Certifying Algorithms for Recognizing Interval Graphs and Permutation Graphs

Dieter Kratsch * Ross M. McConnell † Kurt Mehlhorn ‡ Jeremy P. Spinrad §

Abstract

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 when they claim that the input graph is not a member of the class. We show that our certificates of non-membership can be authenticated in $O(|V|)$ time.

1 Introduction

A recognition algorithm is an algorithm that decides whether some given input (graph, geometrical object, picture, etc.) has a certain property. 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 non-membership. We show that our certificates of non-membership in can be authenticated in $O(n)$ time, where n is the number of vertices. Such an algorithm *accepts* the input if it has the property or *rejects* it if it does not.

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 non-bipartite input graphs. A more complex example is the linear-time planarity test which is part of the LEDA system [13, Section 8.7]. It computes a planar embedding for planar input graphs and a subdivision of K_5 or $K_{3,3}$ for non-planar input graphs.

Certifying versions of recognition algorithms are highly desirable in practice; see [17] and [13, 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 non-planar otherwise. Though the algorithm may have been proven correct, the implementation may contain bugs. When the algorithm declares a graph non-planar, 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.

In this paper, we examine two such problems, namely, the problems of recognizing whether a graph is an *interval graph* and recognizing whether a graph is a *permutation graph.*

An interval graph is the intersection graph of intervals on a line. 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.

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. If a graph is a permutation graph, we may show this by giving a *permutation model*, which consists of two linear orderings $(v_1, v_2, ..., v_n)$ and $(v_{\pi(1)}, v_{\pi(2)}, ..., v_{\pi(n)})$ 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.

Several linear-time recognition algorithms for interval graphs are known [1, 3, 7, 8, 9, 10, 12]. When a graph is an interval graph, these algorithms produce a certificate in the form of an interval model. However, the existence of certificates of rejection in the form of a forbidden substructure characterization is also well-known. Despite this, these algorithms ignore the issue of producing one when they reject a graph.

*kratsch@lita.sciences.univ-metz.fr, Université de Metz, LITA, 57045 Metz Cedex 01, France

†rmm@cs.colostate.edu, Computer Science Department, Colorado State University, Fort Collins, CO, 80523-1873 U.S.A.

‡mehlhorn@mpi-sb.mpg.de, Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

§spin@vuse.vanderbilt.edu, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235, U.S.A.

The only previous linear-time algorithm for permutation graphs is given in [12]. 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. A graph G is a permutation graph iff G and its complement $\overline{G}$ are *comparability graphs.* Gallai gave a forbidden substructure characterization of comparability graphs [5]. We do not know how to obtain this certificate in linear time for non-comparability graphs. 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. This gives a certifying algorithm for permutation graphs. It 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 $\overline{G}$, even when they both fail to be comparability graphs. (No linear-time recognition algorithm is known for comparability graphs.)

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 $P = (v_1, v_2, ..., v_k)$ is a path, $u \in N(v_1)$, and $w \in N(v_k)$, then uP denotes the path $(u, v_1, v_2, ..., v_k)$, and Pw denotes the path $(v_1, v_2, ..., v_k, w)$.

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 to check whether a graph is bipartite.

A good certificate is one whose authentication algorithm is conceptually simpler than algorithms for the original problem, has a better time bound, or 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, 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 reliable implementation of an authentication algorithm is easier to achieve if it is simple. Though simplicity of the verification algorithm is likely to be the primary motivation in practice, simplicity is admittedly a subjective criterion. Let us therefore dub a certificate *strong* if its verification algorithm has a better time bound than known algorithms for the original problem, and *weak* if it has other advantages that must be judged by subjective criteria. A strong certificate simplifies the problem in a way that can be characterized objectively.

Fortunately, strong certificates appear to be common. However, if the rejection certificate is strong, the acceptance certificate is usually weak, and if the acceptance certificate is strong, the rejection certificate is usually weak. For instance, recognizing bipartite graphs has a strong rejection certificate (an odd cycle) and a weak acceptance certificate (a 2-coloring). Recognizing connected graphs has a strong 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 strong rejection certificate (a directed cycle).

Since the rejection certificates that our algorithms give can be verified in $O(n)$ time, they are strong. The acceptance certificates, which are also given by the previous algorithms, are weak.

4 A certifying algorithm for interval 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. A *chordless cycle* is a simple cycle of length at least *four* that has no chord. An undirected graph is *chordal* if it has no chordless cycle.

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 asteroidal triple. For more information on these and other graph classes we refer the reader to [2, 6]. We will rely on the following well-known theorem.

THEOREM 4.1. (LEKKERKERKER AND BOLAND [11]) *A graph is an interval graph if and only if it chordal*

and AT-free.

A graph is chordal iff it has a *perfect elimination ordering*, which is an ordering $(v_1, v_2, ..., v_n)$ of the vertices such that for each v_i, $N(v_i) \cap \{v_{i+1}, v_{i+2}, ..., v_n\}$ is a clique [4]. A perfect elimination ordering of a chordal graph can be found in linear time by the LexBFS algorithm [15, 6]. A modification of this algorithm points out a chordless cycle of length at least four as a certificate of non-membership [16]. Hence, this is a linear-time certifying recognition algorithm for chordal graphs.

4.1 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, 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 radix sorting both lists to get them into the same order. Since authentication takes linear time, the interval model is a weak certificate.

When the graph is not an interval graph, Theorem 4.1 provides the basis of our certificate: we produce either a chordless cycle or an asteroidal triple. Despite initial appearances, these can be turned into strong certificates. For the asteroidal triple, 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 a path, that its edges occur in G, and that no neighbors of the third vertex occurs 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 strong certificate because $O(n)$ is a better bound than $O(n+m)$.

There appears to be no hope of verifying that a proposed chordless cycle C is, in fact, chordless in sublinear worst-case time. In view of this, it is surprising that C still serves as a strong certificate. The $O(n)$ authentication algorithm first verifies that C is a cycle in G of size at least four. It then selects any four consecutive vertices (u, x, y, w) and verifies that x and y have no neighbors on C other than the ones that they are supposed to. 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. If these tests fail, the certificate is ruled faulty. Otherwise, if the certifying algorithm has a bug, C may still have undetected chords. However, in this case, any minimal cycle consisting of x, y, and other vertices of C must be a chordless cycle of size at least four. The user can be certain that such a cycle exists without actually finding it, hence that G is not an interval graph.

4.2 Generating the certificates For our certifying algorithm, we use the linear-time algorithm of Korte and Moehring [10] 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 [16], 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.

This algorithm of Korte and Moehring produces a perfect elimination ordering $(v_1, v_2, ..., v_n)$, 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 4.1. 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

$\{1, 2, ..., 2n\}$.

LEMMA 4.1. *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 4.1. *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* $\max$ *$\{r(u) | u$ is a vertex on $P\}$, and let the* leftward extent *$L(P)$ of P denote* $\min$ *$\{l(w) | w$ 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. Let $D(x) = [l(D(x)), r(D(x))]$.*

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.

LEMMA 4.2. *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 4.2. *If $r(y) < l(D(x))$, then let $R(y) =$* $\min$ *$\{R(P) | P$ 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 $L(z) =$* $\max$ *$\{L(P) | P$ 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 4.3. *If $r(y) < l(D(x)) < r(D(x)) < l(z)$, then $\{x, y, z\}$ is an AT 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 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', 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 4.3 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 4.4, which we give first.

LEMMA 4.4. *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 either to 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 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 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 4.4, 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 4.3 and 4.4, 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 4.3. *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 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 4.5. *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, 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 4.6. *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 4.5. 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 4.7. *It takes linear time to compute $R(y)$ for every $y \in V(G)$ such that $r(y) < l(D(x))$.*

Proof. By Lemma 4.6, 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 $R_i(y) = \mathtt{min}\{R(P)|P$ 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 $(v_n, v_{n-1}, ..., v_1)$ be a topological sort of D_r such that the arcs are directed to the right. Then $R_i(v_i) = \mathtt{min}\ \{R_1(v_i)\} \cup \{R_j(v_j)|(v_i, v_j)$ is an arc of $D_r\}$. This may be computed inductively in right-to-left order in linear time. For any $y = v_k$ such that $r(y) < l(D(x))$, $R(y) = R_k(v_k)$, since y has at most $k-1$ successors in D_r.

5 A certifying algorithm for permutation 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 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. Such an orientation is called a *transitive orientation.*

A linear-time algorithm for finding a transitive orientation of a comparability graph is given in [12]. Let us denote this algorithm by "TO." TO represents its orientation implicitly by giving a linear extension (topological sort) of the orientation that it produces. This allows it to be applied to $\overline{G}$ in time linear in the size of G.

Let us now consider what happens when TO 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 it must contain a *transitivity violation*, which is a pair $\{(a, b), (b, c)\}$ of directed edges in series such that ac is not an edge of G, hence (a, c) is not a directed edge in the orientation. No linear-time algorithm for finding a transitivity violation 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.

TO gives rise to a linear time bound for recognizing permutation graphs, which is also given in [12]. The algorithm makes use of the following:

THEOREM 5.1. *[14, 6] 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. Together, these two linear arrangements are a permutation model of G.

When G is not a permutation graph, the procedure in [12] 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.

5.1 The certificates A linear-time authentication algorithm for a proposed permutation model is given in [12]. The permutation model is therefore a weak certificate.

It remains to describe the rejection certificate. Let us consider an undirected graph 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 are transitive.

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. 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.

DEFINITION 5.1. *A* transposed path *is a path in* Γ_G *between an arc* (x, y) *of* G *and its transpose* (y, x).

THEOREM 5.2. *[5, 6] An undirected graph* G *is a comparability graph iff it has no transposed path.*

A transposed path therefore serves as a certificate that a graph is not a comparability graph. It follows from Theorem 5.1 that a transposed path either in G or in $\overline{G}$ serves as a certificate that G is not a permutation graph.

Let us require that the user supply an adjacency-list representation where the edge lists are sorted by the identifier of the neighbors. The authentication algorithm for the transposed path consists of verifying that every pair of consecutive arcs in the transposed path is in the Γ relation. For each such pair, this requires verifying the presence/absence of three possible edges in G. The certificate gives pointers to the locations in the sorted adjacency lists where the presence or absence of an edge can be checked. The authentication algorithm therefore takes time proportional to the length of the transposed path, which we will show is $O(n)$. This is therefore a strong certificate.

5.2 Finding a transitivity violation The algorithm for recognizing permutation graphs given in [12] uses TO 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 a transitivity violation.

In this subsection, we describe how to find a transitivity violation 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 TO 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 TO. In the next subsection, we show how to find a certificate, given the transitivity violation.

LEMMA 5.1. *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 the other 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 a transitivity violation $\{(a, b), (b, c)\}$. 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 5.2. *Let* G*,* D*, and* D' *be as in Lemma 5.1. Given a three-cycle in* $D \cup D'$ *or* $D^T \cup D'$*, it takes* $O(1)$ *time to return a transitivity violation 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 a transitivity violation in it.

LEMMA 5.3. *Let* G*,* D*, and* D' *be as in Lemma 5.1, 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 does 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 $x \in V$, hence $O(n+m)$ time to label each $x \in V$ with $|N^-(x)|$ in D'.

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'. If for each $i \in \{0, 1, ..., n-1\}$ there exists $x \in V$ such that $p(x) = i$, 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 and 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 5.3 also applies to $D^T \cup D'$. The linear time bound for finding the transitivity violation now follows by Lemma 5.1.

5.3 Finding the rejection certificate, given a transitivity violation In this subsection, we give a linear-time algorithm to find a transposed path in G, given a transitivity violation in the orientation assigned to it by TO. We show how to apply the algorithm to $\overline{G}$ in time linear in the size of G. Since we have shown how to find a transitivity violation in G or in $\overline{G}$ in linear time when G is not a permutation graph, this will complete the certifying algorithm.

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.

The problem of verifying that G is a permutation graph reduces in linear time to the problem of verifying that a set of prime induced subgraphs are permutation graphs [12]. A transposed path in an induced subgraph or its complement is a transposed path in G or its complement. Therefore, when G is not a permutation 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.

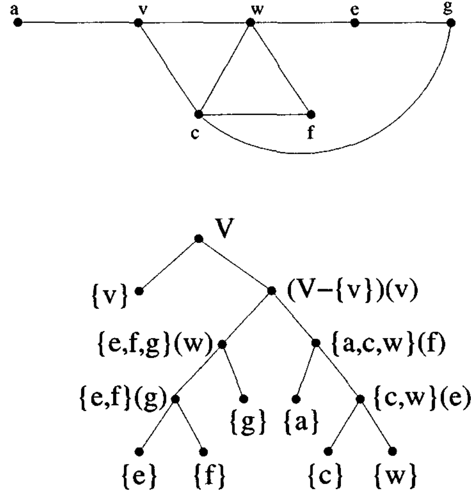

Figure 1: A key step of the transitive orientation algorithm of [12] is to refines a partition of the set V of vertices of a prime graph by iteratively choosing a partition class Y and a *pivot vertex* $x \notin Y$ and splitting Y into neighbors and non-neighbors of x. In this illustration, the partition starts with $\{\{v\}, V-\{v\}\}$. $V-\{v\}$ is split into $\{e, f, g\}$ and $\{a, c, w\}$ with pivot on vertex v. Then $\{e, f, g\}$ is split into $\{e, f\}$ and $\{g\}$ with pivot vertex w, $\{a, c, w\}$ is split into $\{a\}$ and $\{c, w\}$ with pivot f, etc. The procedure halts when all sets have cardinality one. The tree depicts the Hasse diagram of the sets that appear at any point during the procedure. Our modification of the algorithm creates this Hasse diagram for use as a data structure, together with the pivot label at each internal node that gives the pivot that split the corresponding set, and a record of which child contains elements that are adjacent to the pivot.

THEOREM 5.3. *[5, 6] 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.*

We show how to modify TO so that it creates a record that allows us to find a path of length $O(n)$ in Γ between any two arcs that are included in its orientation of a prime graph G.

If G is not a comparability graph, the algorithm produces an orientation with a transitivity violation $\{(a, b), (b, c)\}$. Since (a, b) and (b, c) are included in its orientation of G, the modified algorithm gives us a path P in Γ_G of length $O(n)$ from (a, b) to (b, c). By the definitions of Γ_G and of a transitivity violation, $((b, c), (b, a))$ is an edge of Γ_G. Appending this edge to the path gives a transposed path from (a, b) to (b, a).

We may assume henceforth that G is prime and not a permutation graph. TO begins with a partition $\mathcal{P} = \{\{v\}, V-\{v\}\}$ of the vertices V of G, where v is an arbitrary element of the vertex set V. 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, $Y_a = Y \cap N(x)$ and $Y_n = Y - N(x)$. Let $\mathcal{P} := (\mathcal{P} - \{Y\}) \cup \{Y_a, Y_n\}$.

Figure 1 illustrates the procedure. Because G is prime, it is always possible to partition V down to one-element sets. A linear arrangement of the partition classes is maintained, so that when Y is split, Y_n and Y_a, they 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. This ensures that the final ordering will be a topological sort of a transitive orientation when G is a comparability graph.

DEFINITION 5.2. *When Y is split into Y_a and Y_n by a pivot vertex x, let us call Y_a the* winners *and Y_n the* losers *of the split. At the end of the vertex partition, a vertex is* undefeated *if it started out in a non-singleton class and was never a loser in a split. Let (a, b) be an arc from Y_a to Y_n. By the definition of Y_a, and Y_n, (a, x) is an arc of G. Let (a, x) be the* parent *of (a, b) and (x, a) be the* parent *of (b, a). (Note that parent and child are related by Γ.)*

Example. In Figure 1, w is the undefeated vertex. The parent of arc (c, w) is found by retrieving the pivot label e of the least common ancestor of $\{c\}$ and $\{w\}$. This arc must be incident to e and whichever of c or w was on the neighbors' side of the split when e was used to split $\{c, w\}$. The parent of (c, w) is, therefore, (e, w).

LEMMA 5.4. *If the initial partition is $\{\{v\}, V - \{v\}\}$, then after the vertex partition is complete, the parent relation has the following properties:*

1. *It is a spanning forest of Γ_G.*
2. *Each tree of the forest is rooted at an arc incident to v.*
3. *Let w be the undefeated vertex. The parent relation on arcs of the form $\{(w, x) | x \in V - w\}$ induce a subtree in the parent relation rooted at (w, v), and their transposes induce a subtree in in the parent relation rooted at (v, w).*

Proof. (Sketch) At each point during the progressing partition, let the *exposed arcs* be those whose endpoints do not lie in a single partition class. By induction on the number of pivots, the conditions of the lemma apply to the subgraph of Γ_G induced by the exposed arcs. When the procedure terminates, all arcs are exposed.

TO runs the vertex partition algorithm twice. The first time, it finds an undefeated vertex w. The arcs incident to w induce two subtrees T_a and T_b in the parent relation whose edges are all edges in Γ_G, rooted at (v, w) and (w, v), by Lemma 5.4, Part 3. The second time, it starts with the partition $\{\{w\}, V-\{w\}\}$ to find a spanning forest $\mathcal{T}$ of Γ_G, by Part 1 of the lemma. Together, the edges of T_a, T_b, and $\mathcal{T}$ induce two spanning trees T_1 and T_2 of Γ_G rooted at (v, w) and at (w, v), by Part 2 of the lemma. The nodes of T_1 are those arcs that are included in the orientation given by the algorithm, and the nodes of T_2 are those that are excluded. If (a, b), (b, c) are a transitivity violation in the orientation assigned by TO, then the path in T between (a, b) and (b, c) completes the certificate.

A spanning tree in $\Gamma_{\overline{G}}$ does not have size that is linear in the size of G. To operate on $\overline{G}$, we give T_1 and T_2 implicitly by computing a data structure that allows us to find the parent of an arc easily. This data structure consists of two Hasse diagrams, such as those of Figure 2, one for each of two vertex partitions of TO.

For the first vertex partition, the Hasse diagram is a tree rooted at V with children $\{v\}$ and $V - \{v\}$, and one leaf for each subset of V of cardinality one. Note that the height of each Hasse diagram is at most $n - 1$. The first Hasse diagram is used for finding the parent of any arc incident to the undefeated vertex w, while the second is used for finding the parent of any other arc.

As illustrated in Figure 2, we may find the parent of any edge in either of the two vertex partitions. We can therefore find a path from any arc in T_1 to the root of T_1, by Lemma 5.4. Any two arcs included in the orientation produced by the algorithm are in T_1, so the disjoint portions of their paths to the root yield a path from one to the other.

LEMMA 5.5. *It takes $O(n + m)$ time to find a path of length $O(n)$ in Γ_G between any pair of arcs in the orientation assigned by TO to an arbitrary graph G or its complement.*

Proof. A linear-time algorithm is given in [12] for the vertex partitioning procedure. The procedure is easily modified to record the Hasse diagram for sets that appear in the partitions. The same vertex partition will do for orienting G or $\overline{G}$; only the rule for ordering children to obtain a leaf order corresponding to a topological sort of the orientation differs.

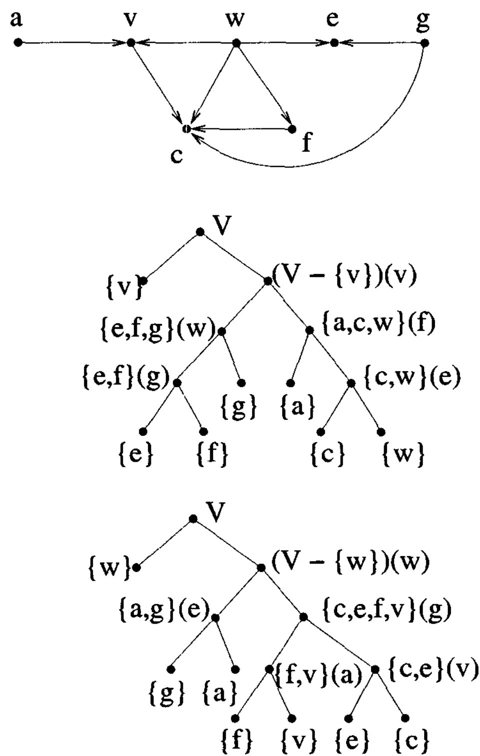

Figure 2: TO performs two vertex partitions. The first is to find the undefeated vertex w, and the second starts with partition $\{\{w\}, V - \{w\}\}$ to find a topological sort of an orientation of G that must be transitive if G is a comparability graph. It orders the second tree so that its left-to-right leaf order is this topological sort. In this illustration, the graph is not a comparability graph, and the orientation results in a transitivity violation, $\{(a, v), (v, c)\}$. Using the Hasse diagrams for the two sorts, we may reconstruct how the algorithm arrived at this orientation of these two edges. Since (v, c) is not incident to w, we start with the second Hasse diagram. The least-common ancestor of (v, c) has pivot label g, so (g, c) is the parent of (v, c). The least-common ancestor of (g, c) is labeled with pivot w, so (w, c) is the parent of (g, c). Since (w, c) is incident to w, we switch to the first Hasse diagram. The least-common ancestor of (w, c) yields (w, e) as the parent, and the least common ancestor of (w, e) yeidls (w, v) as its parent. This arc is the root of a tree in the spanning forest of Γ_G that contains the arcs that are included in the orientation. In a similar way, we find that (w, v) is the parent of the other arc in the transitivity violation, (a, v). The edges of these paths are edges in Γ_G, as is $((a, v), (c, v))$ since $((a, v), (v, c))$ is a transitivity violation. Joining these edges of Γ_G gives the transposed path $((v, c), (g, c), (w, c), (w, e), (w, v), (a, v), (c, v))$.

To find the ancestors of an arc (a, b) in T_1 that are given in one of the Hasse diagrams, 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 of A_1 of (a, b) in the Hasse diagram. Suppose by induction that the least-common ancestor A_k in the Hasse diagram has been found for some ancestor (x, y) of (a, b) in T_1. Let z be the pivot label of this ancestor. Search upward from z until a marked node is 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 T_1 is $O(n)$, and the length of the path is $O(n)$. The path in Γ_G that we have described consists of at most four such paths: at most two from $\mathcal{T}$, and at most two from T_a or T_b. Thus, this path has length $O(n)$.

Remark: It is not hard to show that the length of the returned path is at most $3n - 6$, which gives a bound of $3n - 5$ on the number of Γ links in the rejection certificate.

References

[1] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, JCSS, 13 (1976), pp. 335–379.

[2] A. Brandstädt and V. B. Le and J. P. Spinrad, *Graph Classes : A Survey*, SIAM Monographs on Discrete Mathematics and Applications, 1999.

[3] D. G. Corneil, S. Olariu, and L. Stewart, *The ultimate interval graph recognition algorithm?* Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 9 (1998), pp. 175–180.

[4] G. A Dirac, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25 (1961) pp. 71–76.

[5] T. Gallai, *Transitiv orientierbare Graphen*, Acta Math. Acad. Sci. Hungar., 18 (1967), pp. 25–66.

[6] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[7] 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*, Theoretical Computer Science, 234 (2000), pp. 59–84.

[8] W. L. Hsu, *A simple test for interval graphs*, Lecture Notes in Computer Science, 657 (1993), pp. 11–16.

[9] W. L. Hsu and R. M. McConnell, *PC Trees and Circular-Ones Arrangements*, Theoretical Computer Science, to appear.

[10] N. Korte and R. H. Möhring, *An incremental linear-time algorithm for recognizing interval graphs*, SIAM Journal on Computing, 18 (1989), pp. 68–81.

[11] C. Lekkerkerker and D. Boland, *Representation of finite graphs by a set of intervals on the real line*, Fund. Math. 51 (1962), 45–64.

[12] R. M. McConnell and J. P. Spinrad, *Modular decomposition and transitive orientation*, Discrete Mathematics, 201 (1999), pp. 189–241.

[13] K. Mehlhorn and S. Näher, *The LEDA Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999.

[14] A. Pnueli, A. Lempel, and S. Even, *Transitive orientation of graphs and identification of permutation graphs*, Canad. J. Math., 23 (1971), pp. 160–175.

[15] 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.

[16] 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.

[17] H. Wasserman and M. Blum, *Software reliability via run-time result-checking*, JACM, 44 (1997), pp. 826–849.