Consider a vertex ordering $\pi = v_1, v_2, \ldots, v_n$ such that the sequence $\{\phi(v_i)\}$ is nondecreasing. In other words, the ordering π is such that vertices in the same color class are listed consecutively, and the color classes in turn are listed in increasing order. Consider any path $P = v, w, x$ on three vertices in G. Since ϕ is a restricted star coloring, a pair of adjacent vertices in the path P have different colors. There are two possibilities regarding the terminal vertices v and x in P: either $\phi(v) \neq \phi(x)$ or $\phi(v) = \phi(x) > \phi(w)$. In the former case, the path P trivially satisfies the requirements of a triangular coloring. Given the ordering π as defined in the proof, the path P in the latter case also satisfies the requirements of a triangular coloring, since $\pi(w) < \min\{\pi(v), \pi(x)\}$. Hence, the coloring ϕ and the ordering π together define a triangular coloring of G. □

The inequalities in Lemmas 5.1 and 5.2 can easily be extended in both ends. Clearly, $\chi_1(G) \leq \chi_a(G)$ holds, since an acyclic coloring is also a distance-1 coloring. In a distance-2 colored graph, the vertices in every P_3 use three distinct colors. Hence, a distance-2 coloring is also a restricted star coloring, implying the relationship $\chi_{rs}(G) \leq \chi_2(G)$. We summarize these relationships in Theorem 5.3.

THEOREM 5.3. *For every graph G, the following relationships hold:*

$$\chi_1(G) \leq \chi_a(G) \leq \chi_t(G) \leq \chi_{rs}(G) \leq \chi_2(G) = \chi_1(G^2),$$

$$\chi_1(G) \leq \chi_a(G) \leq \chi_s(G) \leq \chi_{rs}(G) \leq \chi_2(G) = \chi_1(G^2).$$

The relationship between $\chi_t(G)$ and $\chi_s(G)$ is not clear. There are examples of star colorings where there does not exist a vertex ordering that makes a given star coloring a triangular coloring. An example would be a star coloring of a graph in which every vertex is the hub of a two-colored P_3. Conversely, there are examples of colorings that are triangular but not star. A simple example is a two-colored P_4 in which adjacent vertices have different colors. Thus the class $C_s(G)$ of star colorings of a graph G and the class $C_t(G)$ of triangular colorings of G are such that neither $C_t(G) \subseteq C_s(G)$ nor $C_s(G) \subseteq C_t(G)$ holds. However, the relationship $\chi_t(G) \leq \chi_s(G)$ might still hold. We leave settling the latter issue as an open problem.

The last equality in Theorem 5.3 follows from the fact that a distance-2 coloring of G is equivalent to a distance-1 coloring of the square graph G^2. In fact, most of the coloring variants on a graph $G = (V, E)$ listed in Table 5.1 can also be viewed as a distance-1 coloring of an appropriately defined "filled" graph $G' = (V, E \cup F)$.

When the original graph G is the adjacency graph of a Hessian, the filled graphs in the distance-2 and triangular coloring cases have interesting interpretations. The filled graph in the distance-2 coloring case (the square graph G^2) is the *column intersection graph* of the underlying Hessian A. The column intersection graph of a matrix has a vertex for each column, and an edge between a pair of columns exists whenever the two columns share nonzero entries at some common row index. In the case of a triangular coloring for a given vertex ordering π, the filled graph G', in which each fill edge f is generated as per the specification

$$\forall P_3 = v, w, x \text{ in } G : \pi(w) > \max\{\pi(v), \pi(x)\} \Rightarrow f = vx, \tag{5.2}$$

is the column intersection graph of the *lower triangular part* L_π of the permuted matrix P^TAP, where P is a permutation matrix defined by the ordering π. These relationships were observed and used by Coleman and Moré [13], who showed that a *smallest last* (SL) vertex ordering as a choice for π is particularly well suited for triangular coloring.

Given a graph on n vertices, a vertex ordering $v_1, v_2, \ldots, v_n$ is an SL ordering if for all $1 \leq i \leq n$, the vertex v_i is a vertex with the *smallest* degree in the graph induced by vertices $v_1, \ldots, v_i$. Here is why an SL ordering is well suited for triangular coloring: Among the $n!$ possible orderings (permutations) π, an SL ordering minimizes the maximum number of nonzeros in a row in L_π. Thus, an SL ordering provides a lower bound on the triangular chromatic number, a result proven by Coleman and Moré [13].

Orthogonal to its role in triangular coloring, an SL ordering is also known to be one of the effective ordering techniques for reducing the number of colors used by a greedy coloring algorithm. The number of colors used by a greedy distance-1 coloring algorithm that employs an SL ordering to color a graph G is an important graph parameter. The parameter is known as the *coloring number* $col(G)$ of G and is closely related to several other graph notions, including degeneracy, maximal core, and arboricity [17]. In our context, since an SL ordering can be computed in linear time in the number of edges, the coloring number is an easy-to-compute, nontrivial upperbound for the distance-1 chromatic number. As mentioned in the previous paragraph, the coloring number is also a lower bound for the triangular chromatic number. Theorem 5.4 summarizes these relationships.

THEOREM 5.4. *For every graph G, $\chi_1(G) \leq col(G) \leq \chi_t(G)$.*

6. Previous algorithms. Coleman and Cai [9] accurately modeled the partitioning problem that occurs in substitution-based Hessian computation as the acyclic coloring problem. However, the algorithm they suggested is an algorithm for triangular coloring. Specifically, they suggested that a filled graph G' be explicitly constructed as described by the expression (5.2), using an SL vertex ordering for π, and then that G' be distance-1 colored. The same approach has been taken earlier by Coleman and Moré [13] and in the related software papers [10, 11]. However, as we will soon show in the current section, it is possible to achieve a triangular coloring of G by using a greedy algorithm that works directly on G. This section also discusses previously known greedy algorithms for star and distance-k coloring. All of the algorithms discussed in this section are included in our experiments.

In a previous work [17], we suggested an algorithm for star coloring that enforces the coloring requirements in a straightforward manner—by checking paths on four vertices. We refer to this algorithm here as NAIVESTARCOLORING (*NS*). Powell and Toint [28] suggested an algorithm of the same spirit, formulated in the language of matrices, for the restricted star coloring problem. We will call this algorithm here RESTRICTEDSTARCOLORING (*RS*).

Algorithms *NS* and *RS* are both greedy. In determining a set of forbidden colors for a vertex v, algorithm *NS* consults the colors used by the distance-3 neighbors of v, whereas algorithm *RS* checks the colors used by only the distance-2 neighbors. Once the set of forbidden colors is obtained, both algorithms choose the smallest color not included in the set as the color for the vertex v. In a similar fashion, one can design greedy algorithms for distance-1 coloring, distance-2 coloring, and triangular coloring for a given ordering π. We refer to such algorithms as DISTANCE1COLORING ($D1$), DISTANCE2COLORING ($D2$), and TRIANGULARCOLORING (T-π). The suffix in the

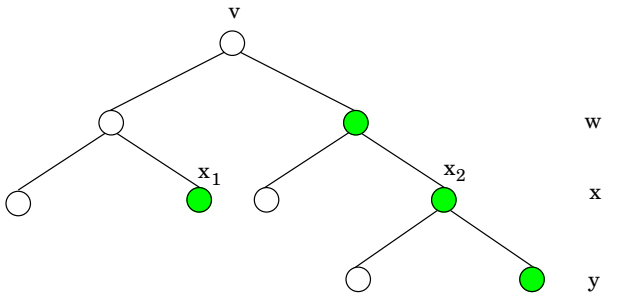

FIG. 6.1. *Visualizing a step in greedy coloring algorithms; v is the vertex currently being colored. Table 6.1 shows how the set of forbidden colors for v is determined in various cases.*

TABLE 6.1
Conditions under which a particular color is forbidden to v (see in conjunction with Figure 6.1). Here ϕ is a coloring, and π is a vertex ordering.

	w	x_1	x_2
$D1$	always		
$D2$	always	always	always
NS	always	always	only if $\phi(w) = \phi(y)$
RS	always	always	only if $\phi(w) > \phi(x)$
T-π	always	only if $\pi(w) > \max\{\pi(v), \pi(x)\}$	only if $\pi(w) > \max\{\pi(v), \pi(x)\}$

notation T-π signifies that a specific vertex ordering π is used by the algorithm.

We use Figure 6.1 in conjunction with Table 6.1 to illustrate how the set of forbidden colors for a vertex is determined by the various greedy algorithms in coloring a graph G on n vertices and m edges. The tree depicted in Figure 6.1 is not a subgraph of G, but rather a visualization of the classification of the neighborhood $N_k(v)$ of the vertex v being colored at the current step of the particular algorithm. The classification is based on whether a vertex in the set $N_k(v)$ is already colored or not. In the figure, the vertices of the graph G that are one, two, and three edges away from the vertex v are represented by the tree nodes at the levels w, x, and y, respectively. A shaded *node* signifies already colored *vertices* and an unshaded node signifies vertices not yet colored. The shaded nodes at level x are further labeled x_1 and x_2, to distinguish between the cases where a vertex w in a path v, w, x is already colored or not.

Table 6.1 summarizes the conditions under which a color used by a vertex in the scenario described in Figure 6.1 is forbidden to the vertex v for the various algorithms. Cases x_1 and x_2 correspond to distinct decisions in algorithms *NS* and *RS*. These two cases are indistinguishable in algorithms $D2$ and T-π and are irrelevant to algorithm $D1$. As column x_1 in Table 6.1 shows, in algorithms *NS* and *RS*, the color of vertex x in a path v, w, x where vertex w is not yet colored is immediately forbidden to the vertex v. This ensures that any extension v, w, x, y of the path v, w, x would end up using at least three colors, as desired, since in the step in which the vertex w is colored, the distance-1 coloring requirement guarantees that the vertex w gets a color distinct from the colors of vertices v and x. In contrast, in the case where vertex w is already colored (see column x_2 in the table), determining whether the color of vertex x should be forbidden to vertex v or not requires a further check. In algorithm *NS*, the check involves vertices w and y (note that y is three edges away from v). In

algorithm *RS*, the check involves vertices w and x. In each of the algorithms listed in Table 6.1, once the set of forbidden colors to the vertex v is determined, the smallest color not included in the set is chosen as the color for v.

The illustration should make it clear that the complexity of algorithm *D*1 is $O(n\bar{d}_1) = O(m)$, the complexity of each of algorithms *D*2, *RS*, and *T*-π is $O(n\bar{d}_2)$, and the complexity of algorithm *NS* is $O(n\bar{d}_3)$.

7. Experimental results.

7.1. Setup. To demonstrate the performance of our new algorithms, we ran experiments on 29 graphs obtained from molecular dynamics applications [8, 30]. The test graphs were chosen because of their large size and irregular structure. The left half of Table 7.1 shows the number of vertices $|V|$, the number of edges $|E|$, the maximum degree Δ, and the average degree $\bar{d}_1$ in each test graph. The right half of the table shows certain computed values that we will discuss shortly. All of the algorithms used in our experiments were implemented in C++. The hardware used was a Linux machine equipped with Dual AMD Opteron Processor 244, 1.8 GHz, 2 GB memory, and 1 GB cache. The operating system was RHEL 3 and the compiler was GCC.

7.2. Algorithms compared and orderings used. In addition to the new acyclic and star coloring algorithms presented in sections 3 and 4—abbreviated henceforth as *A* and *S*—in this section, we report experimental results on the algorithms *D*1, *D*2, *RS*, *NS*, and *T*-*sl* discussed in section 6. The suffix *sl* in the triangular coloring algorithm indicates that we used an SL vertex ordering as the ordering π to define the coloring, since this is optimal as discussed in the paragraph leading to Theorem 5.4. Algorithms *D*1 and *D*2 are included in our experiments primarily to serve as references against which quantities could be normalized.

The order in which vertices are colored in a greedy algorithm determines the number of colors used by the algorithm. As mentioned earlier, an SL ordering (more precisely, a distance-1 SL ordering) is one of the effective known ordering techniques for distance-1 coloring. As a natural extension, for algorithms such as *D*2, where the distance-2 neighborhood of a vertex is explored at each step, a distance-2 SL ordering (D2SL) can be defined and used to attain a similar benefit. Specifically, we define a D2SL ordering using the quantity $|N_2(v)|$ associated with each vertex v in a graph. Using linear-time sorting techniques, a D2SL ordering can be computed within the same order of time as a subsequent distance-2 coloring, i.e., in $O(n\bar{d}_2)$ time.

In our experiments (results to be presented in section 7.3), for each of the algorithms *D*1, *D*2, *RS*, *NS*, *S*, and *A*, we used one of the orderings among natural (the order in which vertices appear in the input graph), D1SL, and D2SL orderings that led to the fewest colors on a majority of the test cases. For algorithm *D*1, this ordering was D1SL; for algorithms *D*2, *RS*, *NS*, and *S*, this was the D2SL ordering; and for algorithm *A*, it was the natural ordering. We have also experimented with appropriately extended variants of *incidence degree* and *largest degree first* orderings [17]. The results we obtained using these are in general similar to or worse than the ones obtained using the SL orderings and hence are not reported here.

In a greedy algorithm for triangular coloring, vertex orderings arise in two different contexts: (1) as part of the definition of the coloring, and (2) as part of the coloring algorithm. In terms of an ordering (π) that defines the coloring, a D1SL ordering in the input graph G is optimal. The same ordering can be used while coloring the vertices sequentially, and that is what we did in algorithm *T*-*sl*. To (potentially) reduce the number of colors used, one could also use a D1SL coloring order based on

TABLE 7.1

*Test graph statistics (left half) and computed values (right half). Column $col(G)$ lists the coloring number of each graph, and columns D1 and D2 list the runtime in seconds of algorithms D1 and D2. The last column shows the ratio of number of edges in a derived graph G' to that in an original G; the graph G' is obtained from the graph G as per the specification in expression (5.2) where π is an SL vertex ordering in G. The symbol *** indicates that a graph G', being too large to fit in memory, could not be computed.*

ID	Name	$\|V\|$	$\|E\|$	Δ	$\overline{d}_1$	$col(G)$	Runtime (sec)		$\frac{\|E'\|}{\|E\|}$
							D1	D2	
1	er-gre-2	36,573	53,046	8	3	4	0.01	4.44	1.1
2	apoa1-2	92,224	139,351	8	3	4	0.00	31.74	1.1
3	HIV-3	11,414	64,134	20	11	8	0.00	0.44	1.6
4	er-gre-3	36,573	169,787	19	9	8	0.00	4.34	1.7
5	popc-br-3	24,916	102,058	22	8	9	0.01	2.05	1.6
6	apoa1-3	92,224	456,220	20	10	9	0.02	26.85	1.7
7	mol1-2	131,072	1,179,648	18	18	11	0.08	65.47	2.8
8	HIV-4	11,414	130,332	39	23	14	0.00	0.58	2.5
9	apoa1-4	92,224	1,131,436	43	25	15	0.04	28.39	3
10	er-gre-4	36,573	451,355	42	25	15	0.01	4.91	3
11	popc-br-4	24,916	255,047	43	20	15	0.01	2.35	2.7
12	popc-br-5	24,916	497,269	75	40	24	0.01	3.16	4.2
13	apoa1-5	92,224	2,254,805	73	49	25	0.08	33.02	5.2
14	HIV-6	11,414	412,623	116	72	34	0.01	1.74	6
15	er-gre-6	36,573	1,482,904	116	81	36	0.04	9.76	7.8
16	popc-br-6	24,916	850,043	125	68	38	0.03	4.89	6.4
17	apoa1-6	92,224	3,864,429	123	84	38	0.11	42.14	8.1
18	mol1-3	131,072	5,636,096	86	86	42	0.27	123.59	9.8
19	popc-br-7	24,916	1,316,771	192	106	54	0.04	8.39	9
20	popc-br-8	24,916	1,914,476	275	154	72	0.05	14.67	12.4
21	popc-br-9	24,916	2,668,555	380	214	98	0.06	25.78	16.4
22	er-gre-10	36,573	6,511,122	460	356	124	0.16	92.92	30
23	popc-br-10	24,916	3,587,724	514	288	126	0.08	43.83	21
24	mol1-4	131,072	14,680,064	224	224	104	0.59	354.34	***
25	HIV-10	11,414	1,655,383	454	290	115	0.03	18.29	***
26	apoa1-10	92,224	17,100,850	503	371	135	0.39	305.52	***
27	HIV-12	11,414	2,683,956	760	470	183	0.05	47.78	***
28	er-gre-12	36,573	10,928,521	775	598	198	0.24	275.53	***
29	popc-br-12	24,916	5,924,137	855	476	199	0.14	116.7	***
tot	(1–23)	1,139,699	35,129,231						
tot	(1–29)	1,447,312	88,102,142						

vertex degrees in the filled graph G' derived from G according to the specification given by expression (5.2).

In a method they call *slsl*, Coleman and Moré [13] *explicitly* construct such a filled graph G' using a D1SL ordering in G as π, and then distance-1 color the graph G' using a D1SL ordering in G' to obtain a triangular coloring of the original graph G. We have implemented this method as well; here we call it *T-slsl* to reflect that it is a triangular coloring of the original graph. The last column of Table 7.1 shows the ratio of the number of edges in the derived graph G' to that in the original graph G. For 6 out of the 29 test graphs we used (the last six in Table 7.1), the graph G' could not be computed as it was too big to fit in the size of the available memory. The asterisks used in the corresponding entries in the last column of Table 7.1 indicate this.

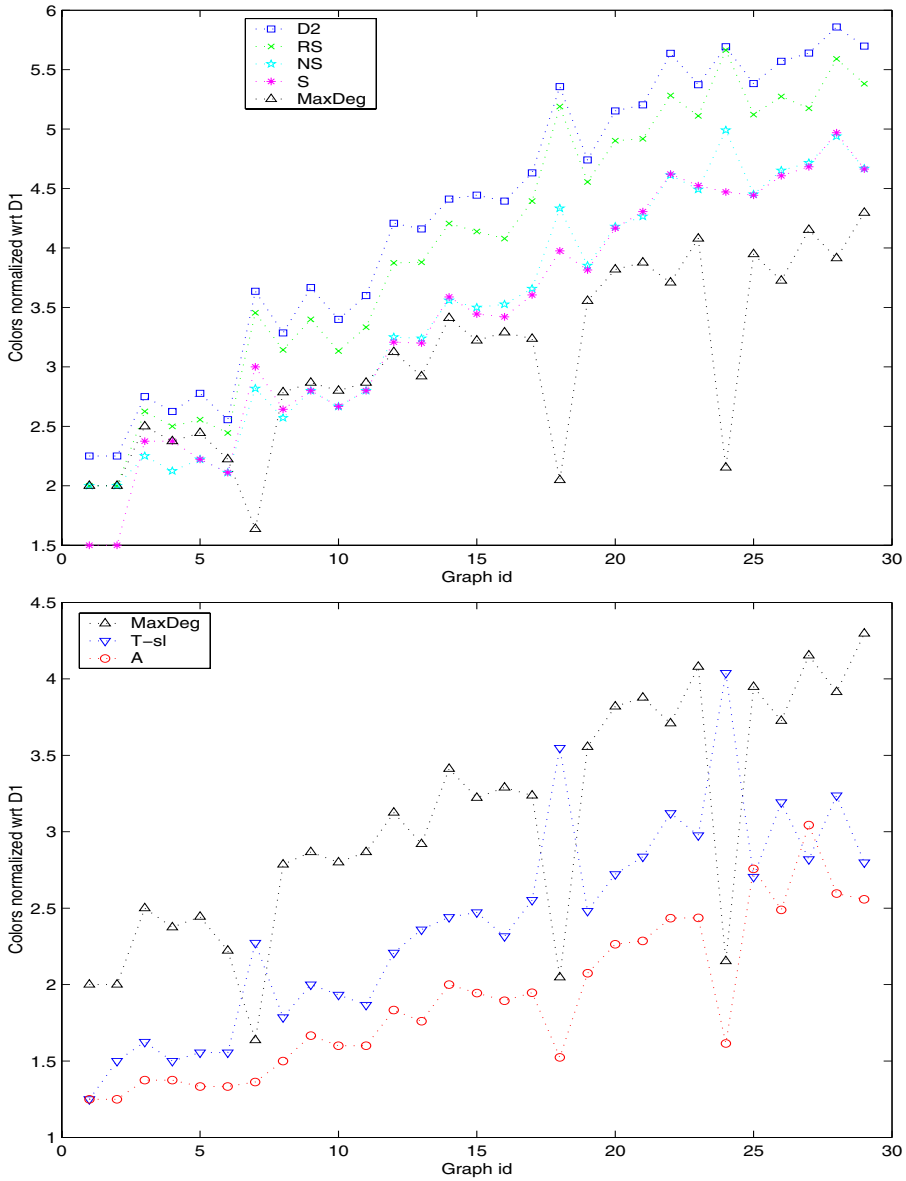

FIG. 7.1. *Normalized number of colors used by various greedy algorithms. In algorithm T-sl, a D1SL ordering of the vertices in G is used to both define and perform the coloring. Algorithm D1 uses a D1SL ordering for coloring. In algorithms D2, RS, NS, and S, vertices are colored in a D2SL ordering in G; in algorithm A, vertices are colored in the natural order they appear in G. The plots are normalized relative to the number of colors used by algorithm D1, i.e., the coloring numbers of the graphs. These numbers are listed in Table* 7.1.

7.3. Results. We will now describe a few figures and tables that show our experimental results. The figures will be discussed collectively in section 7.4.

The upper graph of Figure 7.1 displays plots of the number of colors used by algorithms *D*2, *RS*, *NS*, and *S* normalized relative to the number of colors used by algorithm *D*1, the coloring numbers of the graphs. The bottom graph shows analogous

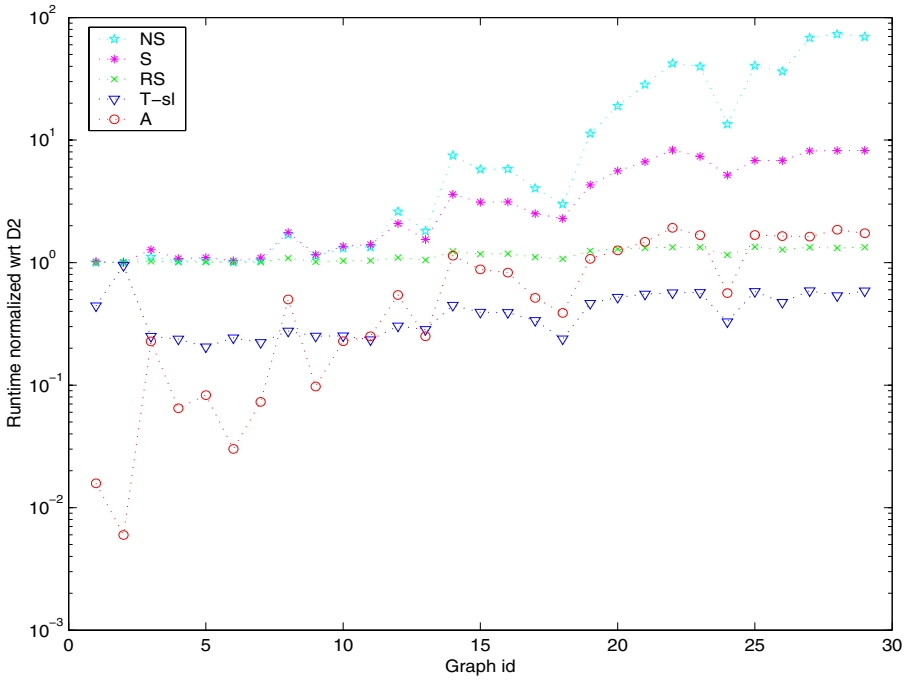

FIG. 7.2. *Execution time comparison. Vertices are colored in the orderings specified in Figure* 7.1. *The plots, in semilog scale, are normalized relative to the runtime of Algorithm D*2*, values listed under column D*2 *of Table* 7.1.

plots for algorithms *T-sl* and *A*. The common curve in the two parts is the plot of the maximum degrees in the graphs, again normalized relative to the coloring numbers. The coloring numbers of the graphs, sorted in increasing order, are listed under column $col(G)$ in Table 7.1.

Figure 7.2 shows semilog plots of the runtimes of algorithms *RS*, *NS*, *S*, *T-sl*, and *A* normalized relative to the runtime of algorithm $D2$, the values of which are listed in the column labeled $D2$ in Table 7.1. Column $D1$ in the same table shows the runtimes of algorithm $D1$. In these two columns as well as in execution times reported elsewhere in this section, the reported times include time spent on ordering, when applicable, but not time for reading test graph files from disk.

Figure 7.3 shows a comparison between algorithms *A* and *T-slsl*. The results reported in the figure concern the test graphs on which the latter could be run successfully, the first 23 of the 29 test graphs used.

Table 7.2 shows a summary of the results for the various algorithms; the upper row gives data summed over all test cases, and the lower row shows data summed over test cases on which algorithm *T-slsl* was successfully run.

To show the reduction in number of colors that algorithms $D1$, $D2$, *RS*, *NS*, and *S* were able to get through their use of D1SL or D2SL orderings, we ran these algorithms with natural vertex ordering as well. Table 7.3 summarizes the results obtained. In addition to absolute figures, the table shows the savings in number of colors (as a percentage) and the relative time cost in comparison with a coloring that uses an appropriate SL ordering (data listed in Table 7.2).

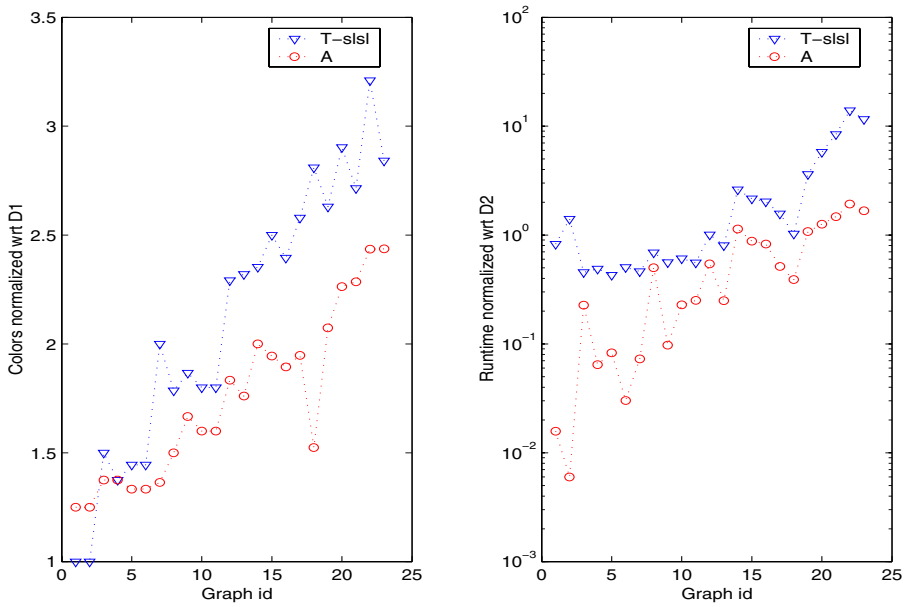

FIG. 7.3. *Comparison of normalized number of colors (left) and normalized runtimes (right) between algorithms A and T-slsl. The number of colors are normalized relative to that used by algorithm D1, and the runtimes are normalized relative to that of algorithm D2.*

TABLE 7.2

Top row: Total number of colors and runtime summed over all test graphs. Bottom row: Totals over test graphs on which algorithm T-slsl could be run successfully (i.e., graphs without asterisks in Table 7.1). The order in which vertices are colored is as specified in Figure 7.1.

	D2	*RS*	*NS*	*S*	*T-sl*	*T-slsl*	*A*	*D1*
Colors	9,240	8,749	7,636	7,558	5,065		4,110	1,757
Runtime (min)	28.2	34.4	930	162	12.4		32.5	0.04
Colors	3,951	3,734	3,207	3,191	2,189	2,149	1,709	823
Runtime (min)	9.6	11	128	34.5	3.8	41.8	7	0.02

TABLE 7.3

Top row: Total number of colors and runtime summed over all test graphs, when vertices are colored in their natural order in an input graph (Nat). Bottom row: Color savings and relative time cost in comparison with coloring using an appropriate SL ordering (numbers listed in Table 7.2).

	D2	*RS*	*NS*	*S*	*D1*
Colors	11,429	11,275	9,973	9,470	2,384
Runtime (min)	4.32	9.60	679	126	0.03
SL vs. Nat colors saved (%)	19	22	23	20	26
SL to Nat time ratio	6.5	3.6	1.4	1.3	1.3

7.4. Discussion. A reader can make several observations from Figures 7.1–7.3 and Tables 7.1–7.3. We highlight the following few points to supplement such observations in the context of Hessian computation.

- The advantage of exploiting sparsity using coloring cannot be overstated. If all of the Hessians (with the nonzero structures of the test graphs) were to be computed without exploiting sparsity or symmetry, the time required would be proportional to 1.5 million steps (the sum total of the vertices in Table 7.1).

If the strongest coloring model considered here—acyclic coloring—were to be used, all of these Hessians could be evaluated in about four thousand time steps (the number of colors, or compressed columns, in the A column at the top half of Table 7.2). Moreover, in general, coloring constitutes only a small fraction (typically far less than 10%) of the overall computation time involved in obtaining numerical values in a derivative matrix [18, 19]. Note that the total coloring time required to acyclic-color *all* of the test graphs, totaling over 88 million edges, is about 30 minutes.

- As the upper part of Figure 7.1 shows, among the three algorithms *RS*, *NS*, and S for direct Hessian computation where symmetry is exploited, algorithm S uses the fewest colors. Algorithm *RS* uses slightly fewer colors than the symmetry-ignoring algorithm $D2$. The difference in the number of colors used by algorithms *RS* and S is significant, whereas that between *NS* and S is marginal.
- The maximum degree Δ in a graph is a lower bound for the distance-2 chromatic number; this fact is clearly reflected in Figure 7.1. One can also see that for a majority of our test graphs, algorithms *RS*, *NS*, and S used more than Δ colors. However, there were cases, noticeably among the relatively sparse graphs of ID less than 10, where these algorithms used fewer than Δ colors.
- The more important advantage of algorithm S over algorithm *NS* is its lower time complexity, a fact supported by the nearly sixfold reduction in observed running times, as Figure 7.2 and Table 7.2 attest. Algorithms S and *RS* have the same theoretical complexity, and, in practice, algorithm S is observed to be slower by a small factor.
- As the bottom graph of Figure 7.1 illustrates, the coloring algorithms for Hessian computation via substitution—algorithms *T-sl* and A—use many fewer colors than the coloring algorithms for direct Hessian computation—algorithms *RS*, *NS*, and *S*. On the test graphs we used, our acyclic coloring algorithm A consistently used fewer colors than the triangular coloring algorithm *T-sl*. Moreover, in *all* of our test cases, algorithm A used fewer than Δ colors, while for algorithm *T-sl* the statement holds true for a majority, but not all, of the test cases. Interestingly, the exceptions in the latter case are regular graphs (graphs 7, 18, and 24 in Table 7.1). Note that the reference against which the number of colors in Figures 7.1 and 7.3 are normalized is a lower bound for triangular coloring.
- As Figure 7.3 shows, our acyclic coloring algorithm A also used fewer colors than algorithm *T-slsl*. On the subset of the test cases on which algorithm *T-slsl* could be run successfully, algorithm A used 20% fewer colors than *T-slsl*. In terms of runtime, algorithms A and *T-sl* are comparable, whereas *T-slsl* is much slower.
- In comparison with a natural ordering, an appropriate SL vertex (re)ordering for coloring reduced the number of colors used by algorithms $D1$, $D2$, *RS*, *NS*, and S by roughly 20% (see Table 7.3). In contrast, the numbers listed in Table 7.2 show that the savings in number of colors rendered by algorithm *T-slsl* compared to algorithm *T-sl* is only 1.8%, obtained at a cost of runtime that is larger by a factor of 11. The acyclic coloring algorithm A did not benefit from an SL ordering.
- Disregarding time for computing vertex orderings, the practical runtimes of algorithms $D2$, *RS*, S, *T-sl*, and A are expected to be similar since the time

complexity of each of these algorithms is nearly $O(n\bar{d}_2)$. In the observed runtimes in Figure 7.2, the plots for *T-sl* and *A* for the most part lie below the line $y = 1$, which corresponds to the runtime of algorithm *D*2. This is mainly due to the variation in the orderings used: for algorithm *D*2 (as well as *RS* and *S*) a D2SL ordering in G was used, for algorithm *T-sl* a D1SL in G was used, and for algorithm *A* the natural ordering was used.

- Even when vertices were colored in natural order, the acyclic coloring algorithm *A* was observed to be slightly faster than the star coloring algorithm *S*. On the other hand, time complexity analyses of the two algorithms show *A* to be slightly slower than *S*. The most likely reason for *S* being slower than *A* in practice is that there are many more two-colored stars to be maintained in algorithm *S* than there are two-colored trees to be maintained in algorithm *A*, which in turn makes the former use more memory operations, and therefore more runtime.

8. Conclusion. We have presented new algorithms for star and acyclic coloring, models for efficient Hessian computation via direct and substitution methods. We experimentally demonstrated that the new algorithms outperform previously known methods. Our acyclic coloring algorithm runs faster, uses fewer colors, and requires less memory and storage space than the *T-slsl* triangular coloring method of Coleman and Moré [13]. In comparison with our implementation of the *T-sl* triangular coloring algorithm—which unlike the original algorithm of Coleman and Moré works directly on an input graph G—the new acyclic coloring algorithm produces fewer colors at comparable runtime. Our new star coloring algorithm runs faster and uses fewer colors than our previous algorithm for the same problem [17]. In comparison with the previously known restricted star coloring algorithm, our new star coloring algorithm yields a significant reduction in number of colors at the cost of a slight increase in observed runtime.

Using an acyclic coloring, Coleman and Cai [9] showed that the nonzero entries of a Hessian can be evaluated via substitution by considering a pair of color classes at a time. Our new acyclic coloring algorithm has such structures (two-colored trees) in place, which facilitates an efficient implementation of a procedure for recovering a Hessian from a compressed representation. We have implemented such a routine. In collaboration with Andrea Walther at the Technical University of Dresden, our coloring and Hessian recovery codes have been incorporated in the automatic differentiation tool ADOL-C [20]. A paper that demonstrates the efficacy of the new star and acyclic coloring algorithms in the overall process of computing the Hessian of a given function using automatic differentiation has been submitted for publication elsewhere [18]. The results we obtained in our experiments using ADOL-C show that the new coloring algorithms render huge savings in overall runtime, and that Hessian computation via substitution using acyclic coloring is faster than a direct computation using star coloring, considering the overall process.

The underlying idea in the new algorithms, the exploitation of two-colored structures, can be extended to *bicoloring* models in efficient computation of Jacobians [14]. We are currently working in this direction.

We conclude this paper by pointing out a few other directions for further work.

- As discussed in the paragraph immediately after Theorem 5.3, the relationship between the triangular and star chromatic numbers of a graph is not clear. Prove or disprove the claim $\chi_t(G) \leq \chi_s(G)$.

- The size of a largest clique in a graph is an obvious lower bound for the distance-1 chromatic number $\chi_1(G)$. Analogously, $\Delta + 1$ is a lower bound for the distance-2 chromatic number $\chi_2(G)$, since there exists a clique of that size in the square graph G^2. Finding nontrivial lower bounds for acyclic and star chromatic numbers is a worthwhile issue.
- Appropriate extensions of orderings such as *smallest last*, *largest first*, and *incidence degree* helped reduce the number of colors used by our star coloring algorithm, but not that of the acyclic coloring algorithm. It would be interesting to find orderings suitable for the latter algorithm.

Acknowledgments. We thank Jorge Moré for suggesting a comparison between acyclic and triangular coloring and for his helpful comments on a draft of this paper. We also thank Paul Hovland for making the test graphs available to us. Our sincere thanks go to the anonymous referees for their valuable comments on the content and presentation of this paper and for bringing related works in the literature to our attention.

REFERENCES

[1] G. Agnarsson, R. Greenlaw, and M. M. Halldórsson, *On powers of chordal graphs and their colorings*, Congr. Numer., 100 (2000), pp. 41–65.
[2] G. Agnarsson and M. M. Halldórsson, *Coloring powers of planar graphs*, SIAM J. Discrete Math., 16 (2003), pp. 651–662.
[3] G. Agnarsson and M. M. Halldórsson, *Coloring powers of planar graphs*, in Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2000, pp. 654–662.
[4] M. O. Albertson, G. G. Chappell, H. A. Kierstead, A. Kündgen, and R. Ramamurthi, *Coloring with no 2-colored P_4's*, Electron. J. Combin., 11 (2004), article R26.
[5] S. Arora and C. Lund, *Hardness of approximations*, in Approximation Algorithms for NP-Hard Problems, D. S. Hochbaum, ed., PWS Publishing, Boston, MA, 1997, Chap. 10, pp. 399–446.
[6] H. Balakrishnan, C. L. Barrett, V. S. A. Kumar, M. V. Marathe, and S. Thite, *The distance-2 matching problem and its application to the MAC-layer capacity of ad hoc wireless networks*, IEEE J. Selected Areas in Commun., 22 (2004), pp. 1069–1079.
[7] O. V. Borodin, *On acyclic colorings of planar graphs*, Discrete Math., 25 (1979), pp. 211–236.
[8] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus, *CHARMM: A program for macromolecular energy, minimization, and dynamics calculations*, J. Comput. Chem., 4 (1983), pp. 187–217.
[9] T. F. Coleman and J.-Y. Cai, *The cyclic coloring problem and estimation of sparse Hessian matrices*, SIAM J. Algebraic Discrete Methods, 7 (1986), pp. 221–235.
[10] T. F. Coleman, B. Garbow, and J. J. Moré, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Softw., 10 (1984), pp. 329–347.
[11] T. F. Coleman, B. Garbow, and J. J. Moré, *Software for estimating sparse Hessian matrices*, ACM Trans. Math. Softw., 11 (1985), pp. 363–377.
[12] T. F. Coleman and J. J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 20 (1983), pp. 187–209.
[13] T. F. Coleman and J. J. Moré, *Estimation of sparse Hessian matrices and graph coloring problems*, Math. Program., 28 (1984), pp. 243–270.
[14] T. F. Coleman and A. Verma, *The efficient computation of sparse Jacobian matrices using automatic differentiation*, SIAM J. Sci. Comput., 19 (1998), pp. 1210–1233.
[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., MIT Press, Cambridge, MA, 2001.
[16] A. R. Curtis, M. J. D. Powell, and J. K. Reid, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl., 13 (1974), pp. 117–119.
[17] A. H. Gebremedhin, F. Manne, and A. Pothen, *What color is your Jacobian? Graph coloring for computing derivatives*, SIAM Rev., 47 (2005), pp. 629–705.
[18] A. H. Gebremedhin, A. Pothen, A. Tarafdar, and A. Walther, *Efficient computation of sparse Hessians: An experimental study using ADOL-C*, submitted, 2006.

[19] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, Philadelphia, 2000.
[20] A. Griewank, D. Juedes, and J. Utke, *ADOL-C: A package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Softw., 22 (1996), pp. 131–167.
[21] B. Grünbaum, *Acyclic colorings of planar graphs*, Israel J. Math., 14 (1973), pp. 390–408.
[22] S. Hossain and T. Steihaug, *Computing a sparse Jacobian matrix by rows and columns*, Optim. Methods Softw., 10 (1998), pp. 33–48.
[23] T. Jensen and B. Toft, *Graph Coloring Problems*, Wiley-Interscience, New York, 1995.
[24] S. O. Krumke, M. V. Marathe, and S. S. Ravi, *Models and approximation algorithms for channel assignment in radio networks*, Wireless Networks, 7 (2001), pp. 567 – 574.
[25] V. S. A. Kumar, M. V. Marathe, S. Parthasarathy, and A. Srinivasan, *End-to-end packet-scheduling in wireless ad-hoc networks*, in Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 2004, pp. 1014–1023.
[26] Y.-L. Lin and S. S. Skiena, *Algorithms for square roots of graphs*, SIAM J. Discrete Math., 8 (1995), pp. 99–118.
[27] S. T. McCormick, *Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem*, Math. Program., 26 (1983), pp. 153–171.
[28] M. J. D. Powell and P. L. Toint, *On the estimation of sparse Hessian matrices*, SIAM J. Numer. Anal., 16 (1979), pp. 1060–1074.
[29] L. Stockmeyer and V. Vazirani, *NP-completeness of some generalizations of the maximum matching problem*, Inform. Process. Lett., 15 (1982), pp. 14–19.
[30] M. M. Strout and P. D. Hovland, *Metrics and models for reordering transformations*, in Proceedings of the 2004 Workshop on Memory System Performance, ACM, New York, 2004, pp. 23–34.