SIAM J. COMPUT.
Vol. 36, No. 2, pp. 326–353

CERTIFYING ALGORITHMS FOR RECOGNIZING INTERVAL GRAPHS AND PERMUTATION GRAPHS*

DIETER KRATSCH†, ROSS M. MCCONNELL‡, KURT MEHLHORN§, AND JEREMY P. SPINRAD¶

Abstract. A *certifying algorithm* for a problem is an algorithm that provides a certificate with each answer that it produces. The certificate is a piece of evidence that proves that the answer has not been compromised by a bug in the implementation. We give linear-time certifying algorithms for recognition of interval graphs and permutation graphs, and for a few other related problems. Previous algorithms fail to provide supporting evidence when they claim that the input graph is not a member of the class. We show that our certificates of nonmembership can be authenticated in $O(|V|)$ time.

Key words. certificates, certifying algorithms, interval graphs, permutation graphs

AMS subject classifications. 68W40, 05C85, 68N30

DOI. 10.1137/S0097539703437855

1. Introduction.

A recognition algorithm is an algorithm that decides whether some given input (graph, geometrical object, picture, etc.) has a certain property. Such an algorithm *accepts* the input if it has the property or *rejects* it if it does not. A *certifying algorithm* for a decision problem is an algorithm that provides a certificate with each answer that it produces. The certificate is a piece of evidence that proves that the answer has not been compromised by a bug in the implementation.

We give linear-time certifying algorithms for recognition of interval graphs and permutation graphs. Previous algorithms fail to provide supporting evidence of nonmembership. We show that our certificates of nonmembership can be authenticated in $O(n)$ time, where n is the number of vertices.

A familiar example of a certifying recognition algorithm is a recognition algorithm for bipartite graphs that computes a 2-coloring for bipartite input graphs and an odd cycle for nonbipartite input graphs. A more complex example is the linear-time planarity test which is part of the library of efficient data structures and algorithms (LEDA) system [19, section 8.7]. It computes a planar embedding for planar input graphs and a *Kuratowski subgraph* (a subdivision of K_5 or $K_{3,3}$) for nonplanar input graphs.

Certifying versions of recognition algorithms are highly desirable in practice; see [28, 20, 21] and [19, section 2.14] for general discussions on result checking. Consider a planarity testing algorithm that produces a planar embedding if the graph is planar, and simply declares it nonplanar otherwise. Though the algorithm may have been proven correct, the implementation may contain bugs. When the algorithm

*Received by the editors November 23, 2003; accepted for publication (in revised form) August 12, 2005; published electronically June 19, 2006.
http://www.siam.org/journals/sicomp/36-2/43785.html

†Université de Metz, LITA, 57045 Metz Cedex 01, France (kratsch@sciences.univ-metz.fr).
‡Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873 (rmm@cs.colostate.edu).
§Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany (mehlhorn@mpi-sb.mpg.de).
¶Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235 (spin@vuse.vanderbilt.edu).

declares a graph nonplanar, there is no way to check whether it did so because of a bug.

Given the reluctance of practitioners to assume on faith that a program is bug-free, it is surprising that the theory community has often ignored the question of requiring an algorithm to certify its output, even in cases when the existence of adequate certificates is well known.

An *authentication algorithm* is an algorithm that check the validity of the certificate. In contrast to the case of an algorithm that solves the problem from scratch, an authentication algorithm may reject the certificate if it is invalid, leaving open the answer to the original problem. However, it must be the case that the authentication algorithm provides a correct answer to the original problem if the certificate is valid, and never declares an incorrect answer to the problem due to an invalid certificate. The certificate has practical value when the algorithm for solving the problem from scratch is difficult but the authentication algorithm is trivial.

For instance, in the case of planarity testing, the authentication algorithm cycles through the edges of the Kuratowski subgraph, verifying that they are, in fact, present in the graph and form a Kuratowski subgraph as claimed. Though planarity testing in linear time is a complicated problem, checking a Kuratowski subgraph is a trivial task, especially if the edges are supplied in a convenient order by the certificate.

The notion of a certifying algorithm is related to the concept of run-time checking. Typically, run-time checking has been applied to very small fragments of code; for example, a program to calculate square roots might check at run-time that the square of the result equals the input value. An attempt to create a corresponding model of run-time checking for more complex algorithms is given in [28] with motivations which are very similar to our own. The authors conceive of a method for checking whether a (possibly complex) program that computes a function has produced a correct output. Their model does not require an algorithm to provide supporting evidence with its answer, but subsequent calls to it are allowed in order to present it with nondeterministically generated challenges after it has already returned a solution. If it has answered incorrectly, it can pass with only a certain probability of success. Confidence in the answer to within any desired probability can be attained through a sufficiently large set of challenges.

An important distinction between run-time checking and certifying algorithms is that run-time checking is generally conceived as being separate from the process of producing the output, while certifying algorithms produce proofs of correctness together with their output.

To clarify the difference between these two approaches, let us consider two valid, but opposing, views on proofs of output correctness. One can take the view that it is desirable to have a mechanism that uses as little information as possible in verifying the correctness of the output, preferably only the standard format of the solution. This approach has the advantage of enabling a user to check, using a single program, the correctness of the output of any program that has been designed to solve the problem.

Papers such as [1] have devised clever mechanisms of this form for verifying that the output of any program that solves a diverse set of problems, including such problems as graph isomorphism and equivalence search, is correct. If such a mechanism exists, one can take the position that there is no reason for the creator of the program to work on a technique for verifying that the output is correct, since any errors will be caught in a separate checking phase.

We instead want to shift the burden of creating proof of correctness to the software

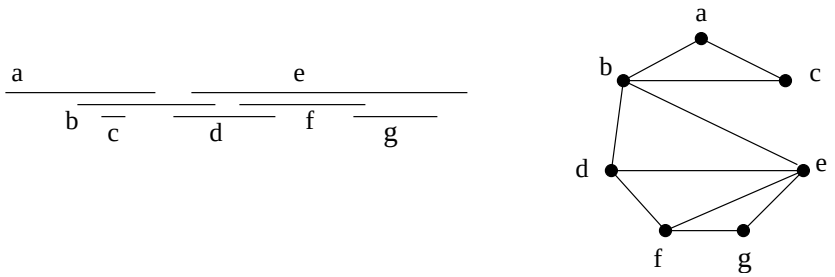

FIG. 1. *An interval graph is the intersection graph of a set of intervals on a line. The graph has one vertex for each of the intervals, and two vertices are adjacent if the corresponding intervals intersect.*

engineering phase as part of the design of the algorithm for solving the problem upon which to base the program. In finding the solution to the problem, the algorithm often goes through steps that can be used to create a simple certificate of correctness of the output. We believe that it should be part of the routine process of algorithm design to look for this certificate, and provide one wherever possible, rather than leaving this to a separate checking phase.

An interval graph is the intersection graph of intervals on a line (see Figure 1). That is, each vertex corresponds to an associated interval, and two vertices are adjacent iff the corresponding intervals intersect. The intervals constitute an *interval model* of the graph. Interval graphs have applications in molecular biology and scheduling.

Several linear-time recognition algorithms for interval graphs are known [2, 4, 9, 11, 12, 13, 18]. These graphs come up in the context of a variety of problems in scheduling and molecular biology; see [8, 24] for surveys. When a graph is an interval graph, these algorithms produce a certificate in the form of an interval model. When a graph is not an interval graph, none of these algorithms provides a certificate. However, the existence of certificates of rejection in the form of a forbidden substructure characterization is also well known. We extend the linear-time algorithm of Korte and Möhring [13] so that it also produces a certificate when a graph is not an interval graph; see section 5.

A permutation graph is the graph of inversions in a permutation. That is, each vertex corresponds to an element of the ground set of a permutation, and two vertices are adjacent iff the permutation reverses the relative order of the two corresponding elements (see Figure 2). If a graph is a permutation graph, we may show this by giving a *permutation model*, which consists of two linear orderings $(v_1, v_2, \ldots, v_n)$ and $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$ of the vertices, such that two vertices v_i and v_j are adjacent iff v_i is before v_j in exactly one of the orderings.

The only previous linear-time algorithm for recognizing permutation graphs is given in [18]. This algorithm produces a permutation model if the graph is a member of the class, and presents its failure to produce such a model as the only evidence that a graph is not a member of the class. We give a linear-time algorithm that produces a certificate also in the case of nonmembership; see section 6. The algorithm is based on the following connection to comparability graphs. A graph G is a permutation graph iff G and its complement $\overline{G}$ are comparability graphs.

A dag is *transitive* if, whenever (a, b) and (b, c) are directed edges of the dag, (a, c) is also a directed edge. A transitive dag is a graphical representation of a

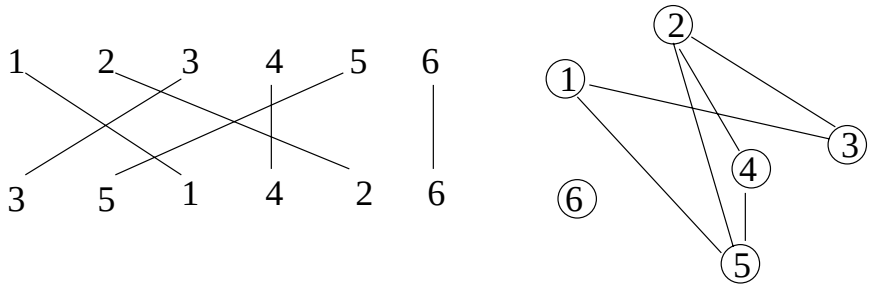

FIG. 2. *When a permutation acts on a sequence of elements, the corresponding permutation graph has one vertex for each of the elements. Two vertices are adjacent if the permutation swaps the relative order of the two elements.*

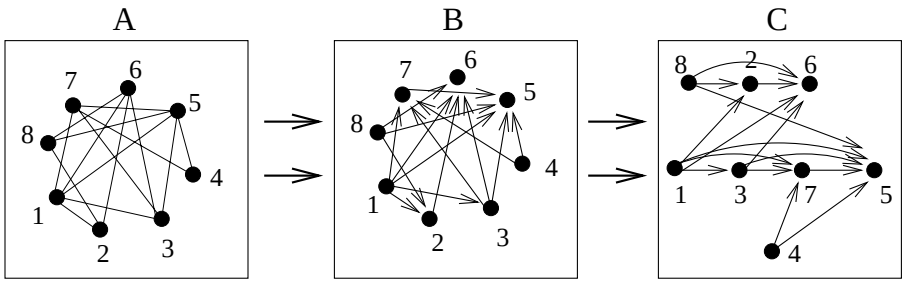

FIG. 3. *A comparability graph is a graph whose edges can be oriented so that the resulting digraph is acyclic and transitive. Part A depicts a comparability graph, part B is a transitive orientation, and part C is a redrawing of part B that makes it easy to see that the result is transitive.*

partial order (poset) relation. A graph is a *comparability graph* if orientations can be assigned to its edges so that the resulting digraph is a transitive dag (see Figure 3). Such an orientation is called a *transitive orientation*. A transitive orientation of a comparability graph can be found in linear time [18].

Gallai gave a forbidden substructure characterization of comparability graphs [7]. Since a graph G is a permutation graph iff G and $\overline{G}$ are comparability graphs, this forbidden structure in G or in $\overline{G}$ serves as a certificate that a graph is not a comparability graph. We do not know how to obtain this certificate in linear time for noncomparability graphs; no linear-time recognition algorithm is known. However, we show that when G is not a permutation graph, we may produce this certificate for G or for $\overline{G}$ in linear time. The algorithm fails to give a linear-time certifying algorithm for comparability graphs only because we cannot control whether the algorithm will find the certificate in G or in $\overline{G}$ when both G and $\overline{G}$ fail to be comparability graphs.

Certifying algorithms also exist for optimization problems; linear programming duality is a prime example. Primal and dual optimal solutions of a linear program certify each other. A *proper coloring* of a graph is a coloring of the vertices such that no adjacent vertices have the same color, and it is *minimum* if there is no proper coloring that uses fewer colors. A linear-time algorithm is known for finding a maximum clique and a minimum proper coloring of a comparability graph [18]. In a comparability graph, the size of the maximum clique is always the same as the minimum number of colors in a proper coloring. Since the vertices of a clique must all be colored differently in any proper coloring, the clique is a certificate of minimality of the coloring, and the coloring is a certificate that the clique is of maximum size. We give an algorithm that

provides a certificate that the input graph is not a comparability graph whenever this algorithm fails to find the desired coloring and clique.

2. Preliminaries. We consider only finite, undirected, and simple graphs. Let $G = (V, E)$ be a graph. We let n denote the number of vertices and m denote the number of edges.

For $W \subseteq V$ we denote by $G[W]$ the subgraph of G induced by W and we write $G - W$ instead of $G[V - W]$. The neighborhood of v is $N(v) = \{u \in V | uv \in E\}$ and $N[v] = N(v) \cup \{v\}$. If G is a directed graph, $N^-(v) = \{u \in V | (u, v) \in E\}$ and $N^+(v) = \{u \in V | (v, u) \in E\}$.

If $P = (v_1, v_2, \ldots, v_k)$ is a path, $u \in N(v_1)$, and $w \in N(v_k)$, then uP denotes the path $(u, v_1, v_2, \ldots, v_k)$, and Pw denotes the path $(v_1, v_2, \ldots, v_k, w)$. A P_4 is an induced subgraph on four vertices that is a path.

For an arc $e = (a, b)$, let its *transpose* e^T denote (b, a).

2.1. Representation of graphs. Analysis of the running time of an algorithm requires an agreement between the user and the designer of an algorithm about an appropriate format for the input data. No graph algorithm can claim an $O(n + m)$ running time if the input is provided in the form of an adjacency matrix, so a consensus has arisen that it is reasonable to require that the input be an adjacency list representation of the graph. If the user must spend $\Theta(n^2)$ time converting an adjacency matrix representation to this form before giving it as an input to the graph algorithm, this cost is not attributed to the running time of the graph algorithm.

Let us view a graph as an abstract data type that supports the following queries, possibly in addition to other queries:

- `Neighbors(`x`):` Given a vertex x, find its neighbors.
- `Adjacency(`x, y`):` Given two vertices, x and y, determine whether they are adjacent.

Algorithm texts usually contain a discussion of the trade-offs in space and time bounds for these two operations that arise in choosing an adjacency matrix representation or an adjacency list representation of a graph. The list of neighbors of a vertex in an adjacency list representation is typically unordered. This gives an extremely poor time bound for the `Adjacency` operation ($\Theta(n)$ in the worst case as opposed to $O(1)$ for the adjacency matrix).

This can easily be remedied by assuming that the adjacency lists are represented by a standard representation of an ordered set, such as a sorted array or a balanced binary search tree. This improves the time for the `Adjacency` query from $\Theta(n)$ in the worst case to $O(\log n)$, without increasing the space requirements or the time bound for the `Neighbors` operation. Moreover, it is possible to convert the unsorted variant to the sorted variant in $O(n + m)$ time by labeling the vertices with identifiers from 1 to n and then radix sorting the set of all edges according to the identifiers of their endpoints.

The only disadvantages to adopting this *ordered adjacency list representation* as a standard representation for sparse graphs seem to be cases where the data type must also support dynamic graph operations such as insertion of edges or contraction of vertices. Maintaining the sorted variant of the data structure can add an $O(\log n)$ factor to the cost of such operations. However, most graph algorithms deal with static graphs, and in the cases where the costs of these dynamic operations are an issue, an algorithm can ignore the ordering in the input data and neglect to maintain the invariant that it remain ordered as it executes.

Why has this ordered adjacency list representation not been adopted as a standard representation over the seemingly less desirable unordered one? This is probably because any algorithm that makes repeated use of the `Adjacency` query can produce the ordered representation from the unordered one in linear time before beginning work on the problem. Since a graph algorithm is assumed to run in $\Omega(n+m)$ time, the issue has been deemed inconsequential to the analysis of running times.

The possibility of graph algorithms that run in sublinear time, however, requires a reexamination of this issue. In contrast to algorithms that solve a problem on a graph from scratch, it is often the case that authentication algorithms can run in sublinear time.

An example of this occurs in connection with the class of *cographs*, which are the class of graphs that have no induced subgraph that is a P_4. That is, they have no induced subgraph that consists of exactly three edges forming a path (v_1, v_2, v_3, v_4) on four vertices. It takes $O(n+m)$ time to determine whether a given graph G is a cograph without the aid of a certificate [5]. However, given a legitimate P_4 that has been pointed out by a distrusted source, it takes $O(n)$ time to verify it on an unordered adjacency list representation, and $O(\log n)$ time on the ordered representation, since the `Adjacency` query takes $O(\log n)$ time. We therefore have a case where the choice of an unordered adjacency list representation versus an ordered one makes a difference in the analysis of the time bound of an algorithm.

We think that an ordered adjacency list representation would be a reasonable alternative to the unsorted one as a general-purpose representation of sparse graphs. In any case, there are compelling reasons to use the ordered representation when analyzing sublinear algorithms, and we will assume this representation throughout the paper. To be specific, since we deal only with problems involving static graphs, let us assume that each adjacency list is implemented as a sorted array of neighbors.

Before leaving this topic, we should mention that although the `Adjacency` operation takes $O(\log n)$ time, we can improve this to $O(1)$ by supplying what amounts to a certificate of the answer to the `Adjacency` query. If the answer to `Adjacency`(x, y) is `true`, the certificate consists of the location of the edge in the data structure, and if the answer is `false`, the certificate consists of the location where it would appear if it occurred. In either case, the user can check the result in $O(1)$ time. The certificate of a `false` answer cannot be made to work in this way on an unordered adjacency list representation.

Given this, it is easy to see that verifying the presence of a P_4 in a graph can be improved to $O(1)$ by building a certificate out of a collection of "subcertificates" for individual `Adjacency` queries on all pairs of elements of $\{v_1, v_2, v_3, v_4\}$. The same technique is useful for checking any induced subgraph that has been pointed out as a certificate of an answer to a graph problem.

3. What constitutes a certificate. Since software that generates a certificate could have a bug, a proposed certificate must be authenticated by verifying that it does, in fact, prove the result. For instance, if an odd cycle in a graph is presented as a certificate that the graph is not bipartite, authentication consists of verifying that it is a cycle, it has odd length, and that the claimed edges occur in the graph. The cycle can be given as a sequence of pointers to its edges in the input data structure, and takes $O(n)$ time for the user to authenticate, which is better than the $O(n+m)$ bound for checking whether a graph is bipartite.

A good certificate has an authentication algorithm that is conceptually simpler than algorithms for the original problem, or has a better time bound, or has both. If

the authentication step is simple and efficient enough, it may be possible to perform the check by visual inspection.

When the certificate is checked automatically, determining reliability of the implementation of the authentication algorithm is an obvious goal. Otherwise, consider the following scenario. The implementations of the certifying algorithm and of the authentication algorithm are both faulty. The certifying algorithm produces both an erroneous answer to the decision problem and an erroneous certificate, while the faulty authentication algorithm then claims that the certificate proves the given answer. The user is led to believe an erroneous conclusion.

A certificate is *sublinear* if its authentication algorithm has a tighter time bound than a linear one. For instance, for a problem where arbitrary graphs can be given as input, a certificate with an $O(n)$ authentication algorithm is sublinear, since this bound is never worse than linear and is often better. A certificate is *strong* (with respect to current algorithmic knowledge) if its authentication algorithm has a better time bound than that of the current fastest algorithm that solves the problem without a certificate. It is *weak* if it takes the same time to authenticate as it does to solve the original problem without the certificate, but may have other advantages such as greatly simplifying the task conceptually.

One of the anonymous referees has inquired about the possibility of certifying algorithms or authentication algorithms that require asymptotically more time or space than a noncertifying algorithm. Given the importance of software reliability in some industrial settings, the advantages of using these algorithms might outweigh the extra costs. We have not yet found interesting examples to support this, however.

Some of these notions are related to concepts that come up in the theory of NP-completeness. The class NP of decision problems are those for which, whenever the answer is *true*, this answer can be confirmed in polynomial time if one is supplied an appropriate certificate. The question of whether this is always possible without a certificate is the famous question of whether P = NP. The notion of a certificate and the time bound that it makes possible provides a precise mathematical definition of the class NP.

A certifying algorithm that returns a strong or sublinear certificate can be distinguished on objective mathematical grounds from a noncertifying algorithm. To be able to claim that certifying algorithms are a formal class of algorithms, we could require all certifying algorithms to produce sublinear certificates. Unfortunately, this restriction excludes many algorithms that produce certificates that are clearly useful in practice.

In addition, though sublinear certificates appear to be common, if the rejection certificate is sublinear, the acceptance certificate often fails to be sublinear, and if the acceptance certificate is sublinear, the rejection certificate often fails to be sublinear. For instance, the problem of recognizing bipartite graphs has a sublinear rejection certificate (an odd cycle) and a weak acceptance certificate (a 2-coloring). Recognizing connected graphs has a sublinear acceptance certificate (a spanning tree) and a weak rejection certificate (a cut $\{V', V - V'\}$ that has no edges across it). Directed acyclic graphs have a weak acceptance certificate (a topological sort) and a sublinear rejection certificate (a directed cycle). A similar phenomenon is seen in interval graphs and permutation graphs where the rejection certificate can be checked in $O(n)$ time, whereas the acceptance certificate is an $O(n)$ representation of G that defines the graph class and that requires $O(n + m)$ time to verify that it faithfully represents G.

What constitutes a weak certificate lacks the formal criteria that define sublinear or strong certificates. A weak certificate has value if it dramatically simplifies the task of solving the problem, without necessarily yielding a better time bound. There is no satisfactory formal measure of the conceptual difficulty of an algorithm, though differences are often obvious.

The value of proposed weak certificates must therefore be debated on a case-by-case basis in much the way that the ill-defined notion of "significance" of any new mathematical result is debated and evaluated through the peer-review process. The certificates in our examples were recognized previously as having theoretical relationships to the problems in question, independently of their use in algorithm design or error checking.

We would therefore describe certifying algorithms as embodying an algorithm-design philosophy, rather than a formal class of algorithms. We show through examples that the approach has been largely overlooked by both algorithm designers and software engineers, and we argue that it can have significant economic impact when algorithms are implemented. Moreover, in the case of graph algorithms, there is an extensive literature on forbidden subgraph characterizations of graph classes that promises to be a rich source of potential certificates for future work on certifying variants of existing graph algorithms.

3.1. Proofs of correctness of authentication algorithms. An interesting question raised by one of the anonymous referees of this paper is what relevance the simplicity of the proof of the authentication algorithm should have. Suppose the input is x, the correct output is $y = f(x)$, and the certificate, w, proves that $y = f(x)$. The proof that the authentication algorithm accepts only valid triples (x, y, w) should be trivial. The proof that the existence of such a triple proves that $y = f(x)$ does not need to be easy for the certificate to be useful.

Consider the question of deciding whether a graph is planar. A noncertifying algorithm for solving this problem inputs a graph G and outputs a bit $y = f(G)$ that is equal to 1 iff the graph is connected and planar. A certifying algorithm for planarity testing implemented in the LEDA package returns a Kuratowski subgraph when the input graph is nonplanar. A Kuratowski subgraph K is difficult to find, but easy to check once it is pointed out. Therefore, the proof that the authentication algorithm accepts only valid triples $(G, 0, K)$ is trivial. However, that K proves that $f(G) = 0$ is also easy to understand. It is not necessary for a user to understand why such a K appears in every nonplanar graph in order to understand that it proves that such an instance of a graph is nonplanar and that the program has answered truthfully.

What happens when G is planar gives a better illustration of the point raised by the referee. In this case, LEDA returns a *planar combinatorial embedding*. The validity of a planar combinatorial embedding is also easy to authenticate once it is pointed out, but it is more difficult to explain why it proves that the graph is planar.

For simplicity, assume for the moment that G is connected. Each undirected edge uv of G can be viewed as consisting of two directed edges (u, v) and (v, u); let $twin((u, v)) = (v, u)$. A planar embedding of G induces a cyclic order on the directed edges directed out of each vertex. Let $\pi_v((u, v))$ be the permutation of the edges that maps in clockwise order each edge out of a vertex to the next edge out of the vertex. Clearly, π_v has one cycle for each vertex. Given π_v, let $\pi_f((u, v)) = \pi_v(twin((u, v))$; it is easy to see that this is a permutation that maps each edge to the next edge about the face that lies immediately to the left of the edge. Therefore, π_f has a cycle for each face.

An abstraction of this is a *combinatorial embedding* of G, which consists of an *arbitrary* cyclic order π_v of the directed edges emanating from each vertex, but which may or may not correspond to the cyclic orders induced by any embedding of G in the plane. This nevertheless defines a second permutation $\pi_f((u,v)) = \pi_v(twin((u,v)))$ as before. By a well-known theorem of Euler, a combinatorial embedding is planar iff $2c - m - n - f = 0$, where f is the number of cycles π_f, c is the number of connected components of G, m is the number of edges, and n is the number of vertices.

To present this certificate, it is necessary only to give π_v as a cyclic order of edges out of each vertex and a pointer from each directed edge (u,v) to its twin (v,u). An authentication algorithm must verify that π_v induces a cyclic order on the edges directed out of each vertex, count the cycles in π_v and π_f, and verify that Euler's relation applies.

Though this test is simple, the proof that it implies that G is planar is not easy to explain to an unsophisticated programmer or user. (See [14] for a typical proof.) The usefulness of the certificate lies in the fact that Euler's formula applies to all instances of the problem and has received the thorough scrutiny of many experts. A person who trusts the theorem does not need to understand its proof in order to make use of the certificate. The certificate is useful because it allows trust in a well-known theorem to be substituted for trust in a program of dubious origin.

3.2. Preconditions and postconditions. One can imagine programs where different subproblems are solved by procedures that produce certificates, and where the program contains embedded code to authenticate each certificate before proceeding.

Suppose that an $O(\log n)$ binary search procedure is asked to search a sorted array of integers for an element i, and that the procedure returns with the answer that i does not occur in the array. It could be the case that the array was not correctly sorted, causing the binary search procedure to falsely declare that i does not occur in it, even though the binary search procedure is correctly implemented. Checking for this error would take $O(n)$ time, which would obviate the advantages of using a binary search procedure instead of a linear search.

Instead of this, we could define the binary search procedure as one that has a precondition that the input array be sorted *and a postcondition that its answer is correct whenever the precondition is met.* An error has occurred in the binary search procedure only if the precondition is met and the returned answer is incorrect.

Let us formally define an *error* in the binary search procedure to be a circumstance where the precondition is met but the returned value is incorrect. As in the case of all certifying algorithms, the question of whether a procedure has erred in some circumstance is separated from the question of whether it contains bugs. The binary search procedure can provide a certificate that it has not erred by returning either the index where i occurs, or else the index where i would occur if it is absent in the array. If the procedure claims that i occurs, the authentication algorithm checks the location to make sure that it is there. If the procedure claims that i does not occur, the authentication algorithm checks the location to make sure that the element at that position is smaller than i and the element at the next position is larger i. (Trivial special cases occur when the index is the last in the array.)

Though this example is trivial, it illustrates the possibility of designing certificates that show that *either* the returned value is correct *or* that the preconditions were not met, without distinguishing which of these cases occurred. This exonerates a procedure as the ultimate source of an error.

As a nontrivial example of such a certificate, the transitive orientation algorithm of [18] finds a transitive orientation of a comparability graph in $O(n+m)$ time. If a graph is given to the procedure that is not a comparability graph, then no transitive orientation exists, and the procedure produces an orientation in $O(n+m)$ time that is not transitive, without recognizing this. Various nonlinear bounds for checking whether an orientation is transitive are known, such as $O(nm)$, $O(m^{3/2})$, and the time to multiply two $n \times n$ matrices, $O(n^{2.376})$. These are the best bounds for recognizing comparability graphs.

However, it is reasonable to view as a precondition the requirement that the input to the transitive orientation algorithm be a comparability graph, since the orientation is meaningless if this is not the case. Below, we show how to provide a certificate, called an *orientation tree*, that shows that either the precondition has not been met or the returned orientation is transitive. This certificate is quite simple to check in $O(n+m)$ time.

4. Chordal bipartite graphs. A *chord* in a simple cycle is an edge that is not an edge of the cycle, but whose endpoints are both vertices in the cycle. In a bipartite graph, every cycle is even, so every cycle has length at least four. A graph is *chordal bipartite* if it has no chordless cycle of length greater than four.

Chordal bipartite graphs provide an easy introduction to techniques that we will use to produce certifying algorithms. Lubiw [16] gives an $O(n+m\log^2 n)$ noncertifying algorithm for recognizing them. We show how to modify it to obtain a certifying algorithm.

If M is a matrix, let $M_{i,j}$ denote the element in row i, column j. The rows of a matrix M are in *lexical order* if they are sorted in dictionary order. That is, whenever $\{i, i+1\}$ are a consecutive pair of rows and j is the first column where the rows differ, then $M_{i+1,j} > M_{i,j}$. The columns are in lexical order if, as rows, they are in lexical order in the transpose of the matrix.

A bipartite graph G with bipartition U, W can be represented with a Boolean *bipartite adjacency matrix* A, which has one row i for each element of $u_i \in U$, one column for each element $w_i \in W$, and for each pair $\{u_i, w_j\}$, $A_{i,j} = 1$ if $u_i w_j$ is an edge of G, and $A_{i,j} = 0$ otherwise.

Given a graph G, Lubiw's algorithm tests whether G is bipartite, and, if so, finds a bipartition and a *doubly lexical ordering* of the resulting bipartite adjacency matrix. A doubly lexical ordering of a matrix is a permutation of the rows and columns such that the resulting matrix passes the following two tests:

1. When the order of the columns is reversed, the rows are in lexical order.
2. When the order of the rows is reversed, the columns are in lexical order.

Every matrix has a doubly lexical ordering, so this is not a test of whether G is chordal bipartite. Lubiw's algorithm then searches this doubly lexical matrix A for a configuration called a *Gamma* (Γ). A Γ is a pair (h, i) of rows and a pair (j, k) of columns such that $h < i$, $j < k$, $A_{h,j} = A_{h,k} = A_{i,j} = 1$, and $A_{i,k} = 0$.

The critical theorem is that a bipartite graph is chordal bipartite iff a doubly lexical ordering of its bipartite adjacency matrix has no Γ. Lubiw's algorithm for finding the doubly lexical ordering and testing for Γ's uses a sparse representation of A and takes $O(n+m\log^2 n)$, which gives this bound for recognition of chordal bipartite graphs. The bound for finding the doubly lexical ordering has been improved to $O(n+m\log n)$ by Paige and Tarjan [22], and to $O(n^2)$ by Spinrad [26], and these give bounds of $O(n+m\log n)$ and $O(n^2)$, respectively, for recognizing chordal bipartite graphs.

Though its proof of correctness is not obvious, Lubiw's algorithm for searching for a Γ with respect to given orderings of the bipartition classes is trivial to program and runs in $O(n+m)$ time. Therefore, a doubly lexical ordering is a strong certificate that a graph is chordal bipartite. The certificate is represented by a bipartition of the vertices and two orderings of the bipartition classes. The authentication algorithm verifies that each of the two bipartition classes is an independent set and that the graph has no Γ with respect to the given orderings of the bipartition classes. All of this takes $O(n+m)$ time. The doubly lexical ordering is therefore a strong certificate, but not a sublinear one.

When a graph is not chordal bipartite, the same certificate can be used to show this, so it is a strong certificate for this case also. However, with some additional effort, we can obtain a sublinear certificate. Note that this precludes verifying the correctness of a doubly lexical ordering, which requires $\Theta(n+m)$ time.

Lubiw [16] proves that when a Γ occurs in a doubly lexical ordering, it is part of a chordless cycle that has at least six vertices. To find such a cycle, let $\{(a,b),(b,c),(c,d)\}$ be the Γ, and remove $N(b)\cup\{b\}\cup N(c)\cup\{c\}-\{a,d\}$ from the graph. Since the Γ is part of a chordless cycle, there exists a path from a to d in this induced subgraph. Using breadth-first search (BFS) starting at a, we may find a shortest path P from a to d in this induced subgraph. Since $\{(a,b),(b,c),(c,d)\}$ is a P_4 and all members of P other than a and d are nonneighbors of b and c, the union of P and the $\{(a,b),(b,c),(c,d)\}$ is a chordless cycle of length at least six. This chordless cycle can be returned as a certificate that the input graph is not chordal bipartite.

On first inspection, this does not seem like a sublinear certificate. To verify that the returned cycle C is indeed chordless, a skeptical user must spend $O(n+m)$ time in the worst case to verify that the cycle has no chords. The key to an $O(n)$ authentication algorithm is the observation that its purpose is to verify that the graph has a chordless cycle and to not verify that C is an example of one.

An $O(n)$ authentication algorithm first verifies that C is a cycle in G of size greater than four. It then selects any four consecutive vertices (u,x,y,w) on the cycle and verifies that x and y have no neighbors on C other than the ones that are supposed to have. That is, it verifies that x has no neighbors on C other than u and y, and y has no neighbors on C other than x and w, and that u and w are nonadjacent. If these tests fail, the certificate is ruled faulty. Otherwise, even if C still has undetected chords, the existence of a path from u to w that avoids the neighborhoods of x and y proves the existence of a shortest such path, P. The union of (u,x,y,w) and such a shortest path is a chordless cycle of length greater than four. Therefore, the user may be certain that the graph has a chordless cycle based on this authentication algorithm, even though the algorithm does not fully verify that C is itself chordless.

5. Interval graphs. An undirected graph is *chordal* if it has no chordless cycle of length at least four. Three independent vertices x,y,z of a graph G are an *asteroidal triple* (AT) of G if, between each pair of these vertices, there is a path that contains no neighbors of the third. A graph is said to be *AT-free* if it has no AT. For more information on these and other graph classes we refer the reader to [3, 8]. We will rely on the following well-known theorem.

THEOREM 5.1 (see [15]). *A graph is an interval graph iff it is chordal and AT-free.*

A graph is chordal iff it has a *perfect elimination ordering*, which is an ordering $(v_1,v_2,\ldots,v_n)$ of the vertices such that for each v_i, $N(v_i)\cap\{v_{i+1},v_{i+2},\ldots,v_n\}$ is a clique [6]. A perfect elimination ordering of a chordal graph can be found in

linear time by the lexicographical BFS (LexBFS) algorithm [25, 8]. A modification of this algorithm points out a chordless cycle of length at least four as a certificate of nonmembership [27]. Hence, this is a linear-time certifying recognition algorithm for chordal graphs.

5.1. The certificates. When the graph is an interval graph, we produce an interval model, just as the prior algorithms do. For the authentication step, it is easy to check whether this model corresponds to the input graph in time that is linear in the size of the input graph. The basic trick is to work left-to-right through the model, generating edges implied by the model and rejecting the certificate immediately if the number of edges exceeds the number of edges in the graph. Otherwise, when finished, verify that the generated edges are the same as those in the graph by labeling the vertices with identifiers from 1 to n, and then radix sorting the set of all edges of G according to the identifiers of their endpoints. Since authentication takes linear time, the interval model is a weak certificate.

When the graph is not an interval graph, Theorem 5.1 provides the basis of our certificate: we produce either a chordless cycle or an AT. Despite initial appearances, these can be turned into sublinear certificates. For the AT, we accomplish this by returning not only the triple, but for each pair in the triple, the sequence of edges of a simple path between them that avoids the neighborhood of the third. The sequence of edges may be given by pointers to the corresponding edge structures in the user-supplied data. Given the triple, it is easy to find these three paths in linear time. The authentication algorithm must verify that each proposed path is, in fact, a path, that its edges occur in G, and that no neighbors of the third vertex occur on it. If each path is given by pointers to edges in the input structure in the order in which they occur on the path, this is accomplished in $O(n)$ time by marking the neighbors of each vertex in the triple. This is a sublinear certificate because $O(n)$ is a better bound than $O(n + m)$.

As in the case of chordal bipartite graphs, a chordless cycle can be used to verify in $O(n)$ time that such a cycle exists, without fully verifying that the given cycle is an example of one. The authentication algorithm is the same as in the chordal bipartite case, except that when (u, x, y, w) are consecutive vertices, the certificate is not rejected if u and w are adjacent.

5.2. Generating the certificates. For our certifying algorithm, we use the linear-time algorithm of Korte and Möhring [13] as a subroutine. Though this is not a certifying algorithm, it produces a certificate in the form of an interval representation in the case where the graph is an interval graph.

Suppose the input graph is not an interval graph. Using the algorithm of [27], we return a chordless cycle if the graph is not chordal.

It remains to show how to produce a certificate in the case where the graph is chordal, but not an interval graph. Henceforth, we will assume that this case applies.

The algorithm of Korte and Möhring produces a perfect elimination ordering $(v_1, v_2, \ldots, v_n)$, and incrementally decides whether the subgraph induced by the vertices $\{v_n, \ldots, v_i\}$ is an interval graph. Since we now assume that the graph is not an interval graph, it fails when considering a particular vertex v_{i-1}. The subgraph induced by the vertices $\{v_n, \ldots, v_i\}$ is an interval graph and the subgraph induced by $\{v_n, \ldots, v_i, v_{i-1}\}$ is not.

In the remainder of this section, we use G to denote the subgraph induced by $\{v_n, \ldots, v_i\}$ and we let $x = v_{i-1}$. The graph $G + x$ is a chordal graph, but not an interval graph, and hence it must contain an AT by Theorem 5.1. The neighbors of x

form a clique in G since $(v_1, v_2, \ldots, v_n)$ is a perfect elimination ordering of the input graph.

The Korte–Möhring algorithm provides an interval model of G. We may assume without loss of generality that all endpoints in the interval model of G are pairwise distinct, since, when they are not, they can be perturbed to make this true without altering the represented graph. Let us then number the endpoints in left-to-right order, and for each vertex, let $l(v)$ and $r(v)$ denote the numbers of the left and right endpoints of the interval corresponding to v. This gives a "normalized" interval model where $I(v) = [l(v), r(v)]$ is the interval that corresponds to v, all endpoints are distinct, and $l()$ and $r()$ are integer-valued functions from $V(G)$ to $\{1, 2, \ldots, 2n\}$.

LEMMA 5.2. *Let G be an interval graph such that $G' = G + x$ is not an interval graph and $N(x)$ is a clique. Then x is a member of every AT of G'.*

Proof. Suppose $\{a, b, c\}$ is an AT of G' and $x \notin \{a, b, c\}$. There is a path P from a to b in G' that avoids the neighborhood of c. If P contains x, then, since the neighborhood of x is a clique, x's predecessor and successor on P must be adjacent, and x can be spliced out of P to yield a path in G. Thus, there is a path in G from a to b that avoids the neighborhood of c. By symmetry among the members of $\{a, b, c\}$, $\{a, b, c\}$ is an AT of G, contradicting the assumption that G is an interval graph. □

DEFINITION 5.3. *Let us say that interval $[x_1, x_2]$* precedes *interval $[y_1, y_2]$ iff $x_1 < y_1$ and $x_2 < y_2$. Let P be a path in G. Let the* rightward extent *$R(P)$ of P denote* $\texttt{max}\ \{r(u) | u$ *is a vertex on $P\}$, and let the* leftward extent *$L(P)$ of P denote* $\texttt{min}\ \{l(w) | w$ *is a vertex in $P\}$. Let $D(x) = \bigcap\{I(v) | v \in N(x)\}$ be the intersection of the intervals representing the neighbors of x. Then $D(x) \neq \emptyset$ since the neighbors of x form a clique.*

Since an AT is an independent set, in any AT $\{x, y, z\}$ of G', one of $I(y)$ and $I(z)$ precedes the other, and if $I(y)$ precedes $I(z)$, then $r(y) < l(z)$.

LEMMA 5.4. *Let $\{x, y, z\}$ be an AT in G', where $I(y)$ precedes $I(z)$. Then*

$$r(y) < l(D(x)) < r(D(x)) < l(z).$$

Proof. Assume otherwise, say, $l(D(x)) < r(y)$, and consider any path P from z to $v \in N(x)$ avoiding $N(y)$. Then $v \notin N(y)$. Together with $l(v) \leq l(D(x))$, this implies $r(v) < l(y)$. Since $r(y) < l(z)$, P must contain a neighbor of y, a contradiction. □

DEFINITION 5.5. *If $r(y) < l(D(x))$, then let $R(y) =$* $\texttt{min}\ \{R(P) | P$ *is a path from y to a neighbor of $x\}$. That is, $R(y)$ is the minimum rightward extent of any path from y to a neighbor of x. Similarly, if $r(D(x)) < l(z)$, then let $L(z) =$* $\texttt{max}\ \{L(P) | P$ *is a path from z to a neighbor of $x\}$. That is, $L(z)$ is the maximum leftward extent of any path from z to a neighbor of x.*

LEMMA 5.6. *If $r(y) < l(D(x)) < r(D(x)) < l(z)$, then $\{x, y, z\}$ is an AT in G' iff y and z are in the same component of $G - N(x)$ and $[r(y), R(y)]$ precedes $[L(z), l(z)]$.*

Proof. If $\{x, y, z\}$ is an AT in G', then y and z are in the same component of $G - N(x)$, since there is a path of G' that avoids the neighborhood of x. There is a path from y to x in G' that avoids the neighborhood of z, and hence a path to a neighbor of x in G that contains no neighbor of z. Since $r(y) < l(z)$, the intervals of all vertices on this path have their right endpoint to the left of $l(z)$. Therefore, $R(y) < l(z)$. By mirror symmetry, $r(y) < L(z)$.

If $\{x, y, z\}$ is not an AT in G', then since $\{x, y, z\}$ is an independent set, every path of G' between some pair of the vertices contains a neighbor of the third. If every path between y and z contains a neighbor of x, then y and z are in different components of $G - N(x)$. If all paths from y to a neighbor of x contain a neighbor of

z, then the rightward extent of all such paths is greater than $l(z)$, and $R(y) > l(z)$. By mirror symmetry, $r(y) < L(z)$. □

Lemma 5.6 gives the strategy of our approach for finding an AT in G'. Removing the intervals corresponding to $N(x)$ from the interval model of G gives an interval model of $G - N(x)$. We look for a component whose intervals span $[l(D(x)), r(D(x))]$. For each y in the component such that $r(y) < l(D(x))$ we compute $R(y)$, and for each z in the component such that $r(D(x)) < l(z)$ we compute $L(z)$. We then look for a pair $[r(y), R(y)]$, $[L(z), l(z)]$, such that $[r(y), R(y)]$ precedes $[L(z), l(z)]$, using Lemma 5.7, which we give first.

LEMMA 5.7. *Given two sets $\mathcal{X}$ and $\mathcal{Y}$ of intervals, where the right endpoints of $\mathcal{X}$ are given in ascending order and the left endpoints of $\mathcal{Y}$ are given in descending order, it takes $O(|\mathcal{X}| + |\mathcal{Y}|)$ time to either determine that no interval in $\mathcal{X}$ precedes any interval in $\mathcal{Y}$, or else return $a \in \mathcal{X}$ and $b \in \mathcal{Y}$ such that a precedes b.*

Proof. As a base case, if $\mathcal{X}$ or $\mathcal{Y}$ is empty, there is no such pair. Otherwise, select $u \in \mathcal{X}$ that minimizes $r(u)$, and select $w \in \mathcal{Y}$ that maximizes $l(w)$. If $l(u) < l(w)$ and $r(u) < r(w)$, then return (u, w) as (a, b). Otherwise, if $l(u) \geq l(w)$, then u is not a candidate for a since its left endpoint does not lie to the left of any left endpoint in $\mathcal{Y}$. Let $\mathcal{X} := \mathcal{X} - \{u\}$. By mirror symmetry, if $r(u) \geq r(w)$, then w is not a candidate for b, so let $\mathcal{Y} := \mathcal{Y} - \{w\}$. By induction on the size of $|\mathcal{X}| + |\mathcal{Y}|$, a recursive call on the new $\mathcal{X}$ and $\mathcal{Y}$ solves the original problem. Because of the way the data are sorted, it takes $O(1)$ time to select u and w and to prepare the recursive call, in which $|\mathcal{X}| + |\mathcal{Y}|$ has been reduced by at least 1. □

To use Lemma 5.7, we let $\mathcal{X} = \{[r(y), R(y)] | r(y) < l(D(x))\}$ and let $\mathcal{Y} = \{[L(z), l(z)] | r(D(x)) < l(z)\}$. Since $r()$ and $l()$ are integer functions from 1 to $2n$, sorting the endpoints as required by the lemma takes $O(n)$ time.

Therefore, by Lemmas 5.6 and 5.7, the problem of finding an AT reduces in linear time to computing $R(y)$ at each y such that $r(y) < l(D(x))$ and $L(z)$ at each z such that $r(D(x)) < l(z)$. We give the procedure for $R()$, and by mirror symmetry, this gives the procedure for $L()$.

DEFINITION 5.8. *A path P in G is* increasing *if, whenever u is earlier than v on P, $r(u) < r(v)$. Let D_r be the orientation of $G - N(x)$, where (u, v) is an arc in D_r iff uv is an edge in $G - N(x)$ and $r(u) < r(v)$.*

Our strategy for computing $R()$ is to find a way to restrict our attention to increasing paths from y to $N(x)$, which allows us to work in D_r rather than in G. Since D_r is a dag, this simplifies the problem.

LEMMA 5.9. *If there is a path P from u to v with rightmost extent $R(P) = r(v)$, then there is an increasing path P' from u to v such that $R(P') = R(P)$.*

Proof. If $u = v$, P is vacuously increasing. Suppose P has length greater than 0 and the lemma is true for shorter paths. Let w be the first vertex on P with $r(w) > r(u)$. $R(P) = r(v)$ implies $r(u) < r(v)$, so w exists. Then $I(u)$ and $I(w)$ intersect, and hence u and w are neighbors. By induction, there is an increasing path P'' from w to v and $P' = uP''$ satisfies the lemma. □

LEMMA 5.10. *Let $r(y) < l(D(x))$ and let P be a path in G from y to a neighbor v of x. Then there is a path $P'v$ from y to v such that $R(P'v) \leq R(P)$ and P' is increasing.*

Proof. If $R(P) = r(v)$, then the claim follows from Lemma 5.9. Otherwise, let w be the first vertex on P such that $I(w)$ intersects $D(x)$, and let P'' be the portion of P from y to w. Since $R(P'') = r(w)$, there is an increasing path P' from y to w, and since $I(w)$ intersects $D(x)$, w and v are adjacent. □

LEMMA 5.11. *It takes linear time to compute $R(y)$ for every $y \in V(G)$ such that $r(y) < l(D(x))$.*

Proof. By Lemma 5.10, we need only consider *well-behaved* paths in G that consist of a directed path in D_r, followed by a single edge of G to a neighbor of x. For any such well-behaved path P, $R(P)$ is the maximum of the last two values of $r()$ on the path. For each $y \in V(G) - N(x)$, let $R_i(y) = \texttt{min}\{R(P)|P$ is a well-behaved path of length at most $i\}$, or ∞ if there is no such path. The value of $R_1(y)$ is trivial to compute at all nodes in $V - N(x)$ in linear time. Let $(u_p, u_{p-1}, \ldots, u_1)$ be a topological sort of D_r. Then $R_i(u_i) = \texttt{min}\ \{R_1(u_i)\} \cup \{R_j(u_j)|(u_i, u_j)$ is an arc of $D_r\}$. This may be computed by induction on i in linear time. For any $y = u_k$ such that $r(y) < l(D(x))$, $R(y) = R_k(u_k)$, since y has at most $k - 1$ successors in D_r. □

5.3. Comparison with the original Korte–Möhring algorithm. One of the anonymous referees has asked for a clarification of why the Korte–Möhring algorithm is not already a certifying algorithm, producing a sublinear certificate of rejection, just as ours does.

Answering this question requires more details about how the algorithm works. The Korte–Möhring algorithm paper [13] defines a *modified PQ tree* (*MPQ tree*) for an interval graph G, which gives a way of representing implicitly all possible interval representations of G. Vertices of G are associated with nodes of the MPQ tree; details about this representation are given in the paper.

As described above, the Korte–Möhring algorithm works by induction on a perfect elimination order $(v_1, v_2, \ldots, v_n)$. Let G_i denote the subgraph of G induced by $\{v_n, v_{n-1}, \ldots, v_i\}$, and let T_i be its MPQ tree. At each step $i - 1$ it produces the MPQ tree T_{i-1} of G_{i-1} from the MPQ tree T_i of G_i, unless G_{i-1} fails to be an interval graph, in which case it rejects G_{i-1}, and hence G is not an interval graph.

At each step, recognizing whether G_{i-1} is an interval graph is trivial, since this is the case iff the neighbors of v_{i-1} in G_i have a certain relationship to T_i that is quite easy to check in $O(n)$ time. If G is not an interval graph, the test will fail at some step $i - 1$.

Since it is possible to verify that G is not an interval graph in $O(n)$ time using T_i and v_{i-1}, the referee has suggested that (T_i, V_{i-1}) be returned as a sublinear certificate of rejection.

Though compelling, this idea has the subtle flaw that it violates the principle that it must not be possible for an authentication algorithm to be induced to make a false declaration by an erroneous or counterfeit certificate. It must either declare a correct answer to the original problem, or it must correctly declare that the claimed certificate contains an error and fails to show what it claims to show. Showing that (T_i, v_{i-1}) fails the simple $O(n)$ test demonstrates only that *either* G_{i-1} is not an interval graph, *or* that T_i is not the correct MPQ tree for G_i, *or* that T_i is the correct MPQ tree for G_i but that $(v_1, v_2, \ldots, v_n)$ is not a perfect elimination order. In fact, the authors make clear that an additional requirement for the algorithm to work is for $(v_1, v_2, \ldots, v_n)$ to be a special case of a perfect elimination order produced by the LexBFS algorithm.

Therefore, in addition to verifying that (T_i, v_{i-1}) fails to have the required simple relationship to T_i, it is also necessary to verify that T_i is the correct MPQ tree for G_i and that $(v_1, v_2, \ldots, v_n)$ is a LexBFS order.

A necessary step in checking that T_i is the correct MPQ tree for G_i is to check that an interval model it implies correctly reflects every edge of G_i. Otherwise, it leaves open the possibility that T_i is the MPQ tree for some interval graph $G_i' \neq G_i$.

The authentication algorithm will falsely declare that G_{i-1} is not an interval graph if $G'_i + v_{i-1}$ is not an interval graph, but $G_i + v_{i-1}$ is. There is little hope of finding a sublinear algorithm for checking whether a given interval model faithfully represents a given graph G, though this is easily accomplished in linear time. The claim that it is a sublinear certificate cannot be made, given what is currently known, and there are reasons to believe that this will not change in the future.

Whether (T_i, v_{i-1}) is useful as a weak certificate depends on the question of whether it clearly simplifies conceptually the task of verifying that G is not an interval graph. In constructing T_i, checking whether T_j and v_{j-1} satisfy the required test for $j > i$ is quite simple, but updating T_j to produce T_{j-1} when it passes this test is considerably more complicated. Though it takes linear time, the problem of checking that an interval model $\mathcal{I}_i$ implied by the MPQ tree T_i faithfully represents G_i is a fairly simple task. However, once this has been accomplished, one must still verify that T_i is the correct MPQ tree for $\mathcal{I}_i$. This problem is one of the main subjects of a forthcoming paper [17]; it is shown there that it can be solved in $O(n)$ time. The algorithm is quite involved, though it is possible that a simpler algorithm could accomplish the task in $O(n + m)$ time. As for verifying the LexBFS order, there is an algorithm for checking whether an order is a perfect elimination order [8], but we do not know of an algorithm for verifying a LexBFS order other than rerunning the LexBFS algorithm that produced it. An interesting question is whether the interval representation could help.

In summary, the usefulness of (T_i, v_{i-1}) in simplifying the problem significantly without leaving open the possibility of giving erroneous output has not been demonstrated. If this is accomplished, it appears unlikely that the resulting authentication will compete in simplicity with the one for checking a given AT or given chordless cycle, or that it will be sublinear.

6. Comparability graphs. Let us consider an undirected graph G to be a special case of a digraph, namely, the symmetric digraph where if (x, y) is a directed arc, then so is (y, x). The undirected edge xy is just the pair $\{(x, y), (y, x)\}$. Finding a transitive orientation of an undirected graph G amounts to deleting one arc from each symmetric pair so that the remaining arcs form a transitive digraph.

Suppose (a, b) and (b, c) are arcs of G and (a, c) is not. Any orientation of G where both (a, b) and (b, c) appear must fail to be transitive, due to the forced absence of (a, c) in the orientation. Then (b, c) is *incompatible* with (a, b), as is (b, a), since a transitive orientation must be antisymmetric. We may represent the incompatibility relation with an *incompatibility graph* whose vertices are the arcs of G and whose (undirected) edges are the incompatible pairs of arcs of G. (See Figure 4.)

We show in section 6.3 that the following is an immediate corollary of a result from [7, 8].

THEOREM 6.1. *An undirected graph G is a comparability graph iff its incompatibility graph is bipartite.*

The following is a consequence of Theorem 6.18, which we give in section 6.3, and gives a sublinear certificate that a graph is not a comparability graph, since it can be checked in $O(n)$ time.

THEOREM 6.2. *When G fails to be a comparability graph, its incompatibility graph has an odd cycle of length $O(n)$.*

A linear-time algorithm for finding a transitive orientation of a comparability graph is given in [18]. This transitive orientation algorithm represents the orientation it assigns to the edges implicitly by giving a linear extension (topological sort) of the

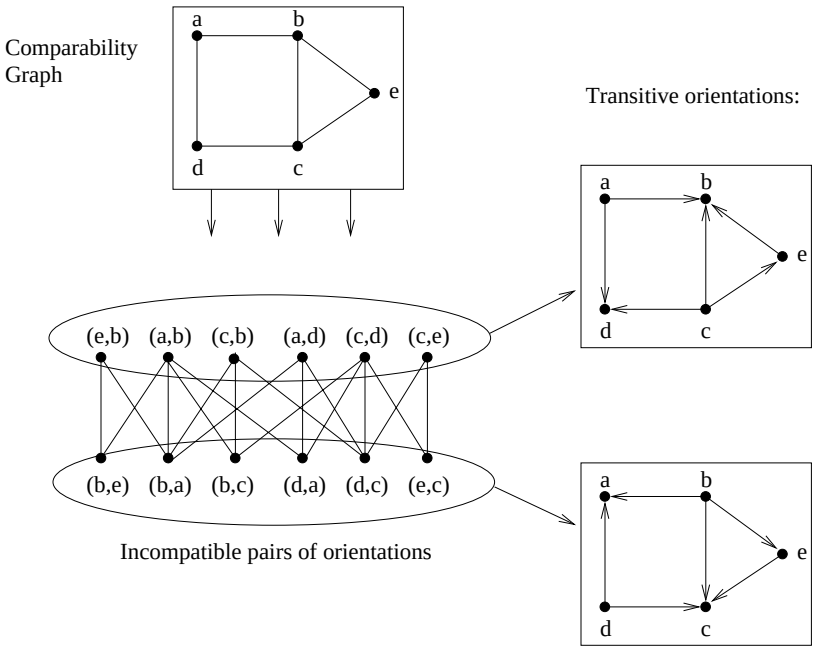

FIG. 4. *The incompatibility graph of a graph. There is one vertex for each directed edge, and two are adjacent if one is the transpose of the other, or if they are of the form* $\{(a,b),(b,c)\}$ *and* (a,c) *is not an edge. Two adjacent edges cannot appear in a transitive orientation. Therefore, a transitive orientation must be an independent set in the incompatibility graph. Reversing the direction of the edges in a transitive orientation yields a new transitive orientation. From this observation, it is easy to see that the incompatibility graph of a comparability graph is bipartite.*

orientation that it produces. This allows it to be applied to $\overline{G}$ in time linear in the size of G.

When the transitive orientation algorithm is asked to provide a transitive orientation of a graph G that is not a comparability graph, it produces an acyclic orientation of the graph, which it represents with a linear extension. This orientation must contain an incompatible pair, namely, a pair $\{(a,b),(b,c)\}$ of directed edges in series such that ac is not an edge of G, and hence (a,c) is not a directed edge in the orientation. No general linear-time algorithm for finding an incompatible pair in a dag is known. Because of this, no linear-time algorithm is known for recognizing comparability graphs, even though a linear-time algorithm for transitively orienting them is available.

We prove the following lemma in section 6.3.

LEMMA 6.3. *Given an incompatible pair in the orientation of a graph* G *produced by the transitive orientation algorithm of* [18]*, it takes* $O(n+m)$ *time to find an odd cycle of length* $O(n)$ *in* G*'s incompatibility graph, and given an incompatible pair in an orientation of* $\overline{G}$ *by the algorithm, it takes* $O(n+m)$ *time to find an odd cycle of length at* $O(n)$ *in* $\overline{G}$*'s incompatibility graph.*

6.1. Minimum proper coloring and maximum clique. The algorithm of [18] for finding a minimum proper coloring and maximum clique in a comparability graph proceeds as follows. Given an arbitrary input graph G, it finds an acyclic orientation that will be transitive if the input is a comparability graph. It then labels each vertex

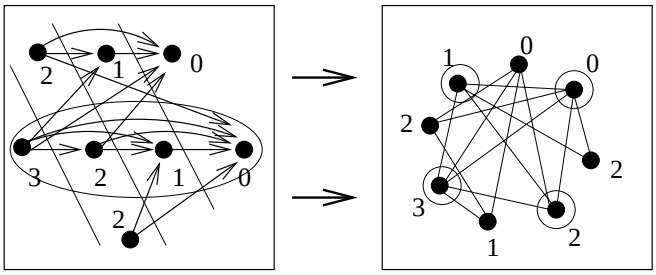

FIG. 5. *Finding a maximum clique and minimum proper coloring in a comparability graph. The color of each vertex is the length of the longest directed path originating in a transitive orientation. A longest path is a clique. Since the size of a clique is a lower bound on the number of colors in a proper coloring, the coloring is a certificate that the clique is of maximum size and the clique is a certificate that the proper coloring is a minimum one.*

with the length of the longest directed path originating at the vertex in the resulting directed acyclic graph. This labeling is easily accomplished in $O(n + m)$ time by labeling each vertex as it finishes during a depth-first search of the graph. (See Figure 5.) Each neighbor of the vertex is already labeled at that point, so the vertex can be labeled with one plus the maximum of the labels of its neighbors. This is a proper coloring, since for any edge xy of G, the label of one of x and y will exceed the other's by at least 1.

If the longest path is a clique, then the coloring is a minimum one and the path is a maximum clique. This is because the size of a clique is a lower bound on the number of colors needed, and since the clique has the same number of vertices as the number of colors used, the coloring serves as a certificate that the clique is a maximum one and the clique serves as a certificate that the coloring is a minimum one. These are weak certificates, since it takes $O(n+m)$ time for a user to verify that the coloring is a proper one and that the path is a clique.

If G is not a comparability graph, the algorithm still assigns an acyclic orientation, which allows the vertices to be colored as before, and finds a maximum-length directed path P. If P is a clique in G, then it is a maximum clique and the coloring is a minimum proper coloring, so these may be returned as each other's certificate just as in the case where G is a comparability graph. If P is not a clique in G, then there exist two consecutive arcs (a, b) and (b, c) on P such that (a, c) is not an edge, and these are easy to find in linear time. Since (a, b) and (b, c) are an incompatible pair, an odd cycle in the incompatibility graph may then be found in $O(n+m)$ time, by Lemma 6.3. As pointed out above, this is a sublinear certificate that G is not a comparability graph.

6.2. Permutation graphs. In [18], it is shown that the transitive orientation algorithm given there gives rise to a linear-time bound for recognizing permutation graphs. The algorithm is based on the following characterization of permutation graphs.

THEOREM 6.4 (see [23, 8]). *An undirected graph G is a permutation graph iff G and its complement $\overline{G}$ are both comparability graphs.*

When G is a permutation graph, the algorithm finds a topological sort of a transitive orientation D of G and a topological sort of a transitive orientation D' of $\overline{G}$. $D \cup D'$ is a *tournament* (an orientation of a complete graph) and acyclic. It then finds the unique topological sort of $D \cup D'$ to yield a linear ordering of V, and the unique

topological sort of $D^T \cup D'$ to give a second linear arrangement of V. By results from [23, 8], these two linear arrangements are a permutation model of G.

When G is not a permutation graph, the procedure in [18] produces a faulty permutation model of G. Success or failure of an authentication algorithm on the permutation model it produces provides the basis for deciding whether the graph is a permutation graph in that algorithm. The procedure is not a certifying algorithm, since the permutation model could also have been faulty due to a bug in the implementation.

A linear-time authentication algorithm for a proposed permutation model is given in [18]. The permutation model is therefore a weak certificate.

The algorithm for recognizing permutation graphs given in [18] uses the transitive orientation algorithm to find linear extensions of orientations D and D' of G and of $\overline{G}$. Since it provides a certificate if G is a permutation graph, we will assume in the remainder of the paper that G is not. In this case, at least one of D and D' has an incompatible pair.

We now describe how to find an incompatible pair in D or D' in time linear in the size of G, given G and linear extensions π and τ of D and of D'. This constitutes proof that the implementation of the transitive orientation algorithm failed to produce an orientation of G or of $\overline{G}$ that is transitive. However, it is not a certificate that G is not a permutation graph, since the failure could be due to a bug in the implementation of the algorithm.

LEMMA 6.5. *Let G be a graph, and let D and D' be acyclic orientations of G and $\overline{G}$. Then $D \cup D'$ and $D^T \cup D'$ are both acyclic iff D and D' are each transitive.*

Proof. Since $D \cup D'$ is a tournament, then if it has a cycle, it has a three-cycle. Suppose there is a directed three-cycle $(x, y), (y, z), (z, x)$ in $D \cup D'$. Since D and D' are both acyclic, one of these arcs belongs to D and another belongs to D'. Suppose without loss of generality that $(x, y), (y, z)$ belong to D. Then since $(x, z) \notin D$, D is not transitive. An identical argument applies if $D^T \cup D'$ contains a directed three-cycle.

Next, suppose that one (or both) of D and D' fails to be transitive. Assume without loss of generality that D fails to be transitive. Then there exists an incompatible pair $\{(a, b), (b, c)\}$. Therefore (a, c) is not an arc of D, and since D is acyclic, (c, a) is not an arc of D. Therefore, (a, c) or (c, a) is an arc of D'; if (a, c) is an arc of D', then $\{a, b, c\}$ induces a three-cycle in $D^T \cup D'$, and if (c, a) is an arc of D', then $\{a, b, c\}$ induces a three-cycle in $D \cup D'$. □

LEMMA 6.6. *Let G, D, and D' be as in Lemma 6.5. Given a three-cycle in $D \cup D'$ or $D^T \cup D'$, it takes $O(1)$ time to return an incompatible pair in D or in D'.*

Proof. Suppose the three-cycle occurs in $D \cup D'$. Since each of D and D' is acyclic, two of the arcs of the cycle occur in one of D and D' and give an incompatible pair in it. □

Let $p(x)$ be the number of predecessors of $x \in V$ in $D \cup D'$. This is just $|N^-(x)|$ in D plus $|N^-(x)|$ in D'.

LEMMA 6.7. *If for each $i \in \{0, 1, \ldots, n-1\}$ there exists $x \in V$ such that $p(x) = i$, then $D \cup D'$ is acyclic.*

Proof. (By induction on i.) Suppose $D \cup D'$ has n vertices and satisfies the conditions of the lemma. The claim is immediate if $n = 1$. Suppose $n > 1$ and the claim holds for $n-1$. There is a vertex s in $D \cup D'$ such that $p(s) = n-1$. Since every other vertex is a predecessor of s, no directed cycle of $D \cup D'$ contains s. However, removal of s leaves an induced subgraph of $D \cup D'$ on $n - 1$ vertices that satisfies

the condition of the lemma, so by the induction hypothesis, there can be no directed cycle that excludes s. □

LEMMA 6.8. *Let G, D, and D' be as in Lemma 6.5, and let π and τ be topological sorts of D and D', respectively. Given G, π, and τ, it takes $O(n+m)$ time to find a three-cycle in $D \cup D'$ or else determine that $D \cup D'$ is acyclic.*

Proof. In $O(n)$ time, we may label the elements of V with their position numbers in π and in τ. In $O(n+m)$ time, we can then label every $x \in V$ with the value of $|N^-(x)|$ in D by counting, for each vertex, the neighbors in G with earlier position numbers. To find $N^-(x)$ in D' we cannot do this directly in linear time, since D' is an orientation of $\overline{G}$, which might not have $O(n+m)$ size. Instead, let $i(x)$ be the number of predecessors of x in τ; $i(x)$ is just the position number of x in τ, minus one. Let $q(x)$ denote the number of neighbors of x in G that have earlier position numbers in τ. We can then compute $|N^-(x)|$ in D' as $i(x) - q(x)$. It takes $O(n+m)$ time to compute $q(x)$ for all $x \in V$, and hence $O(n+m)$ time to label each $x \in V$ with $|N^-(x)|$ in D'.

If the condition of Lemma 6.7 holds, then $D \cup D'$ is acyclic. Otherwise, there exist $x, y \in V$ such that $p(x) = p(y)$. Without loss of generality, suppose that $(x, y) \in D \cup D'$. Since y has x as a predecessor, and x and y have the same number of predecessors, then in $D \cup D'$, x must have a predecessor z that y does not have. In $O(n)$ time, we may list the predecessors of x in D and in D', do the same for y, and compare these two lists to find such a z. Then (x, y), (y, z), (z, x) is a three-cycle. □

By symmetry, Lemma 6.8 also applies to $D^T \cup D'$. The linear time bound for finding the incompatible pair now follows by Lemma 6.6.

6.3. Proof of Theorem 6.1, Theorem 6.2, and Lemma 6.3. In this subsection, we give a linear-time algorithm to find an odd cycle of length $O(n)$ in the incompatibility graph of G, given an incompatible pair in the orientation assigned to it by the transitive orientation algorithm. We show how to apply the algorithm to $\overline{G}$ in time linear in the size of G.

Let Γ be the relation on arcs, where $(u, w)\Gamma(x, y)$ if $u = x$ and w and y are nonadjacent or $w = y$ and u and x are nonadjacent. (This accepted term has nothing to do with the Γ's defined in section 4.) When $(u, w)\Gamma(x, y)$, any transitive orientation that contains one of the arcs must also contain the other. Let $G = (V, E)$ be an arbitrary undirected graph, and let A_G be its directed arcs. Let Γ_G be the graph (A_G, Γ) whose vertices are the arcs of G and whose edges are the pairs of elements in Γ.

DEFINITION 6.9. *A* transposed path *is a path in Γ_G between an arc (x, y) of G and its transpose (y, x).*

The following is well known.

THEOREM 6.10 (see [7, 8]). *An undirected graph G is a comparability graph iff it has no transposed path.*

LEMMA 6.11. *If there is a transposed path of length at most k in Γ_G, then there is an odd cycle in G's incompatibility graph of length at most $k+1$.*

Proof. Note that $e_1 \Gamma e_2$ iff $e_1 \neq e_2^T$ and $e_1 e_2^T$ is an edge of the incompatibility graph. A path $(e_0, e_1, \ldots, e_k)$ in Γ_G can be turned into a path in Γ_G by replacing each edge of odd index with its transpose. An odd-length path from e_0 to e_0^T in Γ maps to an odd-length cycle from e_0 to e_0 in the incompatibility graph, and an even-length path from e_0 to e_0^T in Γ maps to an even-length path from e_0 to e_0^T in the incompatibility graph, which, together with the edge (e_0^T, e_0), defines an odd cycle in the incompatibility graph. □

Proof of Theorem 6.1. The proof is immediate from Lemma 6.11.

Re-expressing Theorem 6.10 as Theorem 6.1 makes it immediately obvious to a user that the certificate proves that G has no transitive orientation. Since a skeptical user can check an edge of the incompatibility graph in $O(1)$ time, an odd cycle of size $O(n)$ in the incompatibility graph is a sublinear certificate that G is not a comparability graph if the odd cycle has size $O(n)$. However, Γ_G is useful for explaining the algorithm to generate the certificate. The constructive proof of Lemma 6.11 shows how to convert a transposed path to an odd cycle of the incompatibility graph in time proportional to the length of the transposed path.

A *module* of an undirected graph $G = (V, E)$ is a set X of vertices such that for each vertex $y \in V - X$, either every element of X is a neighbor of y or no member of X is a neighbor of y. V, the empty set, and the singleton subsets $\{\{x\}|x \in V\}$ are *trivial modules*. G is *prime* if it has only trivial modules. A set of vertices in G is a module iff it is a module in $\overline{G}$, and hence G is prime iff its complement is prime.

The problem of verifying that G is a comparability graph reduces in linear time to the problem of verifying that a set of prime induced subgraphs is a set of comparability graphs [7, 18]. A transposed path in an induced subgraph is also a transposed path in G. Therefore, when G or $\overline{G}$ is not a comparability graph, producing a transposed path in G or in $\overline{G}$ reduces in linear time to the same problem in the special case where G is prime.

THEOREM 6.12 (see [7, 8]). *Let G be a prime undirected graph. If G is not a comparability graph, then Γ_G has one connected component. Otherwise, Γ_G has two components, where one component contains the transposes of the arcs in the other component.*

We show how to modify the transitive orientation algorithm of [18] so that it creates a record that allows us to find a path of length $O(n)$ in Γ_G between any two arcs that are included in its orientation of a prime graph G.

The transitive orientation algorithm of [18] begins with a partition $\mathcal{P} = \{\{v\}, V - \{v\}\}$ of the vertices V of a prime graph G, where v is a selected *initial lone vertex*. In a process called *vertex partitioning*, it iteratively refines the partition using the following step, until $\mathcal{P}$ is the partition of V into one-element subsets:

- Select a vertex x as a *pivot*, and a partition class Y that does not contain x. Split Y into two classes, $Y_a = Y \cap N(x)$ of vertices that are *adjacent* to x and $Y_n = Y - N(x)$ of vertices that are *nonadjacent* to x. Let $\mathcal{P} := (\mathcal{P} - \{Y\}) \cup \{Y_a, Y_n\}$.

Figure 6 gives an example. Performing the first pivot on the initial lone vertex v splits $V - \{v\}$ into nonneighbors $\{c, f, g\}$ and neighbors $\{a, e, w\}$. Performing a pivot on w then splits the class $\{c, f, g\}$ into neighbors $\{c, f\}$ and nonneighbor $\{g\}$. Performing a pivot on $\{f\}$ then splits $\{a, e, w\}$ into nonneighbor $\{a\}$ and neighbors $\{e, w\}$, etc. Since G is prime, any partition class of size greater than one fails to be a module, so it is always possible to find a pivot that will split it. The vertex partitioning procedure halts when all partition classes are of size 1.

During the partitioning, a linear arrangement of the partition classes is maintained, so that when a set Y is split into two sets, Y_n and Y_a, these two sets occupy consecutive places at the former position of Y, with Y_a placed farther than Y_n from the partition class that contains the pivot. Initially, the lone vertex $\{v\}$ is placed first. For example, in Figure 6, when v splits $\{a, c, e, f, g, w\}$, it is in a class that *precedes* $\{a, c, e, f, g, w\}$, so the neighbors $\{a, e, w\}$ are placed *after* the nonneighbors, $\{c, f, g\}$. In the next step, when w then splits $\{c, f, g\}$, w is in a class that *follows* $\{c, f, g\}$, so

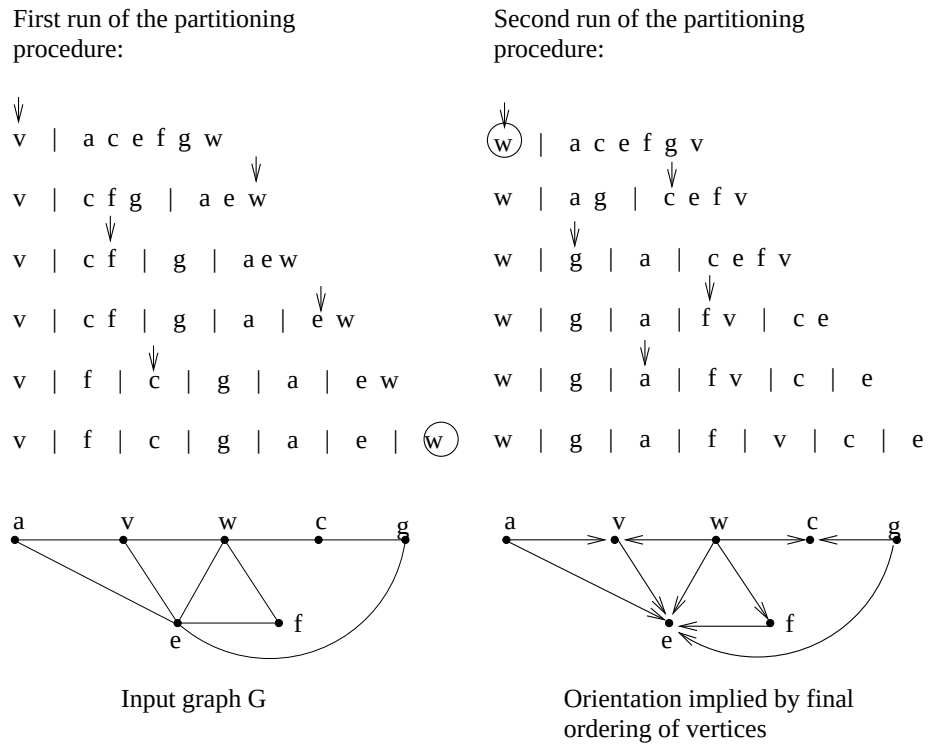

FIG. 6. *The transitive orientation algorithm of* [18] *performs two vertex partition refinements on a prime comparability graph in order to find a linear ordering of the vertices that gives a topological sort of a transitive orientation of the graph. The final ordering given by the second partition,* (w, g, a, f, v, e, c) *in this example, is the topological sort, which gives implicitly the transitive orientation of the graph. Arrows indicate pivot vertices that are used for the next refinement of the partition.*

the neighbors $\{c, f\}$ are placed before the nonneighbor $\{g\}$ in the ordering.

It is shown in [18] that if the initial lone vertex v is a source or sink in a transitive orientation of G, then the final ordering of vertices will be a topological sort of that transitive orientation. Moreover, it is shown that if v is an arbitrary vertex, then the rightmost vertex in the final ordering must be a source or sink in a transitive orientation of G. (The reasons will become apparent below.) Therefore, the procedure is run twice, once starting with arbitrary initial lone vertex v to identify a source/sink w, and once starting with w as the initial lone vertex to find a topological sort of the transitive orientation. Since the topological sort implies the orientations of the edges, this gives the transitive orientation if G is a comparability graph. In the illustration, the ordering (w, g, a, f, v, c, e) produced by the second run of the vertex partitioning gives the transitive orientation depicted in the graph at the bottom.

Let us now examine what happens when G is not a comparability graph. The procedure still produces an ordering of the vertices. Since G has no transitive orientation, this orientation must contain a pair (a, b) and (b, c) of directed edges such that (a, c) is not an edge, namely, an incompatible pair. An incompatible pair does not serve as a certificate that G is not a comparability graph, since it shows only that either G is not a comparability graph or the implementation of the transitive orientation algorithm has a bug.

We therefore seek a mechanism to turn an incompatible pair into a certificate

that G is not a comparability graph.

To accomplish this, we define a *parent* relation on the directed arcs of G that results from the vertex partitioning procedure. Suppose a set Y is split into Y_n and Y_a by a pivot vertex x. Let (y, z) be an arc from Y_n to Y_a. By the definition of Y_n and Y_a, xy is not an edge of G, and xz is, so $(y, z)\Gamma_G(x, z)$ and $(z, y)\Gamma_G(z, x)$. Let (x, z) be the *parent* of (y, z), and let (z, x) be the *parent* of (z, y).

LEMMA 6.13. *If arc a_1 is the parent of arc a_2, then $a_1\Gamma_G a_2$.*

Clearly, every arc has a unique parent except those that are incident to the initial lone vertex, which have no parents. At a given point in the partition refinement, let us say that a directed arc is *protected* if both of its endpoints are currently within a single partition class, and *exposed* if its endpoints are in two different partition classes. The parent relation is acyclic, since the parent is always exposed earlier than the child. All arcs are assigned a parent except those that are incident to the initial lone vertex, so the parent relation arising from one run of the vertex partitioning procedure defines a forest of rooted trees on the arcs of G, and the roots of these trees are the arcs incident to the initial lone vertex.

Let P_1 be the parent relation arising from the first run of the vertex partitioning procedure, which begins with partition $\{\{v\}, V - \{v\}\}$ and discovers a source/sink w. Let P_2 be the parent relation arising from the partition $\{\{w\}, V - \{w\}\}$. The arcs incident to v are the tree roots in P_1, and the arcs incident to w are the tree roots in P_2. Therefore, the only arcs that fail to have parents in both P_1 and P_2 are (v, w) and (w, v). Moreover, since w is in the rightmost class after every pivot during the first run of the partitioning procedure, every time it is in a class that is split by a pivot, it is in the class that contains neighbors of the pivot. Since it is a neighbor of the pivot, the parent of each edge incident to w that gets exposed by the pivot is also incident to w.

Therefore, all arcs incident to w lie in the two trees of P_1 that are rooted at (v, w) and (w, v). Let P_1' be the restriction of P_1 to arcs incident to w; that is, P_1' is two trees rooted at (v, w) and (w, v) that span the arcs incident to w.

It follows that $P_1' \cup P_2$ consists of exactly two trees $T_{(v,w)}$ and $T_{(w,v)}$ that are rooted at (v, w) and (w, v) and that span all directed arcs of G. Whenever a_1 and a_2 are arcs of G where a_1 is a parent of a_2, $a_1\Gamma_G a_2$. By Lemma 6.13, any transitive orientation of G that contains (v, w) must contain every arc in $T_{(v,w)}$ and any transitive orientation of G that contains (w, v) must contain every arc in $T_{(w,v)}$. The arcs spanned by the two trees are the two transitive orientations of G if G has a transitive orientation. Let us therefore call $T_{(v,w)}$ and $T_{(w,v)}$ the *orientation trees*.

Figure 7 depicts one of the two orientation trees produced by the two runs of the vertex partitioning algorithm in Figure 6. Each node is labeled with two vertices and represents the arc from the first vertex label to the second. (The other orientation tree is identical except that the directions of all of the edges are reversed.) The parent relation on arcs determined by the second run (P_2) is shown with solid edges; the parent relation on edges incident to w determined by the first run (P_1') are shown with dashed edges.

Given a parent arc a_1 and child arc a_2, it is easy to check in $O(1)$ time that $a_1\Gamma_G a_2$. Performing this on all parent-child pairs allows a user to confirm in $O(n + m)$ time that *if* G has a transitive orientation, then it must be the orientation consisting of the nodes of one of the two trees. It therefore serves as a certificate either that the precondition that G was a comparability graph was violated or that the orientation is transitive, without identifying which of these two cases occurred. In either case, it exonerates an implementation of the transitive orientation algorithm of providing a

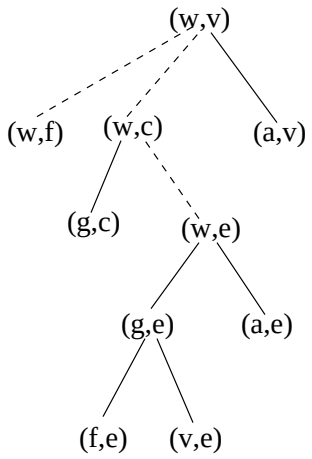

FIG. 7. *The orientation tree defined by the two runs of the vertex partitioning procedure shown in Figure* 6. *Solid edges depict the parent relation implied by the second run of the partitioning procedure and form a forest of trees rooted at arcs incident to* w. *Dashed edges give the parents of edges incident to* w *that are implied by the first run of the partitioning procedure. Together, these edges link this forest together into a tree rooted at* (w, v).

nontransitive orientation of a comparability graph.

In contrast to a simple transitive orientation, the orientation trees contain information that allows us to find a transposed path if G is not a comparability graph. In this case, each of the trees must contain two edges (a, b) and (b, c) that form an incompatible pair, which implies that ac is not an edge of G, and $(b, a)\Gamma_G(b, c)$.

Suppose without loss of generality that (a, b) and (b, c) occur within $T_{(v,w)}$. To document that G is not a comparability graph, we may find the least common ancestor (d, e) of (a, b) and (b, c) in $T_{(v,w)}$. Let $P_{(a,b)}$ be the path from (a, b) to (d, e) and $P_{(b,c)}$ be the path from (b, c) to (d, e) in $T_{(v,w)}$. $P_{(a,b)} \cup P_{(b,c)}$ defines a path in Γ_G from (a, b) to (b, c). Taking these together with $(b, a)\Gamma_G(b, c)$, we get a transposed path from (a, b) to (b, a)—a certificate that G is not a comparability graph.

For instance, suppose edge ae is removed from the graph of Figure 6. The resulting graph is no longer a comparability graph, but it is easy to see that the removal of this particular edge does not affect any of the steps of either of the two runs of the vertex partitioning procedure. Therefore, the transitive orientation algorithm still produces the topological sort (w, g, a, f, v, c, e) of an orientation that contains the incompatible pair $((a, v), (v, c))$. The least common ancestor of (a, v) and (v, e) in the orientation tree of Figure 7 is (w, v). The union of the paths from (a, v) and (v, c) to this least common ancestor forms the path $((v, e), (g, e), (w, e), (w, c), (w, v), (a, v))$ which, together with $((e, v)\Gamma_G(a, v))$, is a transposed path $((v, e), (g, e), (w, e), (w, c), (w, v), (a, v), (e, v))$ from (v, e) to (e, v). A skeptic can check each of the links of this path once they are pointed out and conclude that the graph is not, in fact, a comparability graph.

$T_{(v,w)}$ and $T_{(w,v)}$ have size $O(m)$; we now show how to construct them in $O(m)$ time. For this, we modify the vertex partitioning procedure to produce the data structure pictured on the right side of Figure 8. The construction is illustrated for the run of the vertex partitioning procedure, where w is the initial lone vertex; the data structure, where v is the initial lone vertex, is constructed in the same way. The data structure is a tree whose nodes represent subsets of V. The root of the tree is V, its children are $\{w\}$ and $V - \{w\}$ if w is the initial lone vertex, and the remaining

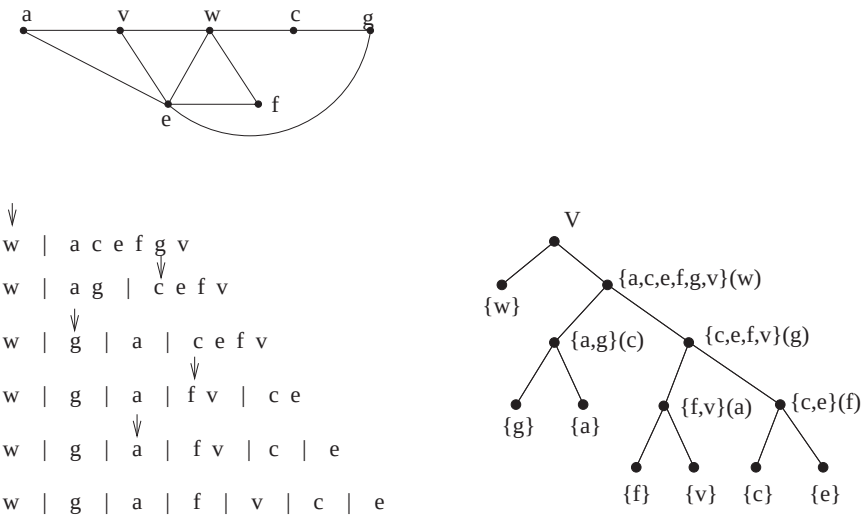

FIG. 8. *The Hasse diagram of a run of the vertex partitioning procedure. If a set Y is split into two sets Y_n and Y_a by a pivot, Y_n and Y_a are its children. In the figure, each set is labeled with its members, as well as the pivot that split it if it is an internal node. For the data structure, it is not necessary to label internal nodes with their members, so the data structure requires $O(n)$ space.*

nodes are the remaining sets that appear at some point during the vertex partitioning procedure. When a partition class Y is split into Y_n and Y_a by a pivot x, Y_n and Y_a are the children of Y. In other words, the tree is the Hasse diagram of the subset relation on the partition classes that appear at some point during the refinement. Let us refer to it as the *Hasse diagram* of the run of the partition refinement algorithm.

Each node is labeled with the identity of the pivot, in parentheses, that caused it to split into its two children during the refinement. For instance, when $\{c, e, f, v\}$ is split into $\{f, v\}$ and $\{c, e\}$ by pivot g, $\{f, v\}$ and $\{c, e\}$ become the children of $\{c, e, f, v\}$, and $\{c, e, f, v\}$ is labeled with the pivot (g) that split it.

To represent this tree with a data structure, we label each leaf with its sole member, and label each internal node only with the pivot that caused it to split, but not with a list of members. The members of the set X represented by an internal node can be found in $O(|X|)$ time by visiting its leaf descendants, which is just as efficient as labeling the node explicitly with the members of X. This allows each internal node to take $O(1)$ space, so the data structure for the Hasse diagram takes $O(n)$ space. The time to construct it does not affect the $O(n + m)$ time to run the vertex partitioning procedure, since it requires creating two children of size $O(1)$ each time a partition class is split by the procedure.

LEMMA 6.14. *Suppose (a, b) is an arc of G. Let Y be the least common ancestor of $\{a\}$ and $\{b\}$ in the Hasse diagram.*

- *If $Y = V$, then (a, b) has no parent.*
- *Otherwise, let c be the pivot that split Y into nonneighbors Y_n and neighbors Y_a of c. Then if $a \in Y_n$ and $b \in Y_a$, then (c, b) is the parent of (a, b), and if $b \in Y_a$ and $a \in Y_n$, then (a, c) is the parent of (a, b).*

Proof. The proof is immediate from the definition of the parent function. □

COROLLARY 6.15. *Given the Hasse diagram and a set A of arcs of G, it takes $O(|A| + n)$ time to find the parents of the members of A.*

Proof. Numbering the leaves from 1 to n in left-to-right order allows one to find, for any two vertices, which is earlier on the leaf order in $O(1)$ time. By the off-line least common ancestors algorithm of Harel and Tarjan [10], given k pairs of nodes in a rooted tree with $O(n)$ nodes, it takes $O(k+n)$ time to find the least common ancestor of each of the k pairs. By the rule for ordering partition classes, for an arc (a,b) with a least common ancestor split by pivot c, $a \in Y_n$ if a is in between b and c in this order, and $b \in Y_n$ otherwise. In the former case, (c,b) is the parent and, in the latter case, (a,c) is the parent. □

COROLLARY 6.16. *It takes $O(n+m)$ time to find the two orientation trees implied by the transitive orientation algorithm.*

Proof. It takes $O(n+m)$ time to get the parents of all arcs not incident to the lone initial vertex w in the parent relation defined by the second run of the partitioning procedure by Corollary 6.15, and it takes $O(n)$ time to get the parents of arcs incident to w, but not v in the first run. The union of these two sets of parent pointers is formed by the two orientation trees. □

Since the arcs of G are nodes of the orientation trees, these trees have $\Theta(m)$ nodes. However, we can now state the following.

COROLLARY 6.17. *The orientation trees have height $O(n)$.*

Proof. Recall that an arc is *exposed* during partitioning at the point when its endpoints are separated into two different partition classes. Each time a partition class splits, the number of partition classes increases by one, and this number is initially equal to one when the initial lone vertex is separated from V and equal to n when the partitioning procedure terminates. Therefore, classes are split at most $n-1$ times. When a split of a class exposes an arc, this means that the parent of the arc was exposed by an earlier split. Therefore, there is no chain longer than $n-1$ in the parent relation implied by one run of the partitioning procedure.

Each path from an arc of G to the arc (w,v) or (v,w) that is the root of the orientation tree follows zero or more parent pointers defined by the second run of the partitioning procedure to arrive at an arc incident to w, followed by zero or more parent pointers defined by the first run of the partitioning procedure, through arcs incident to w, to arrive at (w,v) or (v,w). The height of the tree is therefore at most $2n-2$. □

THEOREM 6.18. *Given an incompatible pair $((a,b),(b,c))$ in the orientation of a graph G produced by the transitive orientation algorithm, it takes $O(n+m)$ time to find an odd cycle of length $O(n)$ in G's incompatibility graph.*

Proof. It takes $O(n+m)$ time to find the orientation tree that contains (a,b) and (b,c) by Corollary 6.16. It takes $O(n)$ time to find the path from (a,b) to (b,c) in this tree by Corollary 6.17. By Lemma 6.13, this path, together with $(b,c)\Gamma_G(b,a)$, is a transposed path from (a,b) to (b,a). The constructive proof of Lemma 6.11 shows how to turn this into an odd cycle of the incompatibility graph of size $O(n)$. □

Proof of Theorem 6.2. The proof is immediate from Theorem 6.18 and Lemma 6.11.

Note that we do not claim an $O(n+m)$ certifying algorithm for recognizing comparability graphs, and no such bound is known for recognizing them, with or without a certificate. The bottleneck for recognition of comparability graphs is finding an incompatible pair in the orientation produced by the algorithm of [18]. It is noteworthy, however, that this gives a certifying algorithm for recognizing comparability graphs that is as fast as any currently known, and that produces a sublinear certificate of rejection.

LEMMA 6.19 (see [18]). *Given a graph G with n vertices and m edges, it takes*

$O(n+m)$ time to find an ordering of the vertices that is a topological sort of a transitive orientation of $\overline{G}$ if $\overline{G}$ is a comparability graph.

The algorithm works by symmetry in the roles of edges and nonedges. It performs the pivots exactly as it does for orienting G, but reverses the roles of the set Y_n of nonneighbors and the set Y_a of neighbors of the pivot. The only effect of this is that it reverses the relative order of Y_n and Y_a when we replace Y with these two sets in the ordering of partition classes. This trick is used in [18] to get linear-time recognition of permutation graphs.

However, one difficulty we now face in producing a certificate that $\overline{G}$ is not a comparability graph in this time bound is that the number of arcs in G, hence the number of nodes of its orientation trees, is $\Theta(n^2 - m)$, which does not conform to our desired $O(n + m)$ time bound. We cannot construct the orientation trees for $\overline{G}$ and stay within our time bound.

Fortunately, given an incompatible pair $((a, b), (b, c))$, the construction of Theorem 6.18 requires us only to find the paths from (a, b) and (b, c) to their least common ancestor in the orientation tree, not the whole orientation tree. These paths have length $O(n)$ by Corollary 6.17.

LEMMA 6.20. *Given an incompatible pair $((a, b), (b, c))$ in the orientation of $\overline{G}$ produced by the algorithm of Lemma* 6.19*, it takes $O(n)$ time to find an odd cycle of size $O(n)$ in the incompatibility graph of $\overline{G}$.*

Proof. Creation of the Hasse diagram is not affected by the modification of the partitioning algorithm for Lemma 6.19. To find the parent of an arc (a, b) of $\overline{G}$ in one run of the partitioning algorithm, mark the ancestors of a in the Hasse diagram. This takes $O(n)$ time. Search upward from b until a marked node is encountered. This is the least common ancestor A_1 of $\{a\}$ and $\{b\}$ in the Hasse diagram. Suppose by induction that the least common ancestor A_k in the Hasse diagram has been found for $\{x\}$ and $\{y\}$, where (x, y) is some ancestor of (a, b) in the parent relation. Let z be the pivot label of this ancestor. Search upward from z until a marked node of the Hasse diagram is encountered. This is the least common ancestor A_{k+1} in the Hasse diagram of the parent of (x, y) in T_1. Since this A_{k+1} is higher in the Hasse diagram than any other least common ancestor found so far, the search upward from z uses a different set of edges of the Hasse diagram from those used by previous upward searches. The cost of this search can be charged to the edges traversed during the search. The cost of finding all ancestors in the Hasse diagram is $O(n)$, and the length of the path is $O(n)$. □

Acknowledgments. The authors would like to thank the two anonymous referees for generous observations, questions, and comments that we have made ample use of in the paper.

REFERENCES

[1] M. BLUM AND S. KANNAN, *Designing programs that check their work,* in Proceedings of the 21st Symposium on the Theory of Computation, ACM, New York, 1989, pp. 86–97.

[2] K. S. BOOTH AND G. S. LUEKER, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[3] A. BRANDSTÄDT, V. B. LE, AND J. P. SPINRAD, *Graph Classes: A Survey*, SIAM Monogr. Discrete Math. Appl. 3, SIAM, Philadelphia, 1999.

[4] D. G. CORNEIL, S. OLARIU, AND L. STEWART, *The ultimate interval graph recognition algorithm?*, in Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 1998, pp. 175–180.

[5] D. G. Corneil, Y. Perl, and L. K. Stewart, *A linear recognition algorithm for cographs*, SIAM J. Comput., 14 (1985), pp. 926–934.
[6] G. A Dirac, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25 (1961), pp. 71–76.
[7] T. Gallai, *Transitiv orientierbare Graphen*, Acta Math. Acad. Sci. Hungar., 18 (1967), pp. 25–66.
[8] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
[9] M. Habib, R. M. McConnell, C. Paul, and L. Viennot, *Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition, and consecutive ones testing*, Theoret. Comput. Sci., 234 (2000), pp. 59–84.
[10] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.
[11] W. L. Hsu, *A simple test for interval graphs*, in Graph-theoretic Concepts in Computer Science, Lecture Notes in Comput. Sci. 657, Springer, Berlin, 1993, pp. 11–16.
[12] W. L. Hsu and R. M. McConnell, *PC trees and circular-ones arrangements*, Theoret. Comput. Sci., 296 (2003), pp. 99–116.
[13] N. Korte and R. H. Möhring, *An incremental linear-time algorithm for recognizing interval graphs*, SIAM J. Comput., 18 (1989), pp. 68–81.
[14] D. C. Kozen, *The Design and Analysis of Algorithms*, Springer, Berlin, 1991.
[15] C. Lekkerkerker and D. Boland, *Representation of finite graphs by a set of intervals on the real line*, Fund. Math., 51 (1962), pp. 45–64.
[16] A. Lubiw, *Doubly lexical orderings of matrices*, SIAM J. Comput., 16 (1987), pp. 854–879.
[17] R. M. McConnell and F. de Montgolfier, *On the Common Factors in a Set of Linear Orders*, Technical report CS-04-102, Colorado State University, Fort Collins, CO, 2004.
[18] R. M. McConnell and J. P. Spinrad, *Modular decomposition and transitive orientation*, Discrete Math., 201 (1999), pp. 189–241.
[19] K. Mehlhorn and S. Näher, *The LEDA Platform for Combinatorial and Geometric Computing*, Cambridge University Press, Cambridge, UK, 1999.
[20] K. Mehlhorn, S. Näher, T. Schilz, M. Seel, R. Seidel, and C. Uhrig, *Checking geometric programs or verification of geometric structures*, Comput. Geom., 12 (1999), pp. 85–103.
[21] K. Mehlhorn, S. Näher, and C. Uhrig, *The LEDA platform for combinatorial and geometric computing*, in Proceedings of the 24th International Colloquium on Automata, Languages, and Programming (ICALP '97), Lecture Notes in Comput. Sci. 1256, Springer-Verlag, Berlin, 1997, pp. 7–16.
[22] R. Paige and R. E. Tarjan, *Three partition refinement algorithms*, SIAM J. Comput., 16 (1987), pp. 973–989.
[23] A. Pnueli, A. Lempel, and S. Even, *Transitive orientation of graphs and identification of permutation graphs*, Canad. J. Math., 23 (1971), pp. 160–175.
[24] F. S. Roberts, *Graph Theory and Its Applications to Problems of Society*, SIAM, Philadelphia, 1978.
[25] D. J. Rose, R. E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.
[26] J. P. Spinrad, *Doubly lexical ordering of dense* 0–1 *matrices*, Inform. Process. Lett., 45 (1993), pp. 229–235.
[27] R. E. Tarjan and M. Yannakakis, *Addendum: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM J. Comput., 14 (1985), pp. 254–255.
[28] H. Wasserman and M. Blum, *Software reliability via run-time result-checking*, J. ACM, 44 (1997), pp. 826–849.