# Lex-BFS and Partition Refinement, with Applications to Transitive Orientation, Interval Graph Recognition and Consecutive Ones Testing.

Michel HABIB* Ross MCCONNELL † Christophe PAUL‡
Laurent VIENNOT §

**Abstract**

By making use of lexicographic breadth first search (Lex-BFS) and partition refinement with pivots, we obtain very simple algorithms for some well-known problems in graph theory.

We give an $O(n + m \log n)$ algorithm for transitive orientation of a comparability graph, and simple linear algorithms to recognize interval graphs, convex graphs, $Y$-semichordal graphs and matrices that have the consecutive-ones property.

Previous approaches to these problems used difficult preprocessing steps, such as computing PQ trees or modular decomposition. The algorithms we give are easy to understand and straightforward to prove. They do not make use of sophisticated data structures, and the complexity analysis is straightforward.

**Keywords**

algorithm, data-structure, partition refinement, graph, boolean matrix

## 1 Introduction

Some efficient algorithms for various classes of graphs and boolean matrices are presented. These classes are comparability, chordal, interval graphs and their complements. To this aim a general framework, namely partition refinement [14, 16], is used. This framework allows a unified and more general treatment of problems on these classes, such as transitive orientation of a comparability graph or its complement, recognition of an interval graph or its complement, and consecutive ones testing of boolean matrices. We give efficient solutions to these problems that do not use the preprocessing steps of computing PQ trees or modular decomposition.

*LIRMM, Montpellier, France; habib@lirmm.fr
†Computer Science and Engineering Department, University of Colorado at Denver, rmcconne@carbon.cudenver.edu
‡LIRMM, Montpellier, France; paul@lirmm.fr
§LIAFA, Paris, France;lavie@litp.ibp.fr

All graphs considered in this paper are finite and simple. A directed graph is *transitive* if, whenever $(a, b)$ and $(b, c)$ are arcs, $(a, c)$ is also an arc. A graph is a *comparability graph* if its edges can be assigned orientations so that the resulting directed graph is transitive and acyclic, hence a partial order.

An *interval graph* is a graph that can be modeled by assigning to each vertex an interval on the set of integers, such that two vertices are adjacent in the graph if and only if their intervals intersect. A graph is a *co-comparability graph* if its complement is a comparability graph. It is readily seen that an interval graph is a co-comparability graph, since two vertices are an edge in the complement if and only if one of the intervals comes before the other, and this relation is transitive.

A *chordal graph* is an undirected graph where every induced cycle on four or more vertices has a chord. It is not hard to see that an interval graph must be chordal. In fact, a graph is an interval graph if and only if it is chordal and its complement is a comparability graph [9].

Chordal graphs are characterized by the existence of a *perfect elimination ordering* of their vertices, which is defined as follows. A *clique* is a set of vertices inducing a complete subgraph. An ordering $x_1, \ldots, x_n$ of vertices is a perfect elimination ordering of a graph $G = (V, \mathcal{E})$ if the neighborhood of each vertex $x_i$ is a clique of the induced subgraph $G_{\{x_i, \ldots, x_n\}}$.

A graph is chordal if and only if there exists an arrangement of its maximal cliques into a tree such that the maximal cliques containing a given vertex always induce a connected subtree [8]. Such a tree is called a *clique tree. Interval graphs* are the chordal graphs admitting a clique tree that is a chain, or equivalently, a numbering of their maximal cliques such that the maximal cliques containing a given vertex occur consecutively. Such a chain is called a *clique chain.* If there are $k$ cliques, this associates with each vertex an interval on the integers from one to $k$, namely, the subscripts of the cliques that contain the vertex. The result is an interval representation of the graph, since two vertices are adjacent if and only if they reside in a common clique.

A boolean matrix has the *consecutive ones property* if its columns can be reordered so that the ones in each row are consecutive.

In many applications, the modular decomposition [5] appears as a preprocessing step of efficient algorithms for transitive orientation [12], and interval graph recognition [10]. The first recognition algorithm, presented in [2], uses a complex procedure for computing a data structure called the PQ-tree. Later, simpler algorithms based on Lex-BFS have been discovered: in [11], a simplification of the PQ-tree, called the MPQ-tree is used, while in [10], the modular decomposition is used, and in [12], a transitive orientation of the complement is used.

Either explicitly or implicitly, most of these disparate algorithms make use of an operation that is sometimes called *pivot.* In a pivot, a partition of the vertices is refined by splitting a partition class according to its adjacency to a selected vertex that is not a member of that partition class. The evolution of the partition refinement yields information about the structure of the graph, which is then used to solve the problem. Pivoting is used in finding twins, recognizing chordal graphs [15], recognizing permutation graphs [12], finding a transitive orientation [12], and modular decomposition [12, 4]. Lex-BFS is a special case of pivoting, and is used for recognizing chordal graphs [15].

In this paper, we attempt to show that pivoting is fundamental to the so-

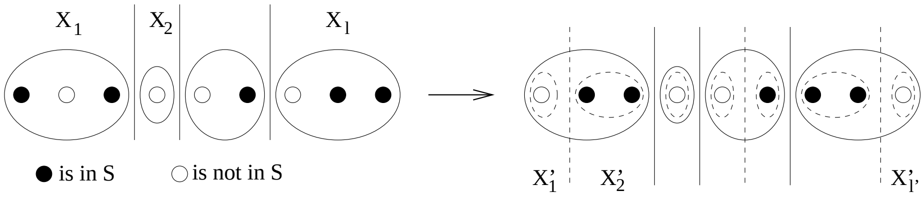


Figure 1: The partition refinement of $(\mathcal{X}_1, \ldots, \mathcal{X}_l)$ into $(\mathcal{X}'_1, \ldots, \mathcal{X}'_l)$ according to the subset $S$ of black elements.

lution of these problems, by showing how efficient algorithms for them can be obtained without much recourse to other techniques. The pivot may be viewed as a generalization to graphs of the Quicksort pivoting rule, used for sorting integers. This general approach was originally put forth in [14] and in [16], who showed how it can lead to a simpler conceptual framework for developing algorithms for some of these problems. By generalizing it to arbitrary set families, we are able to use it to manipulate cliques of interval graphs.

We first show how the $O(n + mlogn)$ transitive orientation algorithm presented in [12] can be adapted so that it does not require the formidable step of pre-computing the modular decomposition. We then present an $O(n + m)$ interval graph recognition algorithm that uses a clique tree for pivoting. A clique tree of a chordal graph can be computed with Lex-BFS in linear time (see [6]). In order to use the same algorithm for the consecutive one's property problem, we propose an adaptation of Lex-BFS that takes as input the cliques of a graph. This Lex-BFS version gives linear time and space algorithms for the recognition of convex graphs and $Y$-semichordal graphs.

## 2 Refining a Partition by Pivoting

All the algorithms we propose are based on the general framework of Algorithm 1, which refines a partition of a set $E$ according to a subset $S$ of $E$.

A partition is an ordered collection of disjoint subsets of $E$ called *classes*, whose union is $E$. A set $S \subseteq E$ is given as a parameter. We refine the partition by splitting each partition class $I_a$ into two subsets, $I_a \cap S$ and $I_a \setminus S$. At the same time, we maintain an order on the partition classes as they evolve. Figure 1 gives an illustration.

**Algorithm 1:** Partition Refinement

**Input**: an ordered partition $L = (\mathcal{X}_1, \ldots, \mathcal{X}_l)$ of a set $E$ and a subset $S \subseteq E$.

**Output**: a refined partition $L' = (\mathcal{X}'_1, \ldots, \mathcal{X}'_m)$.

**begin**
- **for each** *class* $\mathcal{X}_a$ **do**
  - let $\mathcal{Y}$ be the members of $\mathcal{X}_a$ that are in $S$
  - **if** $\mathcal{Y}$ *is not empty and* $\mathcal{Y} \neq \mathcal{X}_a$ **then**
    - remove $\mathcal{Y}$ from $\mathcal{X}_a$ and insert it next to $\mathcal{X}_a$ in $L$

**end**

After the refinement of the partition, no class properly overlaps $S$: any class $\mathcal{X}'_a$ verifies either $\mathcal{X}'_a \subseteq S$ or $\mathcal{X}'_a \cap S = \emptyset$. Depending on the use of the routine, $\mathcal{Y}$ can be inserted immediately behind or immediately in front of $\mathcal{X}_a$.

The refinement can be performed in $O(|S|)$ time by using the following data structure. All the elements of $E$ are stored in a doubly linked list. Each class consists of an interval in this list, and is implemented with a structure that has a pointer to its first element and a pointer to its last element. Each element keeps a pointer to the class that contains it. To maintain an ordering on the classes, these class structures are stored in a doubly linked list.

During the refinement, each element in $S$ is simply removed from the list and inserted at the end of its new class. This preserves the initial ordering of the vertices inside the classes when $S$ is sorted according to this ordering. When it is not important to keep this ordering, it may be simpler to store the vertices in an array, and to exchange the element to be removed and the first (or the last) element of the class being split. The bounds of the new class and the class being split must then be updated.

In graph algorithms, $E$ is usually the vertex set and $S$ is the neighborhood of a pivot vertex. Note that the procedure for refining the partition by the complement of $E \setminus S$ of $S$ produces an identical result if suitable adjustments are made to the ordering rule used to determine whether $\mathcal{Y}$ should be placed before or after $\mathcal{X}_a$ in Algorithm 1. In graph algorithms, this means that, given the adjacency-list representation of a graph, we can run the partition refinement routine on the complement of the graph directly, without having to compute an adjacency-list representation of the complement. This property was used in [12] to recognize permutation graphs, which are those comparability graphs whose complement is also a comparability graph.

# 3 Lex-BFS Orderings

Standard breadth-first search fails to specify completely the order in which vertices must be visited. Lex-BFS imposes additional constraints, by breaking ties according to a rule that we describe below. This guarantees that the order in which vertices are visited has certain desirable properties. We call *Lex-BFS ordering* the order in which the vertices are visited. Lex-BFS was introduced in [15] to recognize chordal graphs. Algorithm 2 is one way to implement Lex-BFS. Since there is only one pivot on each vertex, the time bound is clearly $O(n+m)$. An example is given in Figure 2.

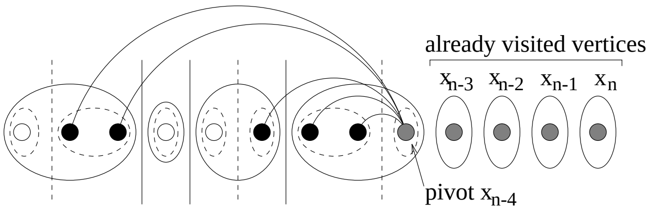


Figure 2: Refining a partition towards a Lex-BFS ordering according to the neighborhood of the pivot, the currently visited vertex by the Lex-BFS.

Given a graph $G$ and a partial numbering $\pi$ of the vertices of $G$, we define

**Algorithm 2:** Lex-BFS Ordering

**Input**: a graph $G = (V, \mathcal{E})$.

**Output**: a Lex-BFS ordering $\pi^{-1}(1), \ldots, pi^{-1}(n)$ of the vertices.

```
begin
    let L be the one element ordered list (V)
    i ← n
    while there exists a non-singleton class in L = (X_1, ..., X_l) do
        let X_a be the last class made of non-visited vertices
        remove a vertex x from X_a
        π(x) ← i
        i ← i − 1
        for each class X_b, b ≤ a do
            let Y be the members of X_b that are adjacent to x
            if Y is nonempty and not equal to X_b then
                remove Y from X_b
                insert Y immediately behind X_b in L
end
```

$RN(x)$ to be the neighbors to the right of $x$, namely, the set $\{y : y \in N(x)$ and $\pi(y) > \pi(x)\}$. Partway through execution of the above algorithm, not all of the eventual members of $RN(x)$ are yet known, so in this context, we we will find it convenient to let $RN(x)$ be defined to be the vertices currently known to belong to the right of $x$ in the eventual ordering. Specifically, if $\pi(x)$ is already defined, $RN(x)$ is defined as before, and if not, $RN(x)$ is the neighbors of $x$ that have already been assigned numbers.

An important function is $label(x)$, which denotes the sequence of $\pi$ labels of $RN(x)$ in ascending order. It is not hard to verify that the algorithm maintains the invariant that two un-numbered vertices $x$ and $y$ are in the same partition class if and only if $label(x) = label(y)$, and that if the reverse of $label(x)$ precedes the reverse of $label(y)$ in lexical order, then $x$'s partition class is before $y$'s. Thus, in the final ordering, the labels of the vertices are in lexical order.

### Lex-BFS Orderings and Chordal Graphs

A graph $G$ is chordal if and only if the ordering of vertices produced by Lex-BFS is a perfect elimination ordering [15]. Thus, an algorithm for recognizing chordal graphs is to use Lex-BFS to obtain an ordering, and then check whether this ordering is a perfect elimination ordering. The algorithm 3 is one way to do this.

For the correctness, note that if $\pi$ is a perfect elimination ordering, then $\{x\} \cup RN(x)$ is a clique $C$, where $x$ is its leftmost member and $parent(x)$ is its next leftmost member. The check obviously cannot fail. If it is not a perfect elimination ordering, then for some $x$, $x \cup RN(x)$ is not a clique. Without loss of generality, let $x$ be the rightmost vertex in $\pi$ with this property. By our choice of $x$, $parent(x)$ fails to have as a neighbor some vertex to its right that is a neighbor of $x$, so the check fails.

For the time bound, finding $RN(x)$ for all $x$ obviously takes $O(n+m)$ time. We may get all $RN$ lists in sorted order by concatenating them and using a

**Algorithm 3:** Chordality test

**Input**: a graph $G = (V, \mathcal{E})$, and a numbering $\pi$ of vertices

**Output**: Returns TRUE if $\pi$ is a perfect elimination ordering

```
begin
    for each vertex x do
        let RN(x) be its neighbors to the right
        let parent(x) be the leftmost member of RN(x) in π
    Let T be the tree defined by the parent pointers
    for each vertex x in T in postorder do
        check that (RN(x) \ parent(x)) ⊆ RN(parent(x))
    if no check failed then return TRUE
    else return FALSE
end
```

two-pass radix sort that sorts all entries by $\pi$ value as the secondary key, and original $RN$ list as primary key. This requires $n$ buckets for each pass, and takes $O(n+m)$ time. Given the lists in sorted order, the remainder of the operations are trivial to carry out in $O(n+m)$ time.

Recall that if $G$ is chordal, it is possible to find a clique tree, that is, to arrange the maximal cliques into a tree such that for each vertex, the subtree induced by the cliques that contain $x$ are connected. The following variant of Algorithm 3 does this:

**Algorithm 4:** Clique tree

**Input**: $G$ is a chordal graph, and $\pi$ is a perfect elimination ordering

**Output**: A clique tree $\mathcal{T}$ of $G$

```
begin
    Let T be defined as in Algorithm 3
    Let r be the root of T
    for each vertex x in T except the root, in preorder do
        if (RN(x) \ parent(x)) ≠ RN(parent(x)) then
            Create a new clique C = {x}
            C(x) ← C
            parent(C) ← C(parent(x))
        else
            C(parent(x)) ← C(parent(x)) ∪ {x}
            C(x) ← C(parent(x))
end
```

The $O(n+m)$ time bound follows from the time bound of operations in Algorithm 3, and the fact that all operations in an iteration of the last *for* loop may be charged at $O(1)$ to $x$ and $O(1)$ to each element of the list $RN(x)$. The sum of cardinalities of $RN$ lists is $O(m)$.

For the correctness, note first that for each vertex $x$, $RN(x)$ is a subset of the ancestors of $x$ in $T$. This is true for the root. Suppose it is true for any vertex at depth $k$, and assume that $x$ is at depth $k+1$. The parent of $x$ is the earliest member of $RN(x)$ in $\pi$. Since $RN(x)$ is a clique, $RN(x) \setminus parent(x)$ is a subset of $RN(parent(x))$. By the inductive hypothesis, $RN(x) \setminus parent(x)$ is a subset of the ancestors of $parent(x)$.

Next, adopt as an inductive hypothesis that just after each vertex is processed, the current set of cliques reflects the maximal cliques of the subgraph of $G$ induced by the set of processed vertices. As a base case, this is obviously true just before second vertex is processed. The correctness of the set of cliques after the inductive step is immediate from the fact that the members of $RN(x)$ have already been processed in the preorder traversal, and $\{x\} \cup RN(x)$ is a clique.

Finally, we show that after each vertex is processed, the parent relation is a clique tree on the subgraph induced by the set of processed vertices. To do this, we show that for an arbitrary processed vertex $y$, the cliques containing $y$ induce a connected subtree of this tree. As a base case, it is true just after $y$ is processed, since it is contained in only one clique of the tree. Suppose it is true just before some subsequent vertex $x$ is processed. If no new clique containing $y$ is created, it continues to be true. So assume that processing $x$ creates a new clique $C$ and $y$ is contained in $C$. It suffices to show that the parent of $C$ is a pre-existing clique that contains $y$. For each processed vertex $z$, $C(z)$ contains $\{z\} \cup RN(z)$. In particular, $C(parent(x))$ contains $\{parent(x)\} \cup RN(parent(x))$. Since $\{parent(x)\} \cup RN(parent(x))$ contains $RN(x)$, $C(parent(x))$ contains $y$. The parent of the new clique is a pre-existing clique containing $y$.

It follows that the tree is a clique tree for $G$ after all vertices are processed.

**Lex-BFS orderings and co-chordal graphs**

Note that in Algorithm 2, the same result could be achieved by removing the *non-neighbors* of the pivot from $X_b$ and placing them *before* $X_b$. Thus, the only asymmetry in the treatment of neighbors and non-neighbors is the decision to place the non-neighbors before the neighbors in the ordering of the refined classes. It follows that if this rule is changed so that the neighbors are placed before $X_b$, rather than after, the resulting ordering is that which would be produced by a Lex-BFS on the complement graph. Changing the ordering rule does not affect the time bound of Algorithm 2, so we get the following:

**Algorithmic Result 1** *If $G$ is a co-chordal graph, it is possible to produce a Lex-BFS ordering of $\overline{G}$ in $O(n+m)$ time.*

To recognize whether a graph is a co-chordal graph, we need only verify that that this ordering is a perfect elimination ordering on the complement. We give the following adaptation of Algorithm 3:

For the correctness, let $\overline{RN}(x)$ be the non-neighbors to the right of $x$ in $\overline{G}$. To run Algorithm 3 on $\overline{G}$, we use the same parent function that we use in Algorithm 5. Instead of using $RN(x)$, we would use $\overline{RN}(x)$, and check whether $\overline{RN}(x) \setminus parent(x)$ is a subset of $\overline{RN}(parent(x))$. This happens if and only if $RN(parent(x))$ fails to be a subset of $RN(x)$, so the two sets of tests are equivalent. The algorithm returns TRUE if and only if Algorithm 3 returns TRUE when given $\overline{G}$ and the same Lex-BFS ordering of $\overline{G}$ as input.

For the time bound, creating and sorting the $RN$ lists is accomplished just as it was in Algorithm 3. To compute $parent(x)$, mark all neighbors of $x$,

**Algorithm 5:** Co-chordalilty test

**Input**: a graph $G = (V, \mathcal{E})$, and a numbering $\pi$ of vertices

**Output**: Returns TRUE if $\pi$ is a perfect elimination ordering on $\overline{G}$

```
begin
    for each vertex x do
        let RN(x) be its neighbors to the right in G
        let parent(x) be the leftmost non-neighbor to its right
        Let T be the tree defined by the parent pointers
    for each vertex x in T in postorder do
        check that RN(parent(x)) is a subset of RN(x)
    if no check failed then return TRUE
    else return FALSE
end
```

and then moving rightward from $x$ in the ordering given by $\pi$, find the first unmarked vertex. This is $parent(x)$. Then unmark the neighbors of $x$. Except for the $O(1)$ time spent at $parent(x)$, the time is charged to marked neighbors of $x$, and takes $O(1 + |N(x)|)$. Computing this for all $x$ thus takes $O(n + m)$. Given the sorted $RN$ lists, the time spent in the subset tests can be charged to members of $RN$ lists and are thus $O(n + m)$.

Algorithm 4 can be adapted in a similar way. We use it below to recognize whether the complement of a graph is an interval graph.

**Algorithm 6:** Co-clique tree

**Input**: A co-chordal graph $G$ and a co-Lex-BFS ordering $\pi$ of $G$

**Output**: A clique tree of the complement of $G$, where each clique is represented by listing its non-members

```
begin
    Let T and RN and be defined as in Algorithm 5
    for each vertex x in T in preorder do
        Let R be the members of RN(x) to the right of parent(x)
        if R ≠ RN(parent(x)) then
            Create a new empty list C
            C(x) ← C
            parent(C) ← C(parent(x))
            leftmost(C) ← x
        else
            C(x) ← C(parent(x))
            leftmost(C(x)) ← x
    for each empty list C created do
        Insert the neighbors of leftmost(C)) in C
end
```

The only difference between running this algorithm on $G$ and $\pi$ and running *Algorithm* 4 on $\overline{G}$ and $\pi$ is the condition of the $for$ loop, and that each final clique is represented by the neighbors in $G$ of its leftmost vertex. That the condition in the $for$ loop is equivalent follows from the fact that $RN(x)$ is the neighbors to the right of $x$ in $G$, instead of in $\overline{G}$. Since $\pi$ is a co-perfect elimination ordering of $\overline{G}$, the complement of the neighborhood in $G$ of the leftmost vertex gives the members of the clique. Thus, the neighborhood in $G$ of the leftmost vertex is the claimed representation of the clique with its non-members. The $O(n+m)$ bound on the steps it shares with Algorithm 5 follows from the time bound of that algorithm. The operations inside an iteration of the *for* loop can clearly be charged to $x$ and members of $RN(x)$, giving an $O(n+m)$ bound for the algorithm. This gives the following:

**Algorithmic Result 2** *Algorithm 5 recognizes co-chordal graphs in $O(n+m)$ time, and a clique tree on the complement of a co-chordal graph may be found in $O(n+m)$ time by algorithm 6*

### Lex-BFS Orderings and Transitive Orientation

A *module* of a graph is a set $M$ of vertices such that for each vertex $x$ not in $M$, either every member of $M$ is adjacent to $x$, or no member of $M$ is adjacent to $x$. The entire vertex set, its singleton subsets, and the empty set are *trivial modules.* A graph with only trivial modules is a *prime* graph. It is easily seen that if $X$ and $Y$ are disjoint modules, then $X$ and $Y$ are either *adjacent* (every member of $X \times Y$ is an edge of $G$) or *nonadjacent* (no member of $X \times Y$ is an edge of $G$). A *modular partition* of $G$ is a partition $\mathcal{P}$ of $V$ such that every member of $\mathcal{P}$ is a module. A modular partition always exists, since the singleton subsets of $V$ are trivially a modular partition. Since all sets in a modular partition are disjoint, their adjacency relation defines a *quotient graph* $G/\mathcal{P}$ whose vertices are the members of $\mathcal{P}$. The quotient graph is isomorphic to the subgraph induced by any set consisting of one vertex from each member of $\mathcal{P}$.

If a comparability graph contains nontrivial modules, then they give a way of breaking the transitive orientation problem into smaller pieces, as follows [7].

**Algorithm 7:** Transitive orientation

**Input**: A comparability graph $G$ and a partition $\mathcal{P}$ of $V$ where each partition class is a module.

**Output**: A transitive orientation of $G$

```
begin
    Let F be an empty set of arcs
    for each X ∈ P do
        Let F_X be any transitive orientation of the edges of G_X
        F ← F ∪ F_X
    Let F' be any transitive orientation of G/P
    for each arc (X,Y) ∈ F' do
        for each edge xy of G where x ∈ X and y ∈ Y do
            F ← F ∪ (x,y)
    return F
end
```

**Algorithmic Result 3** *If $G$ is a comparability graph, then Algorithm 7 produces a transitive orientation of $G$.*

**Lemma 1** *If $G$ is a prime co-comparability graph and $(x_1, x_2, \ldots, x_n)$ is a Lex-BFS numbering of $V$, then there is a transitive orientation of $\overline{G}$ where $x_1$ is a source, and another where it is a sink.*

**Proof:** It suffices to show that there is a transitive orientation where $x_1$ is a sink, since reversing the directions of the arcs in this orientation gives another where $x_1$ is a source.

Let $V = X_1, X_2, \ldots, X_k = \{x_1\}$ be the sequence of partition classes that contain $x_1$ during the course of the execution of the Lex-BFS. These classes are always first in the sequence of partition classes, since they contain $x_1$, which is first in the final partition. Each $X_i : 1 \leq i < k$ is split into $X_{i+1}$ and a class $Y$ by some pivot $z$ not in $X_i$, since the graph is prime and $X_i$ is therefore not a module. Note that every vertex in $Y$ is adjacent to $z$, and every vertex in $X_{i+1}$ is nonadjacent to $z$. Adopt as an inductive hypothesis that there is a transitive orientation that directs all nonedges of $G$ in $\{x_1\} \times (V \setminus X_i)$ into $x_1$ before this split. For the inductive step, note that in such a transitive orientation, any non-edge between $y \in Y$ and $x_1$ must be oriented into $x_1$, since the nonedge $(z, x_1)$ is oriented into $x_1$, and $(z, y)$ is an edge, not a nonedge, and therefore cannot be used in a transitive closure of arcs $(z, x_1)$ and $(x_1, y)$. The inductive hypothesis is therefore true for $X_{i+1}$ also. As a base case, since $X_2 = V \setminus \{x_n\}$, we may arbitrarily orient the nonedge $(x_n, x_1)$ into $x_1$, since any transitive orientation or its inverse will assign this orientation. The truth of the inductive hypothesis for $X_k = \{x_1\}$ establishes the result. $\square$

A result similar to Lemma 1 is given in [12]. However, we wish to avoid reducing the problem to *prime* co-comparability graphs, since this reduction is what makes calculation of the modular decomposition necessary. Thus, the assumption that the graph is prime is inadequate for our purposes. In order to remedy this, we now generalize it to co-comparability graphs that need not be prime.

If $\mathcal{P}$ is a modular partition of an undirected graph $G$, and $\pi$ is a Lex-BFS ordering, then for each $X \in \mathcal{P}$, let the *discovery time* of $X$ be $max\{\pi^{-1}(x) : x \in X\}$. The following result is a key element in our transitive orientation algorithm.

**Lemma 2** *Let $G$ be an arbitrary undirected graph.*

1. *If $M$ is a module of a graph $G$, then any Lex-BFS ordering of $G$ induces a Lex-BFS ordering of the subgraph $G_M$ induced by $M$.*
2. *If $\mathcal{P}$ is a modular partition of $G$, then ordering the members of $\mathcal{P}$ by their discovery times gives a Lex-BFS ordering of $G/\mathcal{P}$.*

**Proof:** For the first part, note that a pivot on a vertex $z \in V - M$ cannot affect the relative order of vertices in $M$, since $z$ is either adjacent to all members of $M$ or to none of them. To establish the relative order the Lex-BFS induces on members of $M$, operations involving vertices not in $M$ can be omitted from consideration. The subsequence of operations involving only members of $M$ are just a Lex-BFS of $G_M$.

For the second part, suppose that $\mathcal{P} = \{Y_1, Y_2, \ldots, Y_k\}$. For each set $Y_i$, let $y_i$ be the first vertex visited in $Y_i$. We analyze the operations that affect the discovery times of the members of $\mathcal{P}$, that is, the operations that affect the relative order of members of $\{y_1, y_2, \ldots, y_n\}$. No pivot on a member of $Y_i$ may

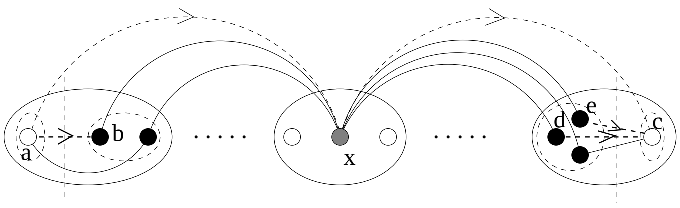


Figure 3: A pivot on a vertex $x$ in the transitive orientation algorithms. Here, the pivot splits two classes. The new interclass non edges $a \to b$, $d \to c$ and $e \to c$ are forced by $a \to x$ and $x \to c$. $x$ is the pivot here.

affect our choice of $y_j$ as the first pivot in any $Y_j$, since $Y_j$ is a module and cannot be split up by a pivot on a vertex not in $Y_j$. The pivot on $y_i$ marks the discovery time of $Y_i$, and can affect the relative order of vertices in two different classes $Y_a$ and $Y_b$. However, no subsequent pivot on a member of $Y_i$ may further affect the relative order of members of $Y_a$ and $Y_b$ since every member of $Y_i$ has the same adjacencies to them as $y_i$ does. We conclude that to establish relative discovery times, we may restrict our analysis to those operations involving members of $\{y_1, y_2, \ldots, y_k\}$. These operations are just a Lex-BFS of $G_{\{y_1,y_2,\ldots,y_k\}}$, which is isomorphic to $G/\mathcal{P}$. $\square$

**Theorem 1** *If $G$ is a co-comparability graph and $x_1$ is the last vertex visited in a Lex-BFS, then there exists a transitive orientation of $\overline{G}$ where $x_1$ is a sink.*

**Proof:** If $G$ is prime, then the result follows from Lemma 1. So assume that $G$ is not prime. Adopt as an inductive hypothesis that the lemma is true for graphs with fewer vertices than $G$.

Let $X$ be the maximal module, other than $V$, that contains $x_1$. Since $\{x_1\}$ is a module, $X$ is always defined. Let $Y$ be a maximum-cardinality module that is contained in $V \setminus X$. At least one of $X$ and $Y$ is a non-singleton set, since $G$ is not prime. Let $\mathcal{P}$ consist of $X$, $Y$, and the singleton subsets of $V \setminus (X \cup Y)$. $G/\mathcal{P}$ and $G_X$ each have fewer vertices than $G$ does.

As pivots are performed, the partition class containing $x_1$ is always first, since $x_1$ is the first vertex in the final ordering. Vertices are successively split off from the class that contains $x_1$. When only one partition class remains, it must be $X$, since $X$ cannot be split by pivots that it does not contain. Thus, $X$ is the last-discovered member of $\mathcal{P}$. By the inductive hypothesis and Lemma 2, $X$ is a sink in a transitive orientation of $\overline{G}/\mathcal{P}$ and $x_1$ is a sink in a transitive orientation of $\overline{G}_X$. The result now follows immediately from Theorem 3. $\square$

Figure 3 illustrates the forcing relation on the non edges during a Lex-BFS.

**Corollary 1** *If $G$ is a chordal co-comparability graph, and $K$ is the last clique discovered during a Lex-BFS, then there is a transitive orientation of $\overline{G}$ where every member of $K$ is a sink.*

**Proof:** $K$ consists of $x_1$ and its neighbors. Consider a transitive orientation of $\overline{G}$ where $x_1$ is a sink. For any vertex $y$ of $K$ and any non-neighbor $u$ of $y$, $u$ is not a neighbor of $x_1$, since it is not in $K$. Since $x_1$ is a sink, $uy$ is forced to be oriented toward $y$, by the orientation of $ux_1$ toward $x_1$ and the adjacency of $x_1$ and $y$. $\square$

# 4 A Transitive Orientation Algorithm

The transitive orientation algorithm of [12] uses modular decomposition to reduce the problem to that of transitively orienting prime co-comparability graphs. To transitively orient a prime co-comparability graph, they begin with an ordered partition $(V \setminus \{v\}, \{v\})$, where $v$ is a sink in a transitive orientation of $\overline{G}$. They then repeatedly perform pivots. When a class $X$ is split into $X_a$ and $X_n$ by a pivot, where $X_a$ are the vertices adjacent to the pivot, they use the following ordering rule: if the pivot vertex is in a class that follows $X$, replace $X$ in the sequence by $X_n, X_a$, in that order; otherwise replace $X$ by $X_a, X_n$.

The inductive hypothesis is that there is a transitive orientation where every of $\overline{G}$ that is not contained in a single partition class is oriented from the later partition class to the earlier one. Suppose this is true before $X$ is split. If the pivot vertex $z$ is in a later class than $X$, then all nonedges to $X_n$ are oriented toward $X_n$. This forces the orientation of all edges of $\overline{G}$ that are in $X_n \times X_a$ also to be oriented toward $X_n$, since orienting them any other way would require transitive edges from $z$ to $X_a$. Since $X_a$ is adjacent to $z$ in $G$, there can be no such transitive edge in $\overline{G}$. The inductive hypothesis thus holds after $X$ is split. It is true for the initial partition because of the choice of $v$. Since $G$ is prime, there is always a pivot that can split a non-singleton class. The final partition thus consists of singletons, and the inductive hypothesis says that the final ordering is a linear extension of a transitive orientation of $\overline{G}$.

This algorithm is not sufficient for our purposes, since we seek to eliminate the assumption that we have the modular decomposition, and thus cannot assume that we have reduced the problem to the special case where $G$ is prime. Suppose we apply their ordering rule, and perform pivots until each partition class is a module. Let $\mathcal{P}$ be the resulting modular partition. Then the inductive hypothesis given in the previous paragraph implies that the resulting ordering of $\mathcal{P}$ is a linear extension of $\overline{G}/\mathcal{P}$. By Theorem 3, it only remains to find a linear extension of a transitive orientation of $\overline{G}_X$ for each $X \in \mathcal{P}$. Our approach is to find these recursively, but as we will see, this must be done in a particular order to avoid ruining the time bound.

To obtain an $O(n + m \log n)$ time bound on prime graphs, one may use the following rule for selecting a pivot [16, 12]: only select a pivot if its current partition class is at most half the size of the partition class that contained it the last time it was used as a pivot. This guarantees that each adjacency list will be touched at most $O(\log n)$ times, which gives an $O(n + m \log n)$ bound on the running time. For the correctness, let $X$ be a largest partition class when the pivot selection rule prevents any more pivots from being selected. Every vertex $y$ not in $X$ has been used as a pivot since the last time $y$ was in a common partition class with the members of $X$; otherwise the rule would allow $y$ to be selected as a pivot. Since $X$ has not been split up by any of these pivots, it is a module. Since $G$ is prime, it must be a singleton set.

It is shown in [16] that the pivot selection rule may be extended when the graph is not prime, in order to perform pivots until every partition class is a module. If the pivot selection rule does not allow any more pivots to be selected, then we have seen that any largest class $X$ is a module. A final pivot on each member of $X$ splits any classes that are distinguished by members of $X$. $X$ can now be removed from consideration, and the algorithm may continue on the remainder of the partition and $G_{V \setminus X}$. The algorithm halts when no

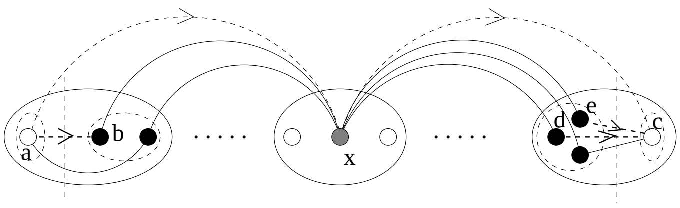


Figure 4: Transitive orientation. The new interclass non edges $a \to b$, $d \to c$ and $e \to c$ are forced by $a \to x$ and $x \to c$. $x$ is the pivot here.

vertex remains, and the removed partition classes are the desired modules. Each adjacency is used at most $\log n$ times when the pivot rule allows it to, plus an additional time, when its class is removed from consideration. Thus, the running time is still $O(n + m \log n)$.

If we apply the algorithm in [16] recursively inside the modules that it finds, using the ordering rule introduced in [12] to order the classes, then we get a linear extension of a transitive orientation of $\overline{G}$, by Theorem 3. Unfortunately, the rule that says that a final pivot on a vertex is necessary when its partition class is discovered to be a module violates the rule that a pivot is only used when its partition class is half as big as the one that contained it when it was last used. This is not a problem for the time bound when the algorithm of [16] is run once, since this situation happens only once for each vertex. When the algorithm is applied recursively, however, it can happen more than $O(\log n)$ times, so the $O(n + m \log n)$ bound fails.

We can get around this problem by changing the order of the recursive calls. When a set $X$ is discovered to be a module, Spinrad's algorithm says that we must perform a pivot on each member of $X$ before we can remove it from consideration. Instead of doing this, we observe that a pivot occurs on each member of $X$ when we make the recursive call on $X$. So, instead of performing a final pivot on each member of $X$, we make the entire recursive call on it, and only then remove it from consideration and proceed with the rest of the work in the main call. We use the pivots inside the recursive call to split also those classes not contained in $X$. This guarantees that we use a pivot in a recursive call only if the last time it was used, it was in a class that was twice as big, *even if the previous pivot occurred in the main call.* This restores the $O(m \log n)$ bound on the number of times a vertex is used as a pivot in all recursive calls put together.

To complete the algorithm, we must show how to identify a sink $v$ in each of the recursive calls. Making a call to Lex-BFS at the beginning of each recursive call would ruin the time bound. Fortunately, each recursive call is applied to a module that was discovered in a higher-level call. Thus, we may preprocess the graph by running a single call to Lex-BFS to number its vertices. By Lemma 2, part 1, whenever we need a sink in the subgraph induced by a module, we may just select the highest-numbered vertex in the module.

The complete algorithm is given as Algorithm 8, with the recursive structure of the algorithm simulated with a set of nested loops. Figure 4 gives an illustration.

A trait shared with it by our algorithm is that it fails to recognize within that time bound that the result is not transitive if the input is not a comparability

**Algorithm 8:** Transitive Orientation

**Input**: a graph $G = (V, \mathcal{E})$.

**Output**: if the input graph is a co-comparability graph, the output is an ordering of the vertices inducing a transitive orientation of the non edges.

```
begin
    compute a Lex-BFS ordering of the vertices x_1, ..., x_n
    let L be the one-element ordered list ({x_1, ..., x_n})
    lastused({x_1, ..., x_n}) = ∞
    while there exists a non-singleton class in L = (X_1, ..., X_l) do
        if there is a partition class X_a such that |X_a| ≤ lastused(X_a) then
            for each vertex x in X_a do
                for each class X_b, b ≠ a do
                    let Y be the members of X_b that are adjacent to x
                    if Y = emptyset or X_b = Y then
                        do nothing
                    else
                        remove Y from X_b
                        if b < a then insert Y immediately behind X_b
                        else insert Y immediately in front of X_b
                    lastused(Y) = lastused(X_b)
            lastused(X_a) = |X_a|
        else
            let X_c be the largest class in L
            let x_l be the last vertex in X_c discovered by the Lex-BFS (the
            vertex with the smallest number)
            replace X_c by X_c \ {x_l}, {x_l} in L
            lastused({x_l}) = ∞
end
```

graph. However, as is shown in [12], this does not prevent it from being used as a key step in many algorithms for other problems where the correctness of a solution must be certified.

The algorithm computes a linear extension of a transitive orientation of $\overline{G}$ if $G$ is a co-comparability graph. By reversing the insertion order rule for new classes, the algorithm computes a linear extension of a transitive orientation of $G$ if it is a comparability graph. A transitive orientation may then be obtained by orienting the edges according to the linear extension. Permutation graphs are those graphs that are both comparability and co-comparability graphs. Combining the two above results and using this fact, permutation graph recognition with same complexity is easily obtained; see [12] for details.

This gives the following:

**Algorithmic Result 4** *Using the algorithm 8, we can compute in $O(n+m\log n)$ time and in linear space a transitive orientation of a comparability graph.*

## 5 Interval and Co-interval Graph Recognition

We have given an algorithm for finding an ordering of vertices of $G$ that is a linear extension of a transitive orientation of $\overline{G}$ if $\overline{G}$ is a comparability graph. Given such an ordering, it is easy to check whether $G$ is an interval graph in linear time [12]. Finding the ordering takes $O(n + m \log n)$ time, yielding a simple $O(n + m \log n)$ interval graph recognition algorithm. In this section, we show how to get this bound down to $O(n + m)$ without compromising the conceptual simplicity.

Hsu and Ma [10] give a linear-time algorithm for recognizing whether a prime graph is an interval graph. They then use modular decomposition to reduce the problem to the special case of prime graphs. We show that there is a way to eliminate the modular decomposition step.

An interval graph is a chordal graph such that there exists a clique tree that is a path. That is, the maximal cliques can be linearly arranged so that all cliques containing a given vertex are consecutive. Such an ordering is called a clique chain. This associates an interval on this clique chain with each vertex, namely, the interval given by the cliques that contain the vertex. This assignment of intervals gives an interval representation of $G$, where two vertices are adjacent if and only if their intervals intersect.

There may be more than one clique chain on $G$. However, suppose that a particular transitive orientation $F$ of $\overline{G}$ is given. If $X$ and $Y$ are maximal cliques, define the relation $X <_F Y$ to hold if and only if there is some edge of $F$ that begins in $X$ and ends in $Y$. In [9], it is shown that either $X <_F Y$ or $Y <_F X$, but not both, for every pair $X, Y$ of maximal cliques, and that this relation is transitive and acyclic. Thus, $F$ defines a unique linear order on the maximal cliques. It is shown that this linear order is a clique chain.

Conversely, it is also shown in [9] that each clique chain defines a transitive orientation of $\overline{G}$. Every edge $(x, y)$ of $\overline{G}$ connects some pair $X, Y$ of maximal cliques of $G$, where $x \in X, y \notin X, y \in Y, x \notin Y$. We may say that the clique chain assigns an orientation $x \rightarrow y$ if and only if $X$ is before $Y$ in the clique chain. Thus, the problem of computing a clique chain and the problem of computing a transitive orientation of $\overline{G}$ may be regarded as dual problems.

In view of these observations, the following is an immediate consequence of Corollary 1:

**Lemma 3** *The last clique discovered in a Lex-BFS is an extreme clique in some clique chain.*

We will assume that $G$ is an interval graph and show how a clique chain can be computed under this assumption. Since only interval graphs have clique chains, the output of the algorithm must fail to be a clique chain if $G$ is not an interval graph. Checking whether the output is a clique chain will then give a recognition algorithm for interval graphs. This test can be achieved in linear time after each refinement step (line 3). For the sake of simplicity, in the interval graph recognition algorithm, the verification is made globally in a separate further step. This also can be done in linear time by the usual technique which traverses the clique chain and builds the interval representation.

Algorithm 9 gives the procedure, and Figure 5 illustrates an execution of the algorithm on an example.

**Algorithm 9:** Interval Graph Recognition (via Clique Partition Refinement)

**Input**: a graph $G = (V, \mathcal{E})$.

**Output**: if the input graph is interval: a clique chain $L$.

**begin**

1 compute the maximal cliques and a clique tree $T = (\mathcal{X}, \mathcal{F})$ using a Lex-BFS according to Algorithm 4
  let $\mathcal{X}$ be the set of maximal cliques $\mathcal{X} = \{C_1, \dots, C_k\}$
  let $L$ be the ordered list $(\mathcal{X})$
  *pivots* $= \emptyset$ is an empty stack
  **while** *there exists a non-singleton class* $\mathcal{X}_c$ *in* $L = (\mathcal{X}_1, \dots, \mathcal{X}_l)$ **do**
    **if** pivots $= \emptyset$ **then**
      let $C_l$ be the last clique in $\mathcal{X}_c$ discovered by the Lex-BFS (the clique with the greatest number)
      replace $\mathcal{X}_c$ by $\mathcal{X}_c \setminus \{C_l\}, \{C_l\}$ in $L$
      $\mathcal{C} = \{C_l\}$
    **else**
      pick an unprocessed vertex $x$ in *pivots* (throw away processed ones)
2     let $\mathcal{X}_a$ and $\mathcal{X}_b$ be the first and last classes containing a member of $\mathcal{C}$
3     replace $\mathcal{X}_a$ by $\mathcal{X}_a \setminus \mathcal{C}, \mathcal{X}_a \cap \mathcal{C}$ and $\mathcal{X}_b$ by $\mathcal{X}_b \cap \mathcal{C}, \mathcal{X}_b \setminus \mathcal{C}$
    **for each** *remaining tree edge* $(C_i, C_j)$ *connecting a clique* $C_i \in \mathcal{C}$ *to a clique* $C_j \notin \mathcal{C}$ **do**
      *pivots* $=$ *pivots* $\cup\, C_i \cap C_j$
      remove $(C_i, C_j)$ from the clique tree
  **for each** *vertex* $x$ **do**
    **if** *the cliques containing* $x$ *are not consecutive in the ordering* **then**
      **return** "$G$ is not an interval graph"
    **return** "$G$ is an interval graph"

**end**

The ordered set of partition classes maintained by the algorithm represents a partial order on the cliques, where clique $A$ is a predecessor of clique $B$ if and only if the partition class containing $A$ class precedes the one containing $B$. The invariant that we maintain is that some clique chain is a linear extension of this partial order. This can only be the case if the members of the set $\mathcal{C}$ of cliques containing a given vertex appear in consecutive partition classes, and if there is more than one such class, only the two end classes $\mathcal{X}_a$ and $\mathcal{X}_b$ of this interval can contain cliques that do not contain the vertex. If there is more than one class containing members of $\mathcal{C}$, we may split each of the $\mathcal{X}_a$ and $\mathcal{X}_b$ into cliques that contain the pivot and cliques that do not, and order the resulting classes so that the classes containing members of $\mathcal{C}$ are still consecutive. Since the cliques that contain the pivot must be consecutive in a clique chain, any clique chain that is a linear extension of the old ordered partition must also be a linear extension of the new one.

To launch the process, we put the last clique discovered during a Lex-BFS in a separate class to the right of all others. We know from Corollary 1 and Lemma 3 that there is an interval representation of $G$ where this clique is rightmost in the clique chain. This establishes the invariant initially.

Each pivot only needs to be used once, but it may not be used until the cliques that contain it reside in more than one class. The cliques containing a vertex induce a connected subtree of the clique tree, which we may refer to as its *containing subtree*. A vertex is eligible if some edge of its containing subtree intersects two partition classes. The set of vertices whose subtrees contain a tree edge $(\mathcal{C}_1, \mathcal{C}_2)$ is $\mathcal{C}_1 \cap \mathcal{C}_2$, since each vertex's containing subtree is connected. Thus, the first time $\mathcal{C}_1$ and $\mathcal{C}_2$ find themselves in different partition classes, we may add $\mathcal{C}_1 \cap \mathcal{C}_2$ to a list of eligible pivots.

Hsu and Ma show that if the graph is prime, this refinement leads to a set of partition classes where each contains one clique. The truth of the main invariant at this point gives the clique chain. However, we are not assuming that the graph is prime, since we wish to avoid the modular decomposition step. Thus, we must consider the possibility that the process will halt when some partition classes contain more than one clique. If $\mathcal{A}$ is a partition class with more than one clique at this point, let $S_A$ denote the set of vertices that occur only in cliques of $\mathcal{A}$.

If $z$ is a vertex not in $S_A$, then $z$ is either in every member of $\mathcal{A}$ or none of them; otherwise $z$ could be used to split $\mathcal{A}$ further. Thus $z$ is either adjacent to every member of $S_A$ or to none of them. We may conclude that $S_A$ is a module. In addition, since $<_F$ is a total order, for each $X, Y$ in $\mathcal{A}$, there exists $x \in X$ and $y \in Y$ such that $(x, y)$ is not an edge of $G$. It follows that $x, y \in S_A$, hence that the relative ordering of cliques in $\mathcal{A}$ may be determined by restricting our attention to $<_{F'}$, where $F'$ is a transitive orientation of $\overline{G}_{S_A}$. Since $S_A$ is a module, Theorem 3 implies that we are free to choose $F'$ to be *any* transitive orientation of $\overline{G}_{S_A}$. The existence of $x$ and $y$ also establishes that $X \cap S_A$ and $Y \cap S_A$ are not contained in the same clique of $G_{S_A}$. Since $<_{F'}$ induces a total order on $\mathcal{A}$, $\mathcal{A}' = \{K \cap S_A : K \in \mathcal{A}\}$ are the maximal cliques of $G_{S_A}$. Thus, we may call the algorithm recursively on $G_{S_A}$ to find a clique chain on $\mathcal{A}'$, and assign this ordering to the corresponding members of $\mathcal{A}$ in order to obtain the desired ordering of members of $\mathcal{A}$.

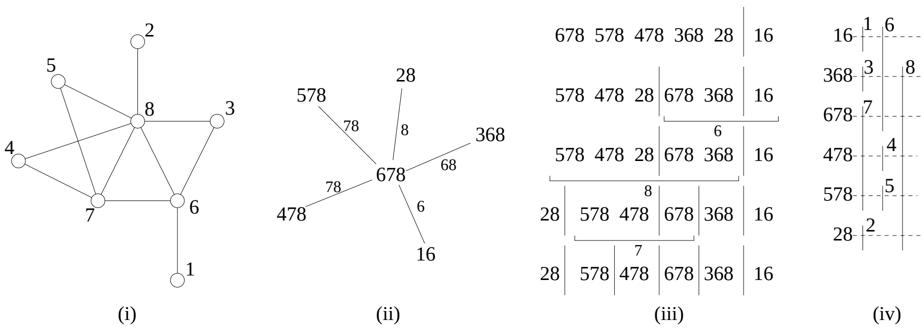


Figure 5: (i) An interval graph. Its vertices are numbered according to a Lex-BFS ordering. (ii) The clique tree associated with the Lex-BFS ordering. (iii) A partition refinement of the clique set. Note that $\{4, 5\}$ is a module. (iv) An interval representation associated with the computed clique chain.

However, naively calling the algorithm recursively in $G_{S_A}$ would result in some inefficiencies that we wish to avoid. Since $S_A$ is a module, we are able to use Lemma 2 and Lemma 3 to avoid computing another Lex-BFS ordering inside the recursive call. Instead, we reuse the ordering on $S_A$ imposed by our initial Lex-BFS. In addition, we avoid computing the members of $\mathcal{A}'$ explicitly, by letting the members of $\mathcal{A}$ stand in for them. We also simulate the recursive call within the loop structure.

For the time bound, we must consider the time bound when $G$ may or may not be chordal. In this case, the purported cliques may not actually be cliques. Because of the way the purported cliques are constructed, each purported clique is the neighborhood of its last-visited vertex, and each vertex is the last-visited vertex of at most one purported clique. Thus, there are are $O(n)$ purported cliques, and their total size is $O(n+m)$.

Each vertex is used once as a pivot, and a clique is touched once for each of its members. This gives an $O(n+m)$ bound for performing pivots and touching cliques. We must also bound the cost of maintaining the list of eligible pivots.

Since each clique has only one parent edge, the $O(n+m)$ bound on the sum of the sizes of the cliques gives an $O(n+m)$ bound on the number of times vertices are inserted in the list of pivots. To identify clique-tree edges when they first intersect two classes as a result of a pivot, we mark all tree edges incident to members of $\mathcal{C} \cap X_a$ and $\mathcal{C} \cap X_b$, since we have to touch these cliques to move them during the pivot. A tree edge that is marked only once will be deleted, so this happens $O(n)$ times. An edge that is marked twice goes between a child clique and its parent, and the child is a touched clique. In this case, we charge the cost of marking the edge to the child. Only cliques that are touched during the pivot are charged in this way, and each touched clique is charged at most once. As a consequence:

**Algorithmic Result 5** *Algorithm 9, tests in linear time and space wether a graph is an interval graph.*

For cointerval graph recognition, we note that Algorithm 6 gives a representation of the clique tree, where each clique $C$ of $\overline{G}$ is represented with the set $\overline{C} = V \setminus C$. Since $\overline{C}$ is just the neighbors in $G$ of the leftmost vertex of $C$, the sum of cardinalities of these complements of cliques is at most $m$. Thus, for each vertex, we may create a list that gives the maximal cliques of $\overline{G}$ that do not contain it. The sum of sizes of these lists is also $m$.

We obtain a cointerval graph recognition algorithm by simulating a run of Algorithm 9 on $\overline{G}$. When we pivot on a vertex, we use the lists of cliques of $\overline{G}$ that do not contain it instead of the list of cliques that do. Since the cliques that contain a vertex are consecutive in the list of partition classes on the cliques, the cliques that do not contain a vertex are contained in a prefix and/or a suffix of the list of partition classes. The end of the prefix identifies $X_a$, and the beginning of the suffix identifies $X_b$. To split $X_a$, we remove the cliques that do not contain the pivot and place them to the left of what remains of $X_a$. We perform the symmetric operation on $X_b$. This duplicates the results of the pivot had we run the original algorithm on $\overline{G}$, but in time proportional to the number of cliques that do not contain the pivot. Similarly, for the final verification step, we check for each vertex that the cliques that do not contain it in the purported clique chain form a prefix and suffix of that chain.

We must also change the way we keep track of eligible pivots. Previously, we had to detect when a tree edge $(C_1, C_2)$ first intersected more than one partition class. This happened when exactly one of $C_1$ and $C_2$ contained the pivot. We perform the equivalent test now by checking whether exactly one of $\overline{C_1}$ and $\overline{C_2}$ contains the pivot. By the charging arguments used before, we can then keep track of these events in $O(n+m)$ time. When such an event happens when we run Algorithm 9 on $\overline{G}$, we insert $C_1 \cap C_2$ into a list of eligible pivots. To simulate this exactly, we would have to insert $\overline{C_1} \cup \overline{C_2}$ into a lit of eligible pivots. This would not satisfy the time bound, since if $C_1$ has multiple children in the clique tree, the members of $\overline{C_1}$ might be inserted multiple times. However, if the members of $\overline{C_1}$ have already been inserted when an edge $(C_1, C_3)$ was processed, the list of eligible pivots will still be complete if we only insert $\overline{C_2}$. Thus, we may mark each $\overline{C}$ the first time we insert its list of members, and refrain from ever doing it again. When it is time to process $(C_1, C_2)$, we insert any or both of $\overline{C_1}$ and $\overline{C_2}$ that are unmarked, and then mark them. Since each $\overline{C}$ is only inserted once in the list of eligible pivots, keeping track of eligible pivots still takes $O(n+m)$ time.

# 6 Testing the Consecutive Ones Property

Let $M$ be a 0-1 matrix with $n$ rows and $k$ columns. $M$ is said to have *the consecutive one's property* for the rows if the columns can be permuted in a such a way that the ones in each row occur consecutively. We give a simple algorithm for testing this property in $O(r+c+m)$ time, where $r$ is the number of rows, $c$ is the number of columns, and $m$ is the number of nonzero entries in $M$.

Let us say that column $i$ of a matrix $M$ is a *subset* of column $j$ if the rows where ones occur in column $i$ are a subset of the rows where they occur in $j$. A column is *maximal* if it is a subset of no other.

The *maximal clique-vertex matrix* of a graph $G$ is a matrix where the rows are the vertices of $G$, the columns its maximal cliques, and the entry in row $i$ column $j$ is one if and only if the vertex $i$ belongs to the maximal clique $C$. A matrix is *conformal* if it is the maximum clique matrix of some graph.

Let $G(M)$ be the graph obtained from a matrix $M$ by letting each row of $M$ be a vertex, and letting two rows be adjacent in $G(M)$ if and only if they have a one in a common column. If $M$ is the maximal clique matrix of a graph $G$, then clearly $G = G(M)$. If not all columns of $M$ are maximal, then $M$ is clearly not the maximal clique matrix of any graph. If the columns of $M$ are maximal, they may still fail to be maximal cliques of $G(M)$, and $G(M)$ may have some maximal cliques that do not appear among the columns of $M$.

**Theorem 2** *For a boolean matrix $M$, the following are equivalent:*

1. *$G(M)$ is an interval graph and $M$ is its maximal clique matrix.*
2. *The columns of $M$ are maximal and $M$ has the consecutive ones property.*

**Proof:** $1 \Rightarrow 2$ : If $G(M)$ is an interval graph and $M$ is its maximal clique matrix, then the columns of $M$ are maximal, and the existence of a chain of cliques guarantees that $M$ has the consecutive ones property.

$2 \Rightarrow 1$ : The consecutive one's property is a hereditary property. Let $c_1$, $c_2$ and $c_3$ be three columns of $M$ such that $c_1, c_2, c_3$ is an appropriate ordering.

Obviously, $(c_1 \cap c_2) \cup (c_2 \cap c_3)$ is contained in $c_2$. MorEover the consecutive one's property ensures that $c_1 \cap c_3$ is included in $c_2$. Therefore for any three columns, there always exists a column that contains the union of the three pairwise intersections. In 1960, Gilmore proved that this property holds for a matrice if and only if the maximal cliques of $G(M)$ is equal to the maximal columns of $M$, see [1]. Thus, $M$ is the clique matrix of $G(M)$. Any ordering of columns of $M$ that realizes the consecutive ones property in $M$ gives a clique chain of $G(M)$. □

The approach of our algorithm is to create a matrix $\widetilde{M}$ that has all maximal columns, and that has the consecutive ones property if and only if $M$ does. If $\widetilde{M}$ has the consecutive ones property, we can use a variant of Algorithm 9 to find a clique chain on $G(\widetilde{M})$. The order of cliques in the clique chain gives a consecutive ones ordering of columns $\widetilde{M}$. This ordering of columns is a certificate that $\widetilde{M}$ has the consecutive ones property, but we must verify the certificate. If $\widetilde{M}$ does not have the consecutive ones property, the algorithm produces some ordering of the columns, but the verification step must fail, since no such certificate can exist. Thus, we only need to prove that the algorithm produces a certificate whenever $\widetilde{M}$ has the consecutive ones property, and may ignore its behavior whenever $\widetilde{M}$ does not. In the remainder of the section, we will therefore assume that $M$ and $\widetilde{M}$ have the consecutive ones property.

Unfortunately, we cannot produce an adjacency-list representation of $G(\widetilde{M})$ within the time bound. Instead, we observe that $\widetilde{M}$ is itself a representation of $G(\widetilde{M})$, since $G(\widetilde{M})$ can be constructed from it. We adapt Lex-BFS and Algorithm 9 to run directly on $\widetilde{M}$.

$\widetilde{M}$ is obtained from $M$ by appending the identity matrix below it. That is, we add $c$ dummy rows, one row for each column. Each dummy row has a one in the corresponding column and zeros in the others. Clearly $\widetilde{M}$ has the consecutive ones property if and only if $M$ does. The columns of $\widetilde{M}$ are maximal, since for each column $i$, the one in the $i^{th}$ dummy row appears only in column $i$. The size of $\widetilde{M}$ and time to construct it is clearly $O(r + c + m)$, if we use a sparse representation of $M$ and $\widetilde{M}$, where for each row we keep a list of column numbers where it has nonzero entries, and for each column, we keep a list of row numbers where it has nonzero entries.

A critical step of running Algorithm 9 on $G(\widetilde{M})$ is the call to Algorithm 4, which requires us to obtain a clique tree for $G(\widetilde{M})$. Though we do not have time to compute an adjacency-list representation of $G(\widetilde{M})$, we demonstrate how this algorithm can be adapted to produce the ordering in $O(r+c+m)$, using $\widetilde{M}$ as the representation of $G(M)$.

If $\mathcal{C}$ is a family of sets of vertices, Algorithm 10 runs in time proportional to the sum of cardinalities. If $G$ is chordal and $\mathcal{C}$ is its maximal cliques, then it gives a Lex-BFS ordering of the vertices of $G$.

**Lemma 4** *If $G$ is a chordal graph with maximal cliques $\mathcal{C}$, then Algorithm 10 computes a Lex-BFS ordering of $G$.*

**Proof:** As before, for each vertex $y$, let $label(y)$ be the numbers of the numbered neighbors of $y$ in ascending order. For each clique $C$ that has un-numbered members, let $label(C)$ be the numbers of the numbered *members* of $C$ in ascending order. Clearly, $L$ maintains the cliques in lexical order of the reverse of their current labels. Adopt as an inductive hypothesis that the first $i$ pivots are a suffix

**Algorithm 10:** Lex-BFS on a clique representation

**Input**: a family of sets $\mathcal{C}$

**Output**: if $G$ is chordal and $\mathcal{C}$ is maximal cliques, a Lex-BFS ordering of the vertices of $G$

```
begin
    let L = (C)
    i ← n
    while L is not empty do
        let C a clique in the rightmost class in L
        pick an unnumbered vertex x from C
        π(x) ← i
        if all members of C are now numbered then
            remove it from its class
        if its class is now empty then
            remove it from L
        for each class X_a in L do
            let Y be the members of X_a that contain x
            if Y is not empty and Y ≠ X_a then
                remove Y from X_a and insert to the right of X_a in L
end
```

of a valid Lex-BFS ordering, and that after the $i^{th}$ pivot, for each un-numbered vertex $y$ and clique $C$ has lexically maximal label among those cliques that contain $y$, $label(y) = label(C)$. As a base case, this is true when $i = 0$. Since the $i+1^{st}$ pivot $x$ is selected from a clique in the rightmost class of cliques this clique has lexically maximal label among all cliques with un-numbered vertices. By the inductive hypothesis, $x$ has maximal label among all un-numbered vertices. Thus, the first $i + 1$ pivots are a suffix of a valid Lex-BFS ordering. Suppose $y$ is an un-numbered vertex, after the first $i + 1$ pivots. No clique containing $y$ contains a non-neighbor of $y$, so no clique's label may be lexically greater than $y$'s. Since $G$ is chordal, $y$ and its numbered neighbors are a clique. There are one or more cliques of $G$ that contain $y$ and its numbered neighbors, so the labels of these cliques must be the same as $y$'s, and their labels must be lexically maximal among all cliques that contain $y$. The inductive hypothesis continues to hold. After all $n$ pivots, the numbering must be a valid lex-BFS numbering of $G$. □

**Lemma 5** *Algorithm 10 takes time proportional to the sum of cardinalities of members of $\mathcal{C}$.*

**Proof:** The cost of a pivot may be charged to its occurrences in members of $\mathcal{C}$. Since no vertex is used twice as a pivot, the bound is immediate. □

It remains to adapt Algorithm 4. Create a search tree $S$ whose vertices are labeled with vertices of $G$, and where each member of $\mathcal{C}$ is the sequence of labels on a path from the root to a leaf, and where these labels appear in descending Lex-BFS order. A vertex of $G$ may label more than one vertex of $S$. However, if $G$ is chordal, then for each vertex $x$, $\{x\} \cup RN(x)$ are a clique, hence $RN(x)$

appears as the labels of a path from the root to a vertex of the tree labeled with $x$. The end of this path may not be a leaf, but it must be the deepest occurrence of $x$ in the tree. Call the end of this path the *principal location* of $x$. Let $z$ be any vertex of $G$, let $s$ be its principal location in $S$, let $s'$ be the parent of $s$ in $S$, and let $y$ be the label of $s'$. Then $y$ is the parent of $z$ in the tree $T$ required for Algorithm 4, and $RN(x) - parent(x) = RN(y)$ if and only if $s'$ is $y$'s principal location. This allows all checks of Algorithm 4 to be carried out in time proportional to the sum of cardinalities of members of $\mathcal{C}$. The algorithm produces a faulty result if $G$ is not chordal, but still runs within the time bound. Summarizing, we get Algorithm 11.

**Algorithm 11:** Consecutive One's Test

**Input**: A 0-1 matrix $M$ with $c$ columns and $r$ rows and $m$ one entries
**Output**: Test whether $M$ has the consecutive one's property
**begin**
1 Compute $\widetilde{M}$ from $M$, using the sparse representation of $M$ and $\widetilde{M}$ described previously
2 Run Algorithm 9, but in the first step, call the foregoing variant of of Algorithm 4, using the columns of $\widetilde{M}$ as the maximal-clique representation of the graph.
**end**

**Algorithmic Result 6** *Algorithm 11, tests in $O(r + c + m)$ time and space whether a 0-1 matrix $M$ has the consecutive one's property.*

# 7 Some applications

A 0-1 matrix $M$ can also be seen as a bipartite graph $B = (X, Y, \mathcal{E})$ such that $X$ is the set of rows and $Y$ the set of columns. There is an edge between $x \in X$ and $y \in Y$ if the corresponding entry is one.

The maximal clique-vertex graph of a graph $G = (V, E)$ is the bipartite graph $B_c(G) = (V, \mathcal{C}(G), \mathcal{E})$. In this section, recognition algorithms for classes of bipartite graphs related to chordal graphs and intervals graphs are given, namely $Y$-semichordal graphs and convex graphs.

## Recognition of $Y$-semichordal graphs

Let $B = (X, Y, E)$ be a bipartite graph and $C = (x_1, y_1 \ldots x_k, y_k)$ a cycle of length $2k \geq 6$. $C$ has an $X$-star if there exist $x \in X$ such that $x \notin \{x_1 \ldots x_k\}$ and $i_1, i_2, i_3 \leq k$ such that $(x, y_{i_j}) \in E$ with $j \in \{1, 2, 3\}$. A $Y$-star is defined analogously.

$B$ is semichordal ($X$-semichordal, $Y$-semichordal respectively) if each cycle of length at least 6 contains an $X$-star or a $Y$-star (an $X$-star, a $Y$-star respectively). Note that the class of semichordal graphs strictly contain the union of $X$-semichordal graph and $Y$-semichordal graph. A more intuitive characterization of $Y$-semichordal graphs was presented in [3] :

**Theorem 3** *[3] A graph $G$ is chordal if and only if $B_c(G)$ is $Y$-semichordal.*

Algorithm 10 shows how to compute a a Lex-BFS ordering of the vertices if the input matrix $M$ is the maximal clique-vertex matrix of a chordal graph, in $O(r+c+m)$ time. Algorithm 3 can then test whether this ordering is a perfect elemination ordering. Thus:

**Algorithmic Result 7** *Let $B$ be a bipartite graph. It can be tested in $O(r+c+m)$ time and space whether $B$ is $Y$-semichordal (or $X$-semichordal).*

### Recognition of convex graphs

A permutation $\sigma$ of $Y$ has the *adjacency property* if for each vertex $x \in X$, its neighborhood $N(x)$ induces a factor of $\sigma$.

**Definition 1** *Let $B = (X, Y, E)$ be a bipartite graph. $B$ is a* convex graph *if there is a permutation of $X$ or $Y$ which fulfills the adjacency property.*

It is obvious that $M$ is the matrix of a convex graph with respect to $Y$ if and only if $\widetilde{M}$ is the matrix of a convex graph too. Finding a permutation of the vertex set $Y$ with the adjacency property is equivalent to finding a permutation of the columns such that for any rows the one entries occur consecutively. In other words testing whether $M$ has the consecutive one's property or testing whether $M$ is the matrix of a convex graph are the same problems.

**Algorithmic Result 8** *Algorithm 11, tests in $O(r+c+n)$ time and space whether a 0-1 matrix $M$ is the adjacency matrix of a convex graph.*

The reader should notice that up to now the only known recognition algorithm for convex graphs used PQ-trees (see [2]).

## 8 Conclusions

We have given simple algorithms and efficient algorithms for clique tree on a chordal graph or its complement, transitive orientation of a comparability graph or its complement, and interval graph recognition. ¿From the transitive orientation results follow simple algorithms for permutation graph recognition, maximum clique and minimum vertex on comparability graphs, maximum independent set and clique cover on co-comparability graphs that run in the $O(n + m \log n)$ time; see [12] for details. To date, the only general linear-time transitive orientation algorithm is quite complex [13]; the simplicity of the $O(n + m \log n)$ algorithm provides some hope for a simple linear transitive orientation algorithm that avoids modular decomposition.

The techniques might be generalized to other recognition algorithms, such as trapezoidal graphs or weakly chordal graphs and perhaps for some other interesting classes of bipartite graphs.

## References

[1] C. Berge. *Graphes et Hypergraphes.* Dunod, 1970.

[2] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using pq-tree algorithm. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

[3] A. Brandstädt. Classes of bipartite graphs related to chordal graphs. *Discrete Applied Mathematics*, 32:51–60, 1991.

[4] A. Cournier and M. Habib. A new linear algorithm of modular decomposition. In *Trees in algebra and programming—CAAP 94* (Edinburgh) Lecture Notes in Computer Science, volume 787, pages 68–84, Berlin, 1994. Springer.

[5] E. Dahlhaus, J. Gustedt, and R.M. McConnell. Efficient and practical modular decomposition. In *Proceedings of the seventh annual ACM-SIAM Symposium on Discrete Algorithm*, pages 26–35. Society of Industrial and Applied Mathematics (SIAM), 1997.

[6] P. Galinier, M. Habib, and C. Paul. Chordal graphs and their clique graph. In M. Nagl (Ed.), editor, *Graph-Theoretic Concepts in Computer Science, WG'95*, volume 1017 of *Lecture Notes in Computer Science*, pages 358–371, Aachen, Germany, June 1995. 21st Internationnal Workshop WG'95, Springer.

[7] T. Gallai. Transitiv orientierbare graphen. In *Acta Math. Acad. Sci. Hungar.*, volume 18, pages 25–66, 1967.

[8] F. Gavril. The intersection graphs of a path in a tree are exactly the chordal graphs. *Journ. Comb. Theory*, 16:47–56, 1974.

[9] M.C. Golumbic. *Algorithms Graph Theory and Perfect Graphs*. Academic Press, New York University, 1980.

[10] Hsu and Ma. Substitution decomposition on chordal graphs and applications. In *Proceedings of the 2nd ACM-SIGSAM Internationnal Symposium on Symbolic and Algebraic Computation*, number 557 in LNCS. Springer-Verlag.

[11] N. Korte and R. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. of Computing*, 18:68–81, 1989.

[12] R. M. McConnell and J. P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms* (Arlington, VA), pages 536–545, 1994.

[13] R.M. McConnell and J.P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of undirected graphs. In *Proceedings of the seventh annual ACM-SIAM Symposium on Discrete Algorithm*, pages 19–35. Society of Industrial and Applied Mathematics (SIAM), 1997.

[14] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journ. Comput.*, 16(6):973–989, 1987.

[15] Donald J. Rose, R. Endre Tarjan, and George S. Leuker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal of Computing*, 5(2):266–283, June 1976.

[16] J.P. Spinrad. Graph partitioning, 1986.