We present a suite of fast and effective algorithms, encapsulated in a software package called ColPack, for a variety of graph coloring and related problems. Many of the coloring problems model partitioning needs arising in compression-based computation of Jacobian and Hessian matrices using Algorithmic Differentiation. Several of the coloring problems also find important applications in many areas outside derivative computation, including frequency assignment in wireless networks, scheduling, facility location, and concurrency discovery and data movement operations in parallel and distributed computing. The presentation in this article includes a high-level description of the various coloring algorithms within a common design framework, a detailed treatment of the theory and efficient implementation of known as well as new vertex ordering techniques upon which the coloring algorithms rely, a discussion of the package’s software design, and an illustration of its usage. The article also includes an extensive experimental study of the major algorithms in the package using real-world as well as synthetically generated graphs.

1. INTRODUCTION AND OVERVIEW

In many mathematical or computational contexts, one encounters the need for partitioning a set of binary-related objects into as few subsets of independent objects as possible so that some “scarce” resource is optimally used. Graph coloring in a generic sense is an abstraction for such a partitioning. It comes in many variations depending on how the notion of “inde-
pendence” is defined, and it finds applications in a wide variety of areas. In algorithms for numerical optimization and solution of differential equations, coloring is used to model matrix partitioning needs in the efficient computation of sparse Jacobian and Hessian matrices [Coleman and More 1983; 1984; Gebremedhin et al. 2005]. In preconditioned iterative methods for sparse linear systems, graph coloring—via an approach known in the literature as “multi-coloring”—is used to maximize exploitable parallelism [Jones and Plassmann 1994; Saad 1996]. Coloring is used as a “probing” method to approximately and quickly compute Schur complements, as is done for example in [Siefert and Sturler 2006] in the context of saddle-point problems. On emerging heterogenous architectures, coloring is used to decompose computation into tasks that can be mapped to different processing units for concurrent execution [DeVito et al. 2011]. In compilers, coloring is used for register allocation [Chaitin et al. 1981]. In high-performance computing, coloring is used, among others, to improve cache performance [Hall et al. 2001]. Channel assignment problems in radio and wireless networks are modeled using different kinds of coloring problems [Krumke et al. 2001].

The sample application areas just mentioned reveal that there is a strong practical motivation for solving graph coloring problems in the context of larger computations. Although there is a vast literature around coloring, much of it is theoretical in nature focusing primarily on computational complexity and other mathematical aspects. Lesser attention has been given to the development, analysis and deployment (via software tools) of effective algorithms for coloring problems that are of high practical significance.

We present, in this paper, a general-purpose software package, called ColPack, consisting of fast and effective algorithms for a variety of graph coloring—and associated—problems arising in scientific computing and other applications. The primary target application of the package is the computation of sparse Jacobian and Hessian matrices using Algorithmic (also known as Automatic) Differentiation (AD), where the compression-recovery paradigm is applied to attain runtime and storage efficiency.

The paper has a twofold purpose:

(1) describe the package ColPack, and
(2) use the package to conduct algorithmic research.

Concerning the first, we provide a high-level description of the major algorithms available in the package, discuss the package’s software design, and illustrate the package’s usage. Regarding the second purpose, we do an extensive experimental study of the major algorithms in the package in terms of their efficacy in solving the coloring problems.

1.1. Coloring problems supported

Table I gives a complete list of the coloring problems for which algorithms are implemented in ColPack. The list is divided in three categories.

The first category consists of coloring problems defined on a general (nonbipartite) graph $G = (V, E)$. Six coloring problems are listed under that category. Star and acyclic coloring are used, among others, to accurately model matrix partitioning requirements in Hessian computation. Restricted star and triangular coloring are weaker counterparts, in a sense that will become clear in section 3, in the same context. Distance-1 and distance-2 coloring are used in many of the contexts mentioned in the first paragraph of this section.

The second and third categories in Table I consist of coloring problems defined on a bipartite graph $G_b = (V_1, V_2, E)$. These are used to model matrix partitioning problems in unidirectional and bidirectional Jacobian computation, respectively.

All of the problems listed in Table I are formally defined and the scenarios in derivative computation under which they arise are briefly discussed in Section 3. We review the basics of AD and the compression-recovery scheme we employ for sparse derivative computation in Section 2.
Table I. List of coloring problems supported by ColPack.

<table>
<thead>
<tr>
<th>General Graph $G = (V, E)$</th>
<th>Bipartite Graph $G_b = (V_1, V_2, E)$: Bicoloring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance-1 coloring</td>
<td>Partial distance-2 coloring on $V_2$</td>
</tr>
<tr>
<td>Distance-2 coloring</td>
<td>Partial distance-2 coloring on $V_1$</td>
</tr>
<tr>
<td>Star coloring</td>
<td>Star bicoloring</td>
</tr>
<tr>
<td>Acyclic coloring</td>
<td>Restricted star coloring</td>
</tr>
<tr>
<td>Triangular coloring</td>
<td></td>
</tr>
</tbody>
</table>

1.2. Coloring algorithms

The objective in every one of the problems listed in Table I is to minimize the number of colors used. Each problem is known to be NP-hard to solve optimally and most are hard to even approximate [Gebremedhin et al. 2005; Gebremedhin et al. 2007]. For example, Zuckerman [2007] showed that for all $\epsilon > 0$, it is NP-hard to approximate distance-1 coloring to within $n^{1-\epsilon}$, where $n$ is the number of vertices in the graph. The various coloring algorithms in ColPack are fast, and yet effective, heuristics; we can provide a posteriori lower bounds on the number of colors needed for particular problems, and thus show that the computed colorings are often close to optimal. In each algorithm vertices are colored sequentially one at a time, in some carefully chosen order, and the color assigned to a vertex is never changed. In this sense, the algorithms are greedy. Whenever necessary, the algorithms maintain and take advantage of two-colored induced subgraphs. All of the algorithms are designed within a common framework. We discuss the framework and provide a high-level description of the different algorithms in Section 4. A detailed presentation and analysis of the algorithms for star and acyclic coloring is available in [Gebremedhin et al. 2007].

1.3. Ordering techniques

The order in which vertices are processed in a greedy (sequential) heuristic determines the number of colors used by the heuristic. ColPack contains implementations of various ordering techniques for each of the coloring problems it supports. The ordering methods are classified as degree-based and coloring-based. Three of the degree-based ordering techniques available in ColPack are: Largest First [Welsh and Powell 1967], Smallest Last [Matula 1968], and Incidence Degree [Coleman and Moré 1983]. In Section 5, we provide a fresh characterization for these previously-known (in the context of distance-1 coloring) ordering techniques, and introduce a new ordering technique called Dynamic Largest First. In addition, we extend the four techniques such that they suit the coloring problems that require visits to distance-2 neighborhoods of vertices and the coloring problems that are defined on bipartite graphs.

The new characterization of the ordering techniques uses only the back- and forward-degrees. This makes it possible to decouple the ordering procedure from the coloring algorithm that uses it. The decoupling helps achieve more modular software design and efficient implementation. These advantages have been exploited in the construction of ColPack. Of the four ordering techniques, SL is of special interest in that it optimizes a mathematically well defined notion. As a result, it has interesting connections with several other graph theoretic notions. A discussion of these relationships is included in Section 5. The coloring-based ordering techniques take as input an initial coloring and output an ordering determined by the coloring. The idea is motivated by the Iterated Greedy algorithm of Culberson [1992].

1.4. Other functionalities

In Section 6.1, we briefly discuss the other major functionality of ColPack, the set of algorithms for recovering the nonzero entries of a Jacobian or a Hessian from a compressed
representation. For a detailed treatment of the recovery algorithms for Hessian computation, see [Gebremedhin et al. 2009].

1.5. Software design

ColPack is written in C++ in an object-oriented manner with the goal of being modular and extensible. We describe its software design in Section 7, and provide sample codes that illustrate its usage in the Appendix.

1.6. Experimental evaluation

Section 8 is devoted to the second purpose of this paper—conduct research using ColPack. Specifically, in that section, we present a comprehensive experimental study of the various ordering techniques in the context of greedy algorithms for the coloring problems listed in Table I. We use as a testbed real-world graphs drawn from various application areas as well as synthetic graphs representing several graph classes. Our findings show that SL (and ID) ordering are in general effective for distance-1, distance-2, star and acyclic coloring, while DLF is effective for coloring of certain random graphs and for star bicoloring of bipartite graphs. Moreover, in the case of distance-1 coloring, for the graphs we considered, we observed that the ordering techniques SL (and ID) outperform a variant of the Saturation Degree ordering technique [Brézal 1979] we implemented, both in terms of number of colors used and runtime. Further, by computing appropriate lower bounds on optimal values for the various coloring problems, we show that the colorings obtained via SL ordering are often extremely close to optimal! Finally, the execution time of the various algorithms is observed to be low in general and agrees well with complexity analyses.

1.7. Code and test set availability

ColPack is publicly available for download (via the GNU General Public License) at www.cs.purdue.edu/homes/apothen/software. A short documentation describing the major interface functions in the package is available as a PDF file at the same website. The real-world test graphs used in the experiments are available via the University of Florida Sparse Matrix Collection [Davis and Hu 2011], and the synthetic test graphs can be reproduced using the software tools mentioned in Section 8.

1.8. Related work

Coleman, Garbow and Moré did the pioneering work on software for sparse Jacobian and Hessian computation [Coleman et al. 1984; 1985]. Their software, written in Fortran, aimed at estimating derivatives using finite differences (Automatic Differentiation tools enable exact derivative evaluation, but they were not as mature in the mid 80’s as they are today). Hasan, Hossain and Steihaug have presented in a recent workshop their work on software for sparse Jacobian determination via a direct method [Hasan et al. 2009].

A survey and synthesis of the use of coloring in derivative computation is available in [Gebremedhin et al. 2005], where the bipartite graph-based partial distance-2 coloring model for Jacobian computation was also introduced. ColPack has been interfaced with ADOL-C, an operator overloading-based AD tool for the differentiation of functions written in C or C++ [Griewank et al. 1996; Walther and Griewank 2012]. ColPack has also been interfaced with the source-to-source transformation AD tool ADIC2 [Narayanan et al. 2010].

2. BACKGROUND ON DERIVATIVE COMPUTATION

We review in this section the basics of AD and describe the framework we use for sparse derivative computation. For a comprehensive discussion of AD we refer the reader to the books [Griewank and Walther 2008] and [Naumann 2012].
Algorithm 1 A framework for sparse derivative computation.

```
procedure SparseCompute(F : R^n → R^m or f : R^n → R)
  Step 1: Determine the sparsity structure of the matrix A = ∇F (or A = ∇^2f).
  Step 2: Using a specialized vertex coloring on an appropriate graph representation of
          the matrix A, obtain an n-by-p seed matrix S that defines a partitioning of the columns of A into p
          groups with p as small as possible.
  Step 3: Compute the compressed matrix B = AS.
  Step 4: Recover the numerical values of the entries of A from B.
end procedure
```

2.1. Basics of Automatic Differentiation

AD provides exact derivative information about a smooth function F : R^n → R^m, x → F(x) = y, given as a computer program in a high-level language. It does so by breaking down the computation of F into a sequence of elementary evaluations upon which the chain rule of calculus is applied systematically. Depending on the direction in which derivatives are propagated along the chain of steps in the decomposed evaluation, one can identify two basic modes of AD, the forward and the reverse. The forward mode propagates derivatives from independent to dependent variables, and the reverse mode propagates derivatives from dependent to independent variables.

One of the attractive features of AD is that the cost of derivative evaluation can be analytically estimated in terms of the operation count OPS(F) involved in evaluating the function F to be differentiated. Using the forward mode of AD, the product of the Jacobian \( ∇F \in R^{m×n} \) and a seed vector \( s \), corresponding to a direction in \( R^n \), can be computed with an operation count of no more than five times \( OPS(F) \) [Griewank and Walther 2008]. Similarly, the product of a seed vector and the Jacobian \( ∇F \) can be obtained using the reverse mode in its basic form at a cost of no more than five times \( OPS(F) \), independent of the number of input variables [Griewank and Walther 2008]. Given a scalar-valued function \( f : R^n → R \), the product of the Hessian \( ∇^2f \in R^{n×n} \) and a seed vector can be obtained at a cost of no more than ten times \( OPS(f) \) by using second-order adjoint mode, which is obtained by combining the forward and the reverse mode [Griewank and Walther 2008].

2.2. Sparse derivative computation: the framework

The results mentioned in the previous paragraph imply that by using an n-by-n identity matrix as a seed matrix \( S \), a Jacobian \( A = ∇F \in R^{m×n} \) (or Hessian \( A = ∇^2f \in R^{n×n} \)) can be determined using AD (or finite differences) as the product \( AS \) at a cost that is proportional to the number of columns \( n \). Whenever the desired derivative matrix \( A \) is sparse, however, the cost can be made proportional to a drastically smaller value \( p \) by choosing a seed matrix \( S \in \{0,1\}^{n×p} \) that enables a group of columns \( A \) to be evaluated together (as a sum), instead of one column at a time.

Procedure SparseCompute, outlined in Algorithm 1, formalizes the steps involved in an efficient computation of a sparse derivative matrix in such a fashion. The seed matrix \( S \in \{0,1\}^{n×p} \) determined in Step 2 is such that its \((j,k)\) entry is one if the \( j \)th column of the matrix \( A \) belongs to group \( k \) and zero otherwise. In this paper we consider only those cases in which \( S \) defines a column partitioning, that is, the sum in every row of \( S \) is exactly one. (Approaches in which a row-sum in \( S \) is not necessarily equal to one exist [Griewank and Walther 2008], but will not be considered here.)

The sparsity structure of the derivative matrix \( A \) is predicted in Step 1 by an AD tool.

The task of finding a suitable seed matrix in Step 2 is modeled as a coloring problem on a suitable graph associated with \( A \) and is solved using methods developed independently of the AD tool. Similarly, once the compressed matrix \( B = AS \) is computed in Step 3 by an AD tool, the entries of \( A \) can be recovered from \( B \) in Step 4 (using the compression information...
Table II. Overview of graph coloring models in computation of derivative matrices. The Jacobian is represented by its bipartite graph, and the Hessian by its adjacency graph. NA stands for not applicable.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>unidirectional partition</th>
<th>bidirectional partition</th>
<th>Recovery</th>
</tr>
</thead>
</table>
| Jacobian | partial distance-
| star bicoloring | Direct | |
| Hessian | star coloring | NA | Direct | |
| Jacobian | NA | acyclic bicoloring | Substitution | |
| Hessian | acyclic coloring | NA | Substitution | |

encoded in the matrix \( S \) via methods developed independently of the AD tool. This clear separation of concerns offers an opportunity for the techniques developed for Steps 2 and 4—which constitute the scope of ColPack and much of this paper—to be interfaced with any AD tool.

3. COLORING PROBLEMS IN COLPACK

The partitioning (coloring) problems in Step 2 of procedure SparseCompute come in several variations depending on:

— whether the derivative matrix of interest is Jacobian or Hessian,
— whether the numerical values of the entries of the matrix \( A \) are to be obtained from the compressed representation \( B \) directly (without any further arithmetic) or indirectly (e.g., by solving for unknowns via successive substitution), and
— in the case of a Jacobian, whether the partitioning is unidirectional (involving only columns or only rows) or bidirectional (involving both columns and rows).

Procedure SparseCompute is described assuming a column-wise unidirectional partitioning (Jacobian computation via the forward mode or Hessian computation via the second-order adjoint mode). In a row-wise unidirectional partitioning, which is a better approach for Jacobian matrices with a few dense rows, the compressed matrix would correspond to the seed-matrix-Jacobian product \( S^T A \) (Jacobian computation via the reverse mode). Similarly, in a bidirectional partitioning, which might be the best approach for Jacobian matrices with both a few dense rows and a few dense columns, the Jacobian entries are recovered from two compressed matrices \( S_1^T A \) and \( A S_2 \) (Jacobian computation via both the forward and the reverse modes).

Table II gives an overview of the coloring abstractions used to model matrix partitioning problems in derivative computation under the various computational scenarios. We describe in the remainder of this section a minimal set of concepts needed to understand these and related weaker models. A more comprehensive discussion of the models is available in [Gebremedhin et al. 2005; Gebremedhin et al. 2007].

3.1. Definitions

In each case in Table II, the structure of a Jacobian matrix \( A \) is represented by the bipartite graph \( G_b(A) = (V_1, V_2, E) \), where the vertex sets \( V_1 \) and \( V_2 \) represent the rows and columns of \( A \), respectively, and each nonzero matrix entry \( A_{ij} \) is represented by the edge \((r_i, c_j)\) in \( E \). Analogously, the structure of a Hessian matrix \( A \) is represented by the adjacency graph \( G_a(A) = (V, E) \), where the vertex set \( V \) represents the columns (or, by symmetry, the rows) of \( A \) and each off-diagonal nonzero matrix entry \( A_{ij} \) and its symmetric counterpart \( A_{ji} \) is represented by the single edge \((c_i, c_j)\) in \( E \). Here and elsewhere in this paper, every diagonal entry \( A_{ii} \) of a Hessian matrix \( A \) is assumed to be nonzero and is not explicitly represented by an edge in \( G_a(A) \).

In a graph \( G = (V, E) \), two distinct vertices are distance-\( k \) neighbors if a shortest path connecting them consists of at most \( k \) edges. A distance-\( k \) coloring of the graph is an assignment of positive integers (called colors) to the vertices such that every pair of distance-
A distance-$k$ coloring of a graph $G = (V, E)$ is equivalent to a distance-1 coloring of the $k$th power graph $G^k = (V, F)$, a graph in which $(v, w) \in F$ whenever vertices $v$ and $w$ are distance-$k$ neighbors in $G$.

A **star coloring** is a distance-1 coloring where, in addition, every path on four vertices uses at least three colors. An **acyclic coloring** is a distance-1 coloring in which every cycle uses at least three colors. The names star and acyclic coloring are due to the structures of two-colored induced subgraphs: a collection of stars in the case of star coloring and a collection of trees in the case of acyclic coloring.

In a bipartite graph $G_b = (V_1, V_2, E)$, a **partial distance-2 coloring** on the vertex set $V_1$ (or $V_2$) is an assignment of colors to the vertices in $V_1$ (or $V_2$) such that a pair of vertices connected by a path of length exactly two edges receives different colors. The term **partial**, which is sometimes omitted when the context is clear, is used to emphasize that the other vertex set remains uncolored.

Star and acyclic bicoloring in a bipartite graph are analogous to star and acyclic coloring in a general graph. But they additionally stipulate that the set of colors assigned to vertices in $V_1$ is disjoint from the set of colors used for vertices in $V_2$, except for a “neutral” color zero assigned to a possibly nonempty subset of vertices in $V_1 \cup V_2$. Specifically, as defined in [Coleman and Verma 1998], a mapping $\phi : [V_1, V_2] \rightarrow \{0, 1, 2, \ldots, p\}$ is a **star bicoloring** of a bipartite graph $G_b = (V_1, V_2, E)$ if the following four conditions hold:

A) If $u \in V_1$ and $v \in V_2$, then $\phi(u) \neq \phi(v)$ or $\phi(u) = \phi(v) = 0$.

B) If $(u, v) \in E$, then $\phi(u) \neq 0$ or $\phi(v) \neq 0$.

C) If vertices $u$ and $v$ are adjacent to a vertex $w$ with $\phi(w) = 0$, then $\phi(u) \neq \phi(v)$.

D) Every path on four vertices uses at least three colors.

An **acyclic bicoloring** is defined in an entirely analogous way except that Condition D is replaced by “Every cycle uses at least three colors”. Notice that Condition B implies that the vertices in $V_1 \cup V_2$ with positive colors constitute a vertex cover in $G_b$.

### 3.2. Colorings and derivative matrices

A distance-2 coloring of the adjacency graph of a Hessian and a partial distance-2 coloring on the column vertices of the bipartite graph of a Jacobian each correspond to **structurally orthogonal** column partitions in the corresponding matrices. Two columns are structurally orthogonal if they do not have a nonzero entry at the same row index. Structural orthogonality is a basic partitioning criterion used in direct methods for sparse derivative computation via compression. Its significance in Jacobian computation was first observed by Curtis, Powell and Reid [1974], and the equivalence between structurally orthogonal partition of a Hessian matrix and distance-2 coloring of its adjacency graph was first established by McCormick [1983]. Because of its equivalence to structurally orthogonal partition in both Jacobians and Hessians, distance-2 coloring can be viewed as an archetypal model in derivative matrix computation [Gebremedhin et al. 2005].

Structurally orthogonal partition is too restrictive a requirement for Hessian computation via a direct method since it does not exploit the symmetry available in a Hessian. A **symmetrically orthogonal partition** of a Hessian $H$—which ensures that for every nonzero $h_{ij}$, either $h_{ij}$ itself or its symmetric counterpart $h_{ji}$ appears as a sole entry in the compressed representation of $H$—overcomes this shortcoming [Gebremedhin et al. 2007]. Coleman and Moré [1984] showed that a symmetrically orthogonal partition of a Hessian can be modeled as a star coloring of the adjacency graph. Coleman and Cai [1986] showed that a partitioning suitable for Hessian computation via a substitution method can be modeled by acyclic coloring.

Star and acyclic coloring as models for Hessian computation were historically preceded by two other models that exploit symmetry only partially. These are restricted star coloring for a direct method (due to Powell and Toint [1979]) and triangular coloring for a substitution method (due to Coleman and Moré [1984]). A **restricted star coloring** is a distance-1 coloring
where, in addition, in every path \(v, w, x\) on three vertices, the terminal vertices \(v\) and \(x\) are allowed to have the same color, but only if the color of the middle vertex \(w\) is lower in value. A color assignment is a *triangular coloring* if there exists a vertex ordering such that the assignment is a distance-1 coloring and in every path \(v, w, x\) on three vertices, the terminal vertices \(v\) and \(x\) receive different colors whenever the middle vertex \(w\) comes after both of the vertices \(v\) and \(x\) in the ordering.

For some sparsity structures, such as an arrow-head type structure, computing a Jacobian by partitioning both columns and rows is more effective than a compression based on unidirectional partitioning. A bidirectional Jacobian computation is modeled by star and acyclic bicoloring of the bipartite graph, as shown by the works of Hossain and Steihaug [1998] and Coleman and Verma [1998]. An optimal method for direct determination of sparse Jacobian matrices has recently been introduced in [Hossain and Steihaug 2012].

### 3.3. Inter-relationships

The coloring variants introduced in Section 3.1 and Section 3.2 can be ranked in an increasing order of restriction in the following manners. In each list, a coloring variant satisfies all of the conditions in the variant immediately preceding it, and by extension all others before that as well. See [Gebremedhin et al. 2007; Gebremedhin et al. 2005] for proofs and other details.

- List 1: distance-1; acyclic; star; restricted star; distance-2 coloring
- List 2: distance-1; acyclic; triangular; restricted star; distance-2 coloring
- List 3: acyclic bicoloring; star bicoloring; partial distance-2 coloring

### 4. Coloring Algorithms in ColPack

Table I (in Section 1) gave an overview of coloring problems, defined on general and bipartite graphs, for which at least one algorithm has been implemented in ColPack. The objective in each of these coloring problems is to minimize the number of colors used. Every one of the problems is NP-hard to solve optimally [Gebremedhin et al. 2005], making the use of heuristics the practical choice. The algorithms in ColPack for these problems are greedy heuristics, in a broad sense: Each algorithm progressively extends a partial coloring by processing one vertex at a time, in some order, in each step permanently assigning a vertex the smallest allowable color. In this section, we describe the algorithms at a fairly high-level. The section is organized in two parts: the first targets coloring problems defined on non-bipartite graphs and the second targets coloring problems defined on bipartite graphs.

#### 4.1. Algorithms for coloring problems on general graphs

Algorithm 2 outlines a template for the various ColPack-algorithms for coloring problems defined on general graphs. In the template, the vertex-indexed array \(color\) is used to store the output of the algorithm, the colors permanently assigned to vertices. Thus \(color[u] = c\) indicates that the vertex \(u\) is assigned the color \(c\). The color-indexed array \(forbiddenColors\) is a working “palette” used during the entire course of the algorithm. It is used to mark the colors that are impermissible to a specific vertex (in a specific step). In particular, \(forbiddenColors[c] = u\) indicates that the color \(c\) is impermissible for the vertex \(u\), and \(forbiddenColors[c] \neq u\) indicates that \(c\) is a candidate color for the vertex \(u\), regardless of the actual value in \(forbiddenColors[c]\). Hence the array \(forbiddenColors\) does not need to be re-initialized during the for-loop over the vertex set of the input graph, which enables runtime efficiency. After the array \(forbiddenColors\) has been populated in a specific step \(i\), it is linearly scanned from left to right until the first index \(c\) in which \(forbiddenColors[c] \neq v_i\), the smallest allowable color for \(v_i\), is encountered and the color \(c\) is assigned to the vertex \(v_i\) (see Line 10). The template is appropriately modified to design algorithms for each of the problems listed in Table I. In each specific coloring algorithm, the time it takes to scan
Algorithm 2: A template for a generic greedy coloring algorithm in ColPack.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>procedure GenericGreedyColoring(G = (V, E))</td>
</tr>
<tr>
<td>2:</td>
<td>Let v_1, v_2, ..., v_</td>
</tr>
<tr>
<td>3:</td>
<td>Initialize forbiddenColors with some value a \notin V</td>
</tr>
<tr>
<td>4:</td>
<td>for i ← 1 to</td>
</tr>
<tr>
<td>5:</td>
<td>for each vertex w in the “appropriate neighborhood” of v_i do</td>
</tr>
<tr>
<td>6:</td>
<td>if color[w] is impermissible for v_i then</td>
</tr>
<tr>
<td>7:</td>
<td>forbiddenColors[color[w]] ← v_i</td>
</tr>
<tr>
<td>8:</td>
<td>end if</td>
</tr>
<tr>
<td>9:</td>
<td>end for</td>
</tr>
<tr>
<td>10:</td>
<td>color[v_i] ← min{c &gt; 0: forbiddenColors[c] \neq v_i}</td>
</tr>
<tr>
<td>11:</td>
<td>if applicable, update the collection of 2-colored induced subgraphs involving v_i</td>
</tr>
<tr>
<td>12:</td>
<td>end for</td>
</tr>
<tr>
<td>13:</td>
<td>end procedure</td>
</tr>
</tbody>
</table>

forbiddenColors to determine a color for the vertex v_i is upper-bounded by the time it takes to populate forbiddenColors in step i.

4.1.1. Path-based algorithms. The manner in which forbiddenColors is populated in a step of the for-loop over V depends on the specific coloring problem being solved and the algorithm used to achieve the solution. By definition, the requirements in a distance-1 coloring involve the distance-1 neighbors of each vertex, whereas the requirements in a distance-2, restricted star, and triangular coloring involve the distance-2 neighbors of each vertex.

The algorithms in ColPack for each of these problems impose the coloring conditions by visiting the distance-1 or distance-2 neighbors of each vertex, whichever is appropriate, exactly once. More specifically, in each of these algorithms, in the step where a vertex v_i is colored, forbiddenColors is populated by traversing every vertex at distance k edges from v_i, for k = 1 or k = 2. Thus, the time complexity of the distance-1 coloring algorithm is $O(|V| \cdot \bar{d}_1) = O(|E|)$ and that of each of the other three algorithms (distance-2, restricted star and triangular) is $O(|V| \cdot \bar{d}_2)$, where $\bar{d}_k$, the average degree-k, denotes the average size of the subgraph induced by the distance-k neighborhood of a vertex in the graph. Since these algorithms rely on merely path traversals in populating the forbiddenColors array, they are referred to as path-based algorithms.

4.1.2. Structure-based algorithms. In contrast, the coloring requirements in a star coloring involve the distance-3 neighbors of each vertex, and the requirements in an acyclic coloring involve cycles, which could entail visits of vertices on arbitrarily long paths. A path-based approach for solving these problems would result in an $O(|V| \cdot \bar{d}_3)$-time algorithm for star coloring and possibly much slower algorithm for acyclic coloring. (In fact an $O(|V| \cdot \bar{d}_3)$-time path-based star coloring algorithm is available in ColPack via a routine named NaiveStarColoring).

Instead, ColPack’s efficient algorithms for star and acyclic coloring, which were first presented in [Gebremedhin et al. 2007], maintain and efficiently utilize the structure of two-colored induced subgraphs in such colorings to attain their respective goals by visiting just the distance-2 neighbors of each vertex at most twice.

The acyclic coloring algorithm uses the disjoint-set data structure to maintain the collection of two-colored trees. The star coloring algorithm uses a simpler (array-based) data structure to maintain the collection of two-colored stars. In both algorithms, in the step where a vertex v_i is colored, the two-colored induced subgraphs incident on the vertex v_i are probed to determine forbidden colors—colors that would lead to two-colored paths on four vertices in the case of star coloring and to two-colored cycles in the case of acyclic coloring.

The key source of runtime efficiency here is that the probes are accomplished through visits to no farther than the distance-2 neighbors of the vertex v_i. Once the smallest color permis-
sible for the vertex $v_i$ has been determined, the collection of two-colored induced subgraphs needs to be updated (see Line 11) to reflect that $v_i$ has been colored. This update is also performed with visits to just distance-2 neighbors. Thus, the time complexity of the star coloring algorithm is $O(|V| \cdot \mathcal{d}_2)$. And the complexity of the acyclic coloring algorithm is $O(|V| \cdot \mathcal{d}_2 \cdot \alpha)$, where $\alpha$ is the inverse of Ackermann’s function and is associated with efficient implementations of the disjoint-set operations Find and Union required by the algorithm. See [Gebremedhin et al. 2007] for details.

4.2. Algorithms for coloring problems on bipartite graphs

4.2.1. Partial distance-2 coloring. For a bipartite graph $G_b = (V_1, V_2, E)$, a path-based algorithm for a (partial) distance-2 coloring on the vertex set $V_1$ or $V_2$ can be obtained by adapting the template in Algorithm 2 in a fairly straightforward manner. ColPack contains such implementations for both a distance-2 coloring on $V_1$ and a distance-2 coloring on $V_2$. The respective time complexities of the algorithms are:

$$O(|V_1| \cdot \bar{d}_1(V_1) \cdot \Delta(V_2)) = O(|E| \cdot \Delta(V_2)) \text{ and } O(|V_2| \cdot \bar{d}_1(V_2) \cdot \Delta(V_1)) = O(|E| \cdot \Delta(V_1)),$$

where $\bar{d}_1(V_i)$ denotes the average degree-1 in the vertex set $V_i$, $i = 1, 2$, and $\Delta(V_i)$ denotes the maximum degree-1 in the vertex set $V_i$, $i = 1, 2$.

4.2.2. Bicoloring. ColPack contains implementations of a variety of algorithms for star bicoloring (two-sided coloring in a bipartite graph). The algorithmic variations here originate from two orthogonal sources:

— whether two-colored induced subgraphs are maintained or paths are traversed, and
— whether the vertex cover implied in a bicoloring is computed explicitly in a pre-coloring step (for example, as is done in [Coleman and Verma 1998]) or it is computed implicitly as the coloring proceeds.

The following variants of algorithms are currently available in ColPack:

— ExplicitCoverStarBicoloring: vertex cover is explicitly pre-computed; two-colored structures are maintained and used in the bicoloring stage. The star bicoloring algorithm works in a manner analogous to the star coloring algorithm on adjacency graphs of symmetric matrices.

— ExplicitCoverModifiedStarBicoloring: vertex cover is explicitly pre-computed; two-colored structures are maintained and used in the coloring phase. The star bicoloring algorithm has certain modified aspects.

— ImplicitCoverStarBicoloring: vertex cover is implicitly computed as the bicoloring proceeds; two-colored structures are maintained and used in the bicoloring algorithm.

— ImplicitCoverGreedyStarBicoloring: vertex cover is implicitly computed as the bicoloring proceeds; the bicoloring stage is based on path traversal.

5. ORDERING TECHNIQUES IN COLPACK

The order in which vertices are processed in a greedy coloring algorithm determines the number of colors used by the algorithm. ColPack contains implementations of various effective ordering techniques for each of the coloring problems it supports. The ordering techniques can be classified in two categories: degree-based and coloring-based. We discuss the degree-based techniques and related concepts in sections 5.1 through 5.5, and the coloring-based techniques in section 5.6.

5.1. Degree-based orderings, the distance-1 coloring case

There are four fundamental degree-based ordering techniques in ColPack: Largest First (LF), Smallest Last (SL), Incidence Degree (ID), and Dynamic Largest First (DLF). The
5.1.1. Characterization. We characterize the four ordering techniques using a common framework in a manner that is independent of a coloring algorithm. The independence aids the creation of modular (and efficient) implementations and eases the task of identifying other contexts in which the ordering techniques are useful.

Our characterization uses the notions of back and forward degree. In an ordering \( \pi = \{v_1, v_2, \ldots, v_n\} \) of the vertices of a graph \( G = (V, E) \), the back degree of the vertex \( v_i \) is the number of neighbors of \( v_i \) that appear before \( v_i \) in \( \pi \), and the forward degree of \( v_i \) is the number of neighbors of \( v_i \) that appear after \( v_i \) in \( \pi \). Clearly, the degree of \( v_i \), the number of its adjacent vertices in \( G \), is equal to the sum of its back degree and forward degree. See Figure 1 for an illustration. Note that the degree of a vertex is a static property, whereas its back degree and forward degree are dynamic quantities whose values change as the determination of the ordering proceeds. We summarize below the characterization of the four ordering kinds.

**LF** Defined from lowest to highest, \( v_1 \) to \( v_n \), using the static quantity degree. It is simply a sorting of the vertices in a non-increasing order of degrees in \( G \).

**ID** Defined from lowest to highest, \( v_1 \) to \( v_n \), using the dynamic quantity back degree. Initially, the back degree of every vertex is zero. The first vertex \( v_1 \), a vertex with the largest back degree in \( V \), is hence arbitrary. With \( v_1 \) determined, the back degree of every neighbor of \( v_1 \) is, by definition, one more than its original value. The next vertex, \( v_2 \), is a vertex with the largest back degree among the vertices \( U = V \setminus \{v_1\} \). Suppose the first \( i - 1 \) entries of the ordered vertex set have been determined continuing in this manner. The \( i \)th vertex is then a vertex with the largest back degree among the yet-to-be-ordered vertices in the set \( U = V \setminus \{v_1, v_2, \ldots, v_{i-1}\} \).

**SL** Defined from highest to lowest, \( v_n \) to \( v_1 \), using the dynamic quantity back degree. Initially, the back degree of every vertex is equal to its degree. The last vertex \( v_n \) is a vertex with the smallest back degree (and thus degree) in \( V \). With \( v_n \) determined, the back degree of every neighbor of \( v_n \), by definition, is one less than its original value. The next vertex in the ordering, \( v_{n-1} \), is a vertex with the smallest back degree among the vertices \( W = V \setminus \{v_n\} \). Suppose the last \( n - i - 1 \) entries of the ordered vertex set have been determined continuing in this manner. The \( i \)th vertex in the ordering is then a vertex with the smallest back degree among the yet-to-be-ordered vertices in the set \( W = V \setminus \{v_n, v_{n-1}, \ldots, v_{i+1}\} \).

**DLF** Defined from lowest to highest, \( v_1 \) to \( v_n \), using the dynamic quantity forward degree. Initially, the forward degree of every vertex is equal to its degree. The first vertex \( v_1 \) is a vertex with the largest forward degree in \( V \). With \( v_1 \) determined, the forward degree of every neighbor of \( v_1 \) is, by definition, one less than its original value. The next vertex, \( v_2 \), is a vertex with the largest forward degree among the vertices \( U = V \setminus \{v_1\} \). Suppose the first \( i - 1 \) entries of the ordered vertex set have been determined continuing in this manner. The \( i \)th vertex is then a vertex with the largest forward degree among the yet-to-be-ordered vertices in the set \( U = V \setminus \{v_1, v_2, \ldots, v_{i-1}\} \).
Fig. 2. Data structure for efficient implementation of the ordering techniques SL, DLF and ID. The SL ordering case is illustrated here, where the left subfigure shows the initial condition, and the right subfigure shows the condition after vertex $h$ is ordered (last in a list not shown here).

5.1.2. Computation. Besides serving as a conceptual framework, the above characterization of the orderings also facilitates the design and implementation of fast algorithms for computing (obtaining) the orderings.

Consider an LF ordering for example. Since the degree of a vertex in a connected graph $G = (V,E)$ is an integer in the interval $[1, \Delta]$, where $\Delta \leq |V|$ is the maximum degree in the graph, an LF ordering can be obtained in linear time by bucket-sorting the vertices according to their degrees in $G$ using an auxiliary array of size $\Delta$.

An ID, DLF or SL ordering can be computed in a similar fashion, except now the back or forward degrees of adjacent vertices need to be updated every time a vertex is selected to be placed in the ordered list. The update involves relocating the neighbors of the selected vertex into appropriate buckets, but the work can still be made proportional to the degree-1 of the vertex by using suitable pointer techniques. Figure 2 illustrates the technique used in ColPack to achieve this. The illustration is for SL ordering, but the same principle is used for DLF and ID orderings as well. The “Positions” pointer array there is used to locate the position of a vertex in the “Buckets” array, which maintains the relevant degree information, in this case back degree, in constant time. At a given step of the ordering, the last vertex in the first non-empty bucket in the array “Buckets” is the “Candidate” vertex, the vertex to be ordered at that step. Once the candidate vertex is ordered (removed from the Buckets-data structure), each of its neighbors need to be moved to a new bucket to reflect that the back degree of each is decremented by one. In order to achieve this with minimal data movement, the neighbor is first swapped with the last element in the current bucket, and then it is simply moved to the new bucket. The left subfigure in Figure 2 depicts the initial
condition, where vertices are bucket sorted according to their back degrees (which initially is the same as degrees). The right subfigure shows the updated data structure after the vertex with the smallest degree, \(v_1\), has been ordered. In this (and similar) manner, each of the three ordering variants SL, DLF and ID is implemented in ColPack such that its runtime is \(O(|E|)\). In all of the ordering routines in ColPack, ties are broken arbitrarily.

### 5.2. The coloring number of a graph

A general heuristic intuition behind the ordering techniques being discussed is that in the course of a greedy algorithm that colors the vertices in the order \(v_1\) to \(v_n\), vertices that are more constrained in the choice of colors get colored early, making it more likely for the algorithm to use fewer colors compared to, say, an arbitrary order. In contrast to LF, ID and DLF ordering, an SL ordering, however, goes a step further—it minimizes a mathematically well defined objective.

Given a graph \(G\) and a vertex ordering \(\pi\), let \(B_{\pi}(G)\) denote the maximum back degree in the ordering \(\pi\), and \(\chi(G,\pi)\) denote the number of colors used by a greedy algorithm that uses the ordering \(\pi\) to color \(G\). Further, let \(\Delta(G)\) denote the maximum degree in \(G\) and \(\chi(G)\) denote the chromatic number of \(G\), the optimal (least) number of colors required to color \(G\). Then, it is fairly easy to see that the relationship

\[
\chi(G,\pi) \leq B_{\pi}(G) + 1, \tag{1}
\]

always holds, and can be extended at both ends as follows:

\[
\chi(G) \leq \chi(G,\pi) \leq B_{\pi}(G) + 1 \leq \Delta(G) + 1. \tag{2}
\]

Now, different vertex orderings could result in different maximum back degrees. An SL-ordering gives the minimum maximum back degree, not only among the four ordering variants being discussed, but among all possible orderings. That is,

\[
B_{SL}(G) = \min_{\pi} B_{\pi}(G), \tag{3}
\]

where the minimum is taken over the \(n!\) possible vertex orderings of the graph \(G\) [Matula et al. 1972; Finck and Sachs 1969]. The quantity \(B_{SL}(G) + 1\) is a graph parameter (a unique value for a given graph) known as the coloring number \(col(G)\) of \(G\) [Erdős and Hajnal 1966], so named due to its connection to coloring, although its definition, as presented here, is independent of coloring.

Noting that the size of the largest clique in \(G\), the clique number \(\omega(G)\), is an obvious lower bound on the chromatic number \(\chi(G)\), the relationship in Equation (2) can be extended at the lower end in the following manner:

\[
\omega(G) \leq \chi(G) \leq B_{\pi}(G) + 1 \leq \Delta(G) + 1. \tag{4}
\]

And the relationship among the clique number, the chromatic number, the coloring number, the maximum back degree, and the maximum degree for any given graph \(G\) and vertex ordering \(\pi\) can be summarized in the following manner:

\[
\omega(G) \leq \chi(G) \leq \text{col}(G) = B_{SL}(G) + 1 \leq B_{\pi}(G) + 1 \leq \Delta(G) + 1. \tag{5}
\]

Note that although \(B_{SL}(G) \leq B_{\pi}(G)\) holds for any graph \(G\) and ordering \(\pi\), no statement can be made regarding the relationship between the actual number of colors output by a greedy algorithm that uses an SL ordering and the number output by an algorithm that uses the ordering \(\pi\). In other words, the expression \(\chi(G,SL)\) could be less than, greater than, or equal to the expression \(\chi(G,\pi)\) for an ordering \(\pi\) that is not SL. Moreover, since multiple vertices in a graph could have the same degree value, none of the ordering variants LF, ID, DLF and SL is uniquely defined for a given graph. Instead each defines a class of orderings, and a tie-breaking strategy is needed to determine a specific member in each
For instance, two vertex orderings $\pi_1$ and $\pi_2$ could be distinct and yet each be an SL ordering—and $\chi(G, \pi_1)$ could differ from $\chi(G, \pi_2)$. However, no matter how ties are broken in an SL ordering, all such orderings give the same $B_{SL}(G)$, and hence the coloring number is uniquely determined.

5.3. Relationship between coloring number and other graph parameters

The coloring number $\text{col}(G)$ of a graph has interesting connection with several graph concepts beyond coloring, including degeneracy, core and arboricity [Matula et al. 1972; Szekeres and Wilf 1968; Lick and White 1970; Diestel 2000; Gebremedhin et al. 2005]. At the heart of this connection lies a relationship between the parameter $\text{col}(G)$ and yet another graph parameter, the maximum minimum degree in an induced subgraph of $G$, where the maximum is taken over all possible induced subgraphs of $G$. Specifically, let

$$\kappa = \max_{H \subseteq G} \delta(H),$$

where $H \subseteq G$ denotes a nonempty induced subgraph of $G$ and $\delta(H)$ denotes the minimum degree in $H$. Matula [1972] and Szekeres and Wilf [1968], independently, showed that

$$\text{col}(G) = \kappa + 1. \tag{7}$$

In the social and biological network literature, this quantity is known as the $k$-core.

5.4. Saturation Degree-based coloring

Saturation Degree ordering, due to Brelaz [1979], is another known ordering technique for greedy coloring. It can be best viewed in contrast to the ID ordering technique: in a coloring algorithm that uses an ID ordering, the $i$th vertex is a vertex with the largest number of already colored adjacent vertices, whereas in an algorithm that uses an SD ordering it is a vertex that has the largest number of distinctly colored adjacent vertices. Clearly, the characterization of an SD ordering cannot be separated from the rest of the coloring algorithm, and the ordering is non-ameenable for a linear-time implementation in the size of the graph. For these reasons we chose not to include it in ColPack. Turner [1988] has shown that using careful choice of data structures, a coloring algorithm that in effect employs an SD ordering can be implemented such that its complexity is $O(|E| \log |V|)$. We have implemented such an algorithm and we will report results on it in the experiments discussed in Section 8.

5.5. Extension of degree-based orderings to other coloring problems

We have adapted the ordering techniques LF, ID, DLF and SL—the characterizations given in section 5.1.1—to suit the various specialized coloring problems supported by ColPack. The adaptation involves extending the notion of degree, since the specialized algorithms involve visits to the distance-2 neighborhood of a vertex in a graph and/or the coloring is performed on a bipartite graph. In ColPack, each ordering technique is implemented in such a way that its time complexity is upper-bounded by the complexity of the coloring algorithm for which it is designed.

5.6. Coloring-based ordering

Let $\phi : V \rightarrow \{1, 2, \ldots, C\}$ be a distance-1 coloring of a graph $G = (V, E)$ on $n$ vertices. Now consider an ordering $\pi = \{v_1, v_2, \ldots, v_n\}$ in which vertices that belong to a color class—those vertices with the same color—are always listed consecutively, i.e., if $\phi(v_i) = \phi(v_k) = c$, then $\phi(v_j) = c$ for $i < j < k$. A greedy algorithm that uses the ordering $\pi$ obtains a new distance-1 coloring of $G$ using $C$ or fewer colors. Culberson [1992] applied this property of the greedy coloring algorithm in his method called Iterative Greedy (IG) to attempt to successively reduce the number of colors used. As long as vertices of a color class are listed
consecutively, there is a degree of freedom in the way in which the color classes themselves can be ordered. One of the better strategies used in Culberson’s IG method is to order the classes in reverse order of their introduction, i.e., the ordering $\pi$ is obtained by listing vertices of color $C$ consecutively, followed by vertices of color $C-1$, and so on, and finally vertices of color 1.

We truncate this variant of Culberson’s IG method to just two iterations, where the first iteration is used to merely determine an ordering for the second, where a fresh coloring is obtained. Specifically, a coloring-based ordering routine available in ColPack takes as input a coloring $\phi$ of a graph and gives as output a vertex ordering $\pi$ based on a reverse listing of the color classes in $\phi$. ColPack has coloring-based ordering routines for distance-1 and distance-2 coloring of general graphs and for partial distance-2 coloring and bicoloring of bipartite graphs. We note that a back-tracking algorithm similar in spirit to IG method but with better performance in terms of number of colors has been proposed recently in [Bhowmick and Hovland 2008].

6. OTHER FUNCTIONALITIES IN COLPACK

Besides coloring and ordering capabilities, ColPack contains two classes of functionalities.

6.1. Recovery routines

Recall that the last step in the procedure for sparse derivative computation (Algorithm 1, Section 2) involves recovering the numerical values of the entries of a derivative from a compressed representation. The following recovery routines are currently available in ColPack:

— Recovery routines for Hessian computation, both for direct (via star coloring) and substitution-based (via acyclic coloring) methods. The recovery routine for a substitution-based method uses the two-colored trees the acyclic coloring algorithm maintains during its execution. This routine—as well as the corresponding routine for a direct method—has been described and analyzed in detail in a recent paper [Gebremedhin et al. 2009].

— Recovery routines for unidirectional, direct Jacobian computation, both for column-wise and row-wise computation (via appropriate distance-2 coloring).

— Recovery routines for bidirectional, direct Jacobian computation via star bicoloring.

To facilitate the use of ColPack by other scientific computing software tools, each recovery routine in ColPack comes in three variations corresponding to three different formats in which the values in a “de-compressed” derivative matrix are returned. These three formats are: Row Compressed Format (which is used among others by ADOL-C [Walther and Griewank 2012]), Coordinate Format and Sparse Solvers Format.

6.2. Graph reading routines

In ColPack, we use the Compressed Sparse Row (CSR) data structure to store graphs. CSR is a widely used data structure in scientific computing. It consists essentially of two one-dimensional integer arrays (vectors), one corresponding to vertices and the other to edges. Indices to these arrays correspond to vertex or edge identifiers. For each vertex, the adjacency list of the vertex is stored in consecutive locations in the edge array and the vertex array is used to point to the beginning and the end.

As a supporting functionality, ColPack provides routines for constructing bipartite graph data structures (for Jacobians) and adjacency graph data structures (for Hessians) from files specifying matrix sparsity structures. Various file formats, including formats used by Matrix Market [Boisvert et al. 1996], Harwell-Boeing [Duff 1992] and MeTis [Karypis and Kumar 1998] are supported.

1www.intel.com/software/products/mkl/docs/webhelp/appendices/mkl_appA_SMSF.html
7. ColPack’s Organization and Usage

ColPack is written in an object-oriented fashion in C++. It is designed to be modular and extensible. Figure 3 gives an overview of the structure of the major classes of ColPack.

The entire ColPack package is under the ColPack namespace. Two core classes, GraphCore and BipartiteGraphCore, are used to store the general graph and bipartite graph, respectively, data structures in CSR format. The classes GraphCore and BipartiteGraphCore are abstract (pure virtual) with no methods to manipulate data. The classes GraphInputOutput and BipartiteGraphInputOutput, which inherit the classes GraphCore and BipartiteGraphCore, respectively, contain routines for building adjacency graphs and bipartite graphs from files specifying matrix sparsity structures.

The class GraphInputOutput starts up an inheritance chain collectively containing implementations of various coloring and ordering algorithms for general graphs. For a similar purpose on bipartite graphs, the class BipartiteGraphInputOutput starts up two separate inheritance chains, one concerning partial distance-2 coloring and the other bicoloring. ColPack functions that a user typically needs to call directly are made available via the appropriate Interface classes:

— GraphColoringInterface,
— BipartiteGraphPartialColoringInterface, and
— BipartiteGraphBicoloringInterface
The classes `HessianRecovery`, `JacobianRecovery1D` and `JacobianRecovery2D` house the appropriate routines for recovering a derivative matrix from its compressed representation. The class `RecoveryCore`, which is inherited by each of the three classes, contains functions for allocating and deallocating memory associated with the recovery routines.

A description of the major functions in the interface classes of ColPack as well as other related documentation is available at the package’s distribution website.

In the appendix, we have included two code samples that illustrate how ColPack’s capabilities are used.

8. EXPERIMENTAL ANALYSIS

It is useful to know how effective in practice the various ordering techniques discussed in Section 5 are and how they compare against each other when used in the context of the various greedy coloring algorithms discussed in Section 4. We present in this section a set of experimental results that illustrate different aspects of the ordering techniques as well as of the encompassing coloring algorithms. We begin in Sections 8.1 and 8.2 by discussing the experimental setup. We then present results on distance-1 coloring in Section 8.3; results on distance-2, star and acyclic coloring in Section 8.4; and results on coloring bipartite graphs in Section 8.5.

8.1. Data set

We sought to include in the testbed a wide variety of graph structures. The testbed we assembled consists of 30 graphs grouped in two categories.

The first category contains ten graphs obtained from the University of Florida Matrix Collection [Davis and Hu 2011]. These graphs originate from various areas in scientific computing, including structural engineering, civil engineering, and automotive and ship industry.

The second category consists of twenty synthetically generated graphs representing four different graph “classes”. The first represented class is that of planar graphs. The graphs, more specifically, are maximally planar—the degree of every vertex is at least five—and are generated using the expansion method described in [Morgenstern and Shapiro 1991]. The remaining three classes of graphs are generated using the R-MAT algorithm [Chakrabarti and Faloutsos 2006]. The R-MAT algorithm, using various parameter configurations, allows for generating graphs with varying properties in terms of degree distributions and local density. The three classes we generated are Erdős-Renyi random, and two kinds of small-world type graphs (we use the descriptor small-world in a loose sense).

The R-MAT algorithm works by recursively subdividing the adjacency matrix of the graph to be generated (a \(|V| \times |V|\) matrix) into four quadrants \((1, 1), (1, 2), (2, 1), (2, 2)\) and distributing the \(|E|\) edges within the quadrants with specified probabilities. The distribution is determined by four non-negative parameters \((a, b, c, d)\) whose sum equals one. Initially, every entry of the adjacency matrix is zero (no edges added). The algorithm places an edge in the matrix by choosing one of the four quadrants \((1, 1), (1, 2), (2, 1), (2, 2)\) with probabilities \(a, b, c, \) or \(d\), respectively. The chosen quadrant is then subdivided into four smaller partitions and the procedure is repeated until a 1×1 quadrant is reached, where the entry is incremented (the edge is placed). The algorithm repeats the edge generation process \(|E|\) times to create the desired graph \(G = (V, E)\).

The Erdős-Renyi random graphs in our testbed are generated by letting each of the four R-MAT parameters be 0.25. The first type of small-world graphs is generated using the set of parameters \((0.45, 0.15, 0.15, 0.25)\), whereas the second type is generated using the parameter set \((0.55, 0.15, 0.15, 0.15)\). The parameter combination in the second type results in graphs with larger maximum degrees (and also denser local subgraphs) compared to the
Table III. Structural statistics and distance-1 coloring results on the scientific computing (sc) graphs.

<table>
<thead>
<tr>
<th>Name</th>
<th>V</th>
<th>E</th>
<th>∆</th>
<th>BSL</th>
<th>ω</th>
<th>SL</th>
<th>ID</th>
<th>LF</th>
<th>DLF</th>
<th>N</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc1</td>
<td>415,863</td>
<td>9,912,536</td>
<td>76</td>
<td>35</td>
<td>21</td>
<td>35</td>
<td>38</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>sc2</td>
<td>952,203</td>
<td>22,785,136</td>
<td>76</td>
<td>35</td>
<td>21</td>
<td>35</td>
<td>38</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>sc3</td>
<td>140,874</td>
<td>3,836,265</td>
<td>76</td>
<td>35</td>
<td>21</td>
<td>35</td>
<td>38</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>sc4</td>
<td>179,860</td>
<td>4,966,618</td>
<td>76</td>
<td>35</td>
<td>21</td>
<td>35</td>
<td>38</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>sc5</td>
<td>87,804</td>
<td>2,565,054</td>
<td>76</td>
<td>35</td>
<td>21</td>
<td>35</td>
<td>38</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>sc6</td>
<td>52,329</td>
<td>1,323,067</td>
<td>76</td>
<td>35</td>
<td>21</td>
<td>35</td>
<td>38</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>

Table IV. Structural statistics and distance-1 coloring results on the synthetic (planar, Erdős-Renyi random, small-world) graphs.

<table>
<thead>
<tr>
<th>Name</th>
<th>V</th>
<th>E</th>
<th>∆</th>
<th>BSL</th>
<th>ω</th>
<th>SL</th>
<th>ID</th>
<th>LF</th>
<th>DLF</th>
<th>N</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>102,093</td>
<td>306,273</td>
<td>35</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>p2</td>
<td>204,049</td>
<td>612,141</td>
<td>29</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>p3</td>
<td>407,521</td>
<td>1,222,557</td>
<td>29</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>p4</td>
<td>819,502</td>
<td>2,458,500</td>
<td>32</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>p5</td>
<td>1,625,972</td>
<td>4,877,910</td>
<td>37</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>er1</td>
<td>200,000</td>
<td>2,468,251</td>
<td>100</td>
<td>26</td>
<td>3</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>er2</td>
<td>200,000</td>
<td>4,443,541</td>
<td>173</td>
<td>48</td>
<td>3</td>
<td>23</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>er3</td>
<td>200,000</td>
<td>6,910,713</td>
<td>253</td>
<td>77</td>
<td>3</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>29</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>er4</td>
<td>200,000</td>
<td>8,882,591</td>
<td>322</td>
<td>100</td>
<td>3</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>34</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>er5</td>
<td>200,000</td>
<td>11,347,805</td>
<td>410</td>
<td>128</td>
<td>4</td>
<td>43</td>
<td>43</td>
<td>43</td>
<td>41</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>sw1</td>
<td>200,000</td>
<td>2,422,134</td>
<td>534</td>
<td>31</td>
<td>5</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>20</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>sw2</td>
<td>200,000</td>
<td>4,350,665</td>
<td>954</td>
<td>57</td>
<td>8</td>
<td>27</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>sw3</td>
<td>200,000</td>
<td>6,747,685</td>
<td>1,380</td>
<td>89</td>
<td>10</td>
<td>36</td>
<td>36</td>
<td>37</td>
<td>39</td>
<td>61</td>
<td>61</td>
</tr>
<tr>
<td>sw4</td>
<td>200,000</td>
<td>8,656,160</td>
<td>1,783</td>
<td>115</td>
<td>12</td>
<td>43</td>
<td>43</td>
<td>45</td>
<td>46</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>sw5</td>
<td>200,000</td>
<td>11,030,782</td>
<td>2,222</td>
<td>147</td>
<td>14</td>
<td>51</td>
<td>51</td>
<td>53</td>
<td>54</td>
<td>85</td>
<td>85</td>
</tr>
</tbody>
</table>

8.2. Properties of the test graphs

We provide in Tables III and IV basic structural properties as well as some computed quantities in the test graphs. The information in the two tables is catalogued in four parts:

— the first part shows abbreviated names of the graphs: scientific computing (sc), planar (p), Erdős-Renyi (er), and small-world (sw).
— [V] and [E] show the number of vertices and edges in the input graph G.
— ∆ shows the maximum degree in G; BSL shows the minimum maximum back degree in G; and ω shows the clique number of G.
— the last six columns list the number of colors used by the greedy distance-1 coloring algorithm while employing the ordering variants Smallest Last (SL), Incidence Degree (ID), Largest First (LF), Dynamic Largest First (DLF), Natural (N) and Saturation Degree (SD). (Natural corresponds to the order in which vertices appear in the input graph.)

first type. All of the R-MAT graphs were generated using the GTgraph Synthetic Graph Generator Suite\(^2\). Duplicate edges and self loops were removed.

\(^2\)http://www-static.cc.gatech.edu/kamesh/GTgraph/
We will shortly discuss the numbers in these last six columns in some detail, but first we comment on a few other aspects in the tables.

As discussed in Section 5.2, the minimum maximum back degree ($B_{SL}$) is computed using the Smallest Last ordering. It is equal to the coloring number $col(G)$ minus one, and is a tighter upper bound on the chromatic number $\chi(G)$ than the bound $\Delta + 1$.

The clique number $\omega(G)$, which is a lower bound on the chromatic number $\chi(G)$, is NP-hard to compute. We computed it using an exact, exhaustive search algorithm that finds the largest clique to which a vertex belongs. To compute the largest clique to which a given vertex belongs, we implemented the per-vertex clique finding algorithm described in [Turner 1988]. We pruned the exhaustive search (for the entire graph) by maintaining the size of the largest clique found so far and invoking the clique-finding algorithm for a particular vertex only if the degree of that vertex is larger than the current maximum clique. This pruned algorithm, whose worst-case complexity is still exponential, is quite fast in practice. As an example, a typical graph, the algorithm took a few seconds, and on a typical swi graph, it took a few dozens of seconds.

Finally, the SD coloring algorithm (whose results are listed in the last column in Tables III and IV) is our implementation of the algorithm described by Turner [1988]. The algorithm was briefly discussed in Section 5.4, and it is to be recalled that it is not included in ColPack.

Compute platform. All of the experiments in this paper were run on an Intel Nehalem microarchitecture equipped with Intel(R) Core(TM) i7 CPU 860 processors running at 2.80GHz. The system has 4 cores and 8 threads, but the experiments were run on single processor and a single thread. The total memory size of the system is 16 GB, with $4 \times 32$ KB Instruction and $4 \times 32$ KB Data Level-1 cache, $4 \times 256$ KB Level-2 cache, and 8 MB shared Level-3 cache. The operating system is GNU/Linux, and the compiler is g++.

8.3. Results on distance-1 coloring

8.3.1. Number of colors used. As mentioned earlier, the last six columns in Tables III and IV list the number of colors $\chi(G, \pi)$ used by the greedy algorithm for distance-1 coloring when various ordering techniques $\pi$ are used. For each graph, the smallest $\chi(G, \pi)$, among the six ordering variants $\pi$ considered, is shown in boldface. We highlight a few general observations on the results shown in the two tables.

— In nearly every one of the graphs from the scientific computing category, either SL or ID ordering gave the fewest colors. The difference between the two is in turn quite small, usually one or two colors, whereas the reduction they offer compared to LF, DLF and N is relatively large.

— As expected from theory, the greedy algorithm using SL ordering colored planar graphs using at most six colors.

— For the Erdős-Renyi random graphs, although the various ordering techniques in general gave similar number of colors, DLF gave the fewest.

— For the small-world graphs, SL, ID, LF, and DLF reduced the number of colors compared to Natural ordering, but showed little difference among each other, with SL or ID once again giving the fewest colors.

— SD ordering almost invariably used more colors than ID and SL ordering in the graphs from both test sets. (This is rather surprising since SD ordering employs a vertex selection criteria that intuitively seems more accurate than ID ordering in capturing color choice constraints. In our implementation of the SD ordering, ties are broken in favor of the vertex with larger degree; in all other ordering techniques in ColPack ties are broken arbitrarily.)
Fig. 4. Left: Maximum degree $\Delta$, maximum back degree using Natural ordering $B_N$, and maximum back degree using Smallest Last ordering $B_{SL}$, each normalized by $B_{SL}$. Right: the number of colors by the greedy algorithm using Natural ordering (N) and Smallest Last ordering (SL), the minimum maximum back degree $B_{SL}$, and the clique number $\omega$, each normalized by $\omega$. Test graphs: synthetic (random and small-world).
8.3.2. Maximum back degrees, comparison with optimal values. Recall from Equations (4) and (5) in Section 5 that:

— the maximum back degree \( B_\pi(G) \) plus one is a tighter upper bound on the number of colors \( \chi(G, \pi) \) than the maximum degree \( \Delta(G) \) plus one;

— the minimum maximum back degree is attained when Smallest Last ordering is used; and

— the clique number \( \omega(G) \) is a lower bound on the chromatic number \( \chi(G) \).

The results reported in Tables III and IV readily reflect several aspects of these facts.

To provide further insight into the relative proximity among the various quantities, we present in Figure 4 plots that show how the various quantities relate to each other. The plots are for the R-MAT generated synthetic graphs (random and small-world type). In all those figures, we use Smallest Last ordering as the representative effective ordering technique and compare it against Natural ordering. The left subfigures show curves corresponding to three quantities: maximum degree \( \Delta(G) \), maximum back degree using Natural ordering \( B_N(G) \), and maximum back degree using Smallest Last ordering \( B_{SL}(G) \) (the minimum maximum back degree in the graph). Each of the three quantities is normalized by \( B_{SL}(G) \). The right subfigures show curves corresponding to four quantities: the number of colors \( \chi(G,N) \) used by the greedy algorithm when Natural ordering is employed (denoted by N in the figure), the number of colors \( \chi(G,SL) \) used by the greedy algorithm when Smallest Last ordering is used (denoted by SL in the figure), the minimum maximum back degree \( B_{SL}(G) \), and the clique number \( \omega(G) \). Each quantity is normalized by \( \omega(G) \).

For a given graph \( G \), let \( r_\chi(G) \) denote the ratio between the approximate solution \( \chi(G,SL) \) and the optimal solution \( \chi(G) \) (that is, \( r_\chi(G) \equiv \frac{\chi(G,SL)}{\chi(G)} \)). Similarly, let \( r_\omega(G) \equiv \frac{\chi(G,SL)}{\omega(G)} \). Clearly \( r_\chi(G) \leq r_\omega(G) \). The results in Figure 4 show that the ratio \( r_\omega \) is relatively small indicating that the approximation ratio \( r_\chi \) is even smaller. The table below summarizes the observed maximum value on the ratio \( r_\omega \) in each of the five graph groups used in the experiments:

\[
\begin{array}{c|cccccc}
  & sc & p & er & sw I & sw II \\
\hline
r_\omega \text{ is at most} & 1.8 & 2 & 10 & 4 & 2 \\
\end{array}
\]

The ratio \( r_\omega \) is the highest for the group \( er \) because these graphs (random) are the most unstructured among all in the testbed.

Another observation to be made from Figure 4 is that the upper bound \( \Delta(G) + 1 \) customarily given for \( \chi(G) \) is often several factors larger than the bound \( B_{SL}(G) + 1 \). The following table summarizes this observation, again for all test graph groups:

\[
\begin{array}{c|cccccc}
  \Delta(G) & sc & p & er & sw I & sw II \\
\hline
\text{could be as high as} & 8 & 6 & 4 & 18 & 25 \\
\end{array}
\]

8.3.3. Run time. The fact that the greedy coloring algorithm with an ordering such as SL quickly computes a solution so close to optimal cannot be overstated. As discussed earlier, the time complexity of the greedy distance-1 coloring algorithm in \texttt{ColPack} is linear in the size of the graph (Section 4), and each of the ordering techniques LF, DLF, ID and SL is implemented in \texttt{ColPack} so that its time complexity is also linear (Section 5). Figure 5 shows the observed total execution times (i.e. ordering plus coloring) in seconds for distance-1 coloring while using the ordering techniques N, LF, ID, SL, and DLF on the \textit{synthetic} graphs. Also shown in the same figure is the total execution time for the SD ordering-based coloring algorithm (Turner’s algorithm). It can be seen that the N and LF ordering based coloring algorithms take about the same time and are the fastest. These are closely followed by the algorithms using ID, SL and DLF ordering, which in turn take about the same time. And, finally, it can be seen that coloring based on SD ordering is significantly slower than
8.3.4. Coloring-based ordering results. Experiments we did on coloring-based ordering (discussed in Section 5.6) showed that if one starts with a Natural ordering, re-coloring reduces the number of colors used. The magnitude of reduction is comparable to what one would see in using SL ordering compared to Natural ordering. If one starts with an ordering such as Smallest Last, however, re-coloring provides almost no further reduction in number of colors. We omit the results here for space considerations.

8.4. Results on distance-2, star and acyclic coloring

8.4.1. Impact of ordering techniques on number of colors. We have conducted experiments using the degree-1 and degree-2 versions of the various ordering techniques (LF, DLF, SL, ID) in the context of distance-2, star and acyclic coloring. We again omit the results for space considerations, but point out two major observations we made from the results:

— Compared to Natural ordering, the various degree-1-based ordering techniques reduce the number of colors in the distance-2, star and acyclic coloring algorithms. The reduction is
Fig. 6. Number of colors used by the greedy algorithms for distance-1 (D1), acyclic (A), star (S) and distance-2 (D2) coloring on the scientific computing graphs. In all cases, SL ordering was used.

Table V. Number of colors used by the distance-1 (D1), acyclic (A), star and distance-2 (D2) coloring algorithms on the synthetic graphs. In all cases, Smallest Last ordering was used.

<table>
<thead>
<tr>
<th>Name</th>
<th>D1</th>
<th>A</th>
<th>S</th>
<th>D2</th>
<th>Δ</th>
<th>Name</th>
<th>D1</th>
<th>A</th>
<th>S</th>
<th>D2</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>5</td>
<td>7</td>
<td>16</td>
<td>36</td>
<td>35</td>
<td>er1</td>
<td>16</td>
<td>22</td>
<td>104</td>
<td>233</td>
<td>100</td>
</tr>
<tr>
<td>p2</td>
<td>6</td>
<td>7</td>
<td>18</td>
<td>30</td>
<td>29</td>
<td>er2</td>
<td>23</td>
<td>57</td>
<td>230</td>
<td>606</td>
<td>173</td>
</tr>
<tr>
<td>p3</td>
<td>6</td>
<td>7</td>
<td>20</td>
<td>30</td>
<td>29</td>
<td>er3</td>
<td>30</td>
<td>90</td>
<td>423</td>
<td>1,280</td>
<td>253</td>
</tr>
<tr>
<td>p4</td>
<td>6</td>
<td>7</td>
<td>22</td>
<td>33</td>
<td>32</td>
<td>er4</td>
<td>36</td>
<td>117</td>
<td>605</td>
<td>1,974</td>
<td>322</td>
</tr>
<tr>
<td>p5</td>
<td>6</td>
<td>7</td>
<td>22</td>
<td>38</td>
<td>37</td>
<td>er5</td>
<td>43</td>
<td>151</td>
<td>854</td>
<td>3,022</td>
<td>410</td>
</tr>
<tr>
<td>sw1</td>
<td>18</td>
<td>54</td>
<td>211</td>
<td>569</td>
<td>534</td>
<td>sw6</td>
<td>55</td>
<td>298</td>
<td>885</td>
<td>3,755</td>
<td>3,704</td>
</tr>
<tr>
<td>sw2</td>
<td>27</td>
<td>97</td>
<td>418</td>
<td>1,348</td>
<td>954</td>
<td>sw7</td>
<td>83</td>
<td>609</td>
<td>1,534</td>
<td>6,180</td>
<td>5,596</td>
</tr>
<tr>
<td>sw3</td>
<td>36</td>
<td>153</td>
<td>733</td>
<td>2,669</td>
<td>1,380</td>
<td>sw8</td>
<td>112</td>
<td>996</td>
<td>2,179</td>
<td>9,096</td>
<td>7,505</td>
</tr>
<tr>
<td>sw4</td>
<td>43</td>
<td>201</td>
<td>1,099</td>
<td>3,870</td>
<td>1,783</td>
<td>sw9</td>
<td>132</td>
<td>1,108</td>
<td>2,645</td>
<td>11,089</td>
<td>8,893</td>
</tr>
<tr>
<td>sw5</td>
<td>51</td>
<td>262</td>
<td>1,373</td>
<td>5,501</td>
<td>2,222</td>
<td>sw10</td>
<td>154</td>
<td>1,888</td>
<td>3,430</td>
<td>14,101</td>
<td>10,329</td>
</tr>
</tbody>
</table>

8.4.2. Cross-comparison. An issue of separate interest is how the greedy algorithms in ColPack for distance-1, acyclic, star, and distance-2 coloring compare against each other in terms of number of colors used and runtime.

Number of colors. Figure 6 shows results on the scientific computing graphs comparing the four coloring variants against each other in terms of number of colors used when the ordering technique is fixed to be SL. A similar comparison for the synthetic graphs is provided in a tabular format in Table V. We highlight a few observations gleaned from the results contained in the figure and the table.

— In terms of number of colors used, acyclic coloring is closer to distance-1 coloring than it is to distance-2 coloring, and star coloring is closer to distance-2 coloring than it is to distance-1 coloring.

— For planar graphs, the acyclic coloring algorithm in ColPack uses about the same number of colors as the distance-1 coloring algorithm (to be exact, the acyclic coloring algorithm used seven colors whereas the distance-1 coloring algorithm used at most six colors). This analogous to the reduction we saw in the distance-1 coloring algorithm (in Section 8.3); in most cases, SL (or ID) ordering gave the fewest colors in each of the coloring variants.

— The degree-2 versions of the ordering techniques do not offer significant reduction in number of colors compared to the degree-1 versions, whereas they incur considerable increase in runtime.
result is in close agreement with theory [Borodin 1979], and shows that the heuristic in ColPack provides nearly optimal solutions.

— Again for planar graphs, the star coloring algorithm in ColPack uses a number of colors in the range between sixteen and twenty-two. The optimal number of colors needed to star color a planar graph is known to lie between ten and twenty [Albertson et al. 2004]. This indicates that the ColPack-algorithm is close to optimal for such graphs.

— The maximum degree $\Delta(G) + 1$ in a graph is a lower bound on the number of colors needed to distance-2 color a graph. As the data in Table V shows, the ColPack algorithm for distance-2 coloring finds optimal solution for planar graphs. Further, for the other three types of test graphs, it can be seen that the ratio between the number of colors used in distance-2 coloring and the quantity $\Delta(G)$ is relatively small (e.g., 1.37 for sw10, 2.48 for sw5, and 7.37 for er5), indicating that the algorithm yields solutions reasonably close to optimal. The corresponding ratios for the scientific computing graphs (Figure 6) range from 1.00 to 1.03, indicating that the solutions for all practical purposes are optimal.

Run time. As mentioned in Section 4, the star, acyclic and distance-2 coloring algorithms in ColPack have nearly the same time complexity, $O(|V| \cdot d^2)$, with varying magnitudes of the constants. Since star and acyclic coloring involve substantial work related to bookkeeping of two-colored subgraphs (more for star coloring than acyclic coloring), their practical runtime
is expected to be higher than that of distance-2 coloring. These expectations are confirmed by the observed execution times depicted in Figure 7, which shows a plot of the total (coloring plus ordering) execution times on the synthetic test graphs for the three coloring problems and the distance-1 coloring problem (whose complexity is $O(|V| \cdot d_1)$) while using the SL ordering in each case.

8.5. Results on coloring bipartite graphs

8.5.1. Partial distance-2 coloring (unidirectional matrix partition). Versions of the ordering techniques LF, DLF, ID and SL tailored for the partial distance-2 coloring of bipartite graphs (on either of the two vertex sets of the bipartite graph) are available in ColPack. Tables VI and VII show results on these ordering techniques when applied to various scientific computing graphs obtained from the University of Florida Matrix Collection. The results show that the ordering techniques reduce the number of colors compared to natural ordering. The benefit due to ordering here is similar to that seen in the distance-1 coloring context presented in Section 8.3. The maximum degree of a vertex—$\Delta(V_1)$ for a distance-2 coloring on $V_2$ and $\Delta(V_2)$ for a distance-2 coloring on $V_1$—is a lower bound on the optimal number of colors needed. One can then see that many of the graphs are colored optimally or nearly optimally. Note in particular that five of the ten graphs in Table VI are colored optimally—in Tables VI and VII names of matrices that are known to have been optimally colored are shown in boldface.

Remarkably, such results are obtained at low runtimes, as can be seen from Table VIII, which shows the runtimes in seconds for column-wise partial distance-2 coloring when the various ordering techniques are used. The coloring and ordering time are shown separately in the table. Note, for example, that the graph cage12, which has more than 2 million edges, is ordered and colored in less than a second using ColPack.

8.5.2. Star bicoloring (bidirectional matrix partition). Experiments we did in the context of star bicoloring (results not included here) showed that for a majority of the test graphs we

---

Table VI. Structural statistics and partial distance 2 coloring (column-wise) results on various scientific computing graphs.

| Name   | $|V_1|$ | $|V_2|$ | $|E|$ | $\Delta(V_1)$ | SL | ID | LF | DLF | N  |
|--------|-------|-------|-----|-----------|----|----|----|-----|----|
| cre_a  | 3,516 | 7,248 | 18,168 | 360 | 360 | 360 | 360 | 360 | 360 |
| ken_11 | 14,694 | 21,349 | 49,058 | 122 | 125 | 124 | 126 | 122 | 130 |
| ken_13 | 28,632 | 42,659 | 97,246 | 170 | 171 | 171 | 174 | 170 | 176 |
| maros_r7 | 3,136 | 9,408 | 144,848 | 48 | 83 | 90 | 70 | 114 | 74 |
| cre_d  | 8,926 | 73,948 | 246,614 | 808 | 808 | 808 | 808 | 808 | 813 |
| ken_18 | 105,127 | 154,699 | 358,614 | 325 | 326 | 328 | 325 | 330 | 326 |
| af23560 | 23,560 | 23,560 | 460,598 | 21 | 42 | 42 | 43 | 59 | 44 |
| e40r0100 | 17,281 | 17,281 | 553,562 | 62 | 71 | 71 | 78 | 85 | 95 |
| cage11 | 39,082 | 39,082 | 559,722 | 31 | 64 | 67 | 70 | 70 | 81 |
| cage12 | 130,228 | 130,228 | 2,032,536 | 33 | 67 | 72 | 73 | 79 | 96 |

Table VII. Structural statistics and partial distance 2 coloring (row-wise) results on various scientific computing graphs.

| Name   | $|V_1|$ | $|V_2|$ | $|E|$ | $\Delta(V_2)$ | SL | ID | LF | DLF | N  |
|--------|-------|-------|-----|-----------|----|----|----|-----|----|
| cre_a  | 3,516 | 7,248 | 18,168 | 14 | 14 | 14 | 14 | 14 | 16 |
| ken_11 | 14,694 | 21,349 | 49,058 | 3 | 4 | 4 | 5 | 5 | 5 |
| ken_13 | 28,632 | 42,659 | 97,246 | 3 | 4 | 5 | 5 | 5 | 4 |
| maros_r7 | 3,136 | 9,408 | 144,848 | 46 | 80 | 88 | 100 | 113 | 72 |
| cre_d  | 8,926 | 73,948 | 246,614 | 13 | 15 | 15 | 13 | 14 | 15 |
| ken_18 | 105,127 | 154,699 | 358,614 | 3 | 4 | 5 | 5 | 5 | 5 |
| af23560 | 23,560 | 23,560 | 460,598 | 21 | 42 | 43 | 43 | 60 | 43 |
| e40r0100 | 17,281 | 17,281 | 553,562 | 62 | 71 | 71 | 78 | 85 | 87 |
| cage11 | 39,082 | 39,082 | 559,722 | 31 | 64 | 67 | 70 | 70 | 81 |
| cage12 | 130,228 | 130,228 | 2,032,536 | 33 | 67 | 72 | 73 | 79 | 96 |
Table VIII. partial distance 2 coloring (column-wise) runtime results. Coloring time (ct) and ordering (ot) in seconds are shown separately in each case.

<table>
<thead>
<tr>
<th>Name</th>
<th>ct</th>
<th>SL</th>
<th>ID</th>
<th>ct</th>
<th>LF</th>
<th>ID</th>
<th>ct</th>
<th>LF</th>
</tr>
</thead>
<tbody>
<tr>
<td>cre.a</td>
<td>0.0031</td>
<td>0.0112</td>
<td>0.0030</td>
<td>0.0148</td>
<td>0.0035</td>
<td>0.0026</td>
<td>0.0038</td>
<td>0.0128</td>
</tr>
<tr>
<td>ken.11</td>
<td>0.0041</td>
<td>0.0186</td>
<td>0.0042</td>
<td>0.0216</td>
<td>0.0040</td>
<td>0.0044</td>
<td>0.0076</td>
<td>0.0242</td>
</tr>
<tr>
<td>ken.13</td>
<td>0.0105</td>
<td>0.0466</td>
<td>0.0101</td>
<td>0.0548</td>
<td>0.0099</td>
<td>0.0107</td>
<td>0.0196</td>
<td>0.0612</td>
</tr>
<tr>
<td>maros_r7</td>
<td>0.0240</td>
<td>0.0553</td>
<td>0.0259</td>
<td>0.0593</td>
<td>0.0239</td>
<td>0.0180</td>
<td>0.0304</td>
<td>0.0671</td>
</tr>
<tr>
<td>cre.d</td>
<td>0.1860</td>
<td>0.8470</td>
<td>0.1886</td>
<td>0.9380</td>
<td>0.1871</td>
<td>0.1698</td>
<td>0.2243</td>
<td>0.9585</td>
</tr>
<tr>
<td>ken.18</td>
<td>0.0684</td>
<td>0.3486</td>
<td>0.0672</td>
<td>0.3903</td>
<td>0.0667</td>
<td>0.0709</td>
<td>0.1377</td>
<td>0.4387</td>
</tr>
<tr>
<td>af23560</td>
<td>0.0418</td>
<td>0.1101</td>
<td>0.0391</td>
<td>0.1052</td>
<td>0.0405</td>
<td>0.0373</td>
<td>0.0512</td>
<td>0.1308</td>
</tr>
<tr>
<td>e40r0100</td>
<td>0.0954</td>
<td>0.1747</td>
<td>0.0953</td>
<td>0.1737</td>
<td>0.0972</td>
<td>0.0532</td>
<td>0.1022</td>
<td>0.1945</td>
</tr>
<tr>
<td>cage11</td>
<td>0.0559</td>
<td>0.1656</td>
<td>0.0549</td>
<td>0.1718</td>
<td>0.0531</td>
<td>0.0446</td>
<td>0.0748</td>
<td>0.1951</td>
</tr>
<tr>
<td>cage12</td>
<td>0.2250</td>
<td>0.6738</td>
<td>0.2215</td>
<td>0.6967</td>
<td>0.2126</td>
<td>0.1739</td>
<td>0.3602</td>
<td>0.8602</td>
</tr>
</tbody>
</table>

considered, the variant called ImplicitCoveringStarBicoloring when used together with DLF ordering gave the fewest colors compared to the other combinations of the three star bicoloring algorithms and orderings available ColPack. The possible number of combinations here is quite large, and further study needs to be done to identify an algorithm that works well across a wide spectrum of structures.

9. CONCLUSION

We introduced ColPack, a software package comprising primarily of algorithms for a variety of graph coloring problems. The package supports a total of six different kinds of coloring problems defined on general graphs and three kinds of coloring problems defined on bipartite graphs. Many of the coloring problems model matrix partitioning needs arising in compression-based computation of sparse Jacobian and Hessian matrices using Automatic Differentiation—both direct and indirect methods are covered. The package also provides an additional functionality needed in such a context, a set of algorithms for recovering the entries of a derivative matrix from a compressed representation. The package has been interfaced with the Automatic Differentiation tool ADOL-C, and the combined toolkit has been used to solve optimization problems in liquid purification techniques used in the chemical industry and in management of electric power flow in networks [Gebremedhin et al. 2008; Gebremedhin et al. 2009].

Several of the coloring problems supported in ColPack also have applications in many areas beside derivative computation. A few examples include: scheduling problems, facility location problems, frequency assignment problems, concurrency discovery in parallel computing, and compiler design.

A common feature of the algorithms in ColPack for the various coloring problems, which are all known to be NP-hard, is that the algorithms process vertices sequentially one at a time, making a color choice that is not reversed at a later stage. The algorithms for the more intricate coloring problems maintain a collection of two-colored induced subgraphs to determine quickly an allowable color for a vertex.

A significant part of the work in this paper has been the investigation of vertex ordering techniques that are effective for greedy algorithms for the various coloring problems. We provided a fresh characterization for several known degree-based ordering techniques, adapted the techniques to specialized coloring problems, introduced new ordering techniques, and discussed how they are all efficiently implemented in ColPack. We used ColPack to conduct an experimental study on the performance of the vertex ordering techniques in the context of the various greedy coloring algorithms using real-world as well as synthetically generated graphs. We showed that the greedy algorithms in combination with good ordering techniques yield solutions that are near-optimal fast—linear-time in the case of distance-1 coloring.
In related efforts, parallel algorithms have been developed for some of the coloring problems in ColPack. In particular, a family of scalable distributed-memory distance-1 coloring algorithms have been developed in [Bozda˘g et al. 2008]. Under a similar framework, distributed-memory parallel algorithms for distance-2 and restricted star coloring of general graphs and partial distance-2 coloring of bipartite graphs have been developed in [Bozda˘g et al. 2010]. In an ongoing work, we are developing multithreaded versions for various coloring and ordering algorithms to take advantage of the computational power emerging multicore architectures offer even at the desktop level. Recent activities along this line include the works [Patwary et al. 2011; Catalyurek et al. 2012]. A major direction we intend to pursue in future work is to enable sparse derivative computation on multicore architectures. Our long-term goal is to develop multithreaded versions of many of the capabilities of ColPack.

ACKNOWLEDGMENTS

Arijit Tarafdar contributed to the development of ColPack while he was a student at Old Dominion University. We thank Fredrik Manne for his comments on an earlier version of this paper. We are grateful for the comments the anonymous referees provided, which greatly helped us improve the content and presentation of the paper.

REFERENCES


Appendix

Below are two examples illustrating how ColPack can be used. The first example shows how ColPack, along with an appropriate AD tool (in this case ADOL-C), could be used in computing a sparse Jacobian following the four-step framework outlined in Algorithm 1. The second example illustrates a simple use of one of the coloring capabilities of ColPack for any purpose.

```cpp
#include "ColPackHeaders.h"
using namespace ColPack;
using namespace std;

// EXAMPLE 1
int main(){
    // Step 1: Determine sparsity structure of the Jacobian (done by an AD tool)
    unsigned int *** uip3_SparsityPattern = new unsigned int **;
    int i_rowCount, i_columnCount;
    /* obtain uip3_SparsityPattern, i_rowCount and i_columnCount (from an AD tool) */
    // Step 2: Obtain the seed matrix via coloring
    double*** dp3_Seed = new double**;
    int *ip1_SeedRowCount = new int;
    int *ip1_SeedColumnCount = new int;
    BipartiteGraphPartialColoringInterface *g = new BipartiteGraphPartialColoringInterface(SRC_MEM_ADOLC, uip3_SparsityPattern, i_rowCount, i_columnCount);
    g->PartialDistanceTwoColoring("SMALLEST_LAST", "Column_Partial_Distance_Two");
    *dp3_Seed = g->GetSeedMatrix(ip1_SeedRowCount, ip1_SeedColumnCount);
    // Step 3: Obtain the Jacobian-seed matrix product (done by an AD tool)
    double*** dp3_CompressedMatrix = new double**;
    /* obtain dp3_CompressedMatrix using an AD tool */
    // Step 4: Recover the numerical values of the original matrix from the compressed representation
    double*** dp3_JacobianValue = new double**;
    JacobianRecovery1D* jr = new JacobianRecovery1D;
    jr->RecoverD2Cln_RowCompressedFormat(g, *dp3_CompressedMatrix, *uip3_SparsityPattern, dp3_JacobianValue);
    /* Deallocate memory using functions in Utilities/MatrixDeallocation.h */
    return 0;
}

// EXAMPLE 2
int main(){
    string s_inputFile = "example.mtx";
    GraphColoringInterface *g = new GraphColoringInterface(SRC_FILE, s_inputFile, "MM");
    g->SmallestLastOrdering();
    printf("Max back degree: %d\n", g->DistanceOneColoring());
    printf("Nr colors: %d\n", g->GetVertexColors());
    delete g;
    return 0;
}
```

Fig. 8. Examples illustrating the usage of ColPack.