SIAM J. COMPUT.
Vol. 28, No. 3, pp. 1004–1020
© 1999 Society for Industrial and Applied Mathematics

FAST AND SIMPLE ALGORITHMS FOR RECOGNIZING CHORDAL COMPARABILITY GRAPHS AND INTERVAL GRAPHS*

WEN-LIAN HSU† AND TZE-HENG MA†

Abstract. In this paper, we present a linear-time algorithm for substitution decomposition on chordal graphs. Based on this result, we develop a linear-time algorithm for transitive orientation on chordal comparability graphs, which reduces the complexity of chordal comparability recognition from $O(n^2)$ to $O(n+m)$. We also devise a simple linear-time algorithm for interval graph recognition where no complicated data structure is involved.

Key words. chordal graph, triangulated graph, interval graph, analysis of algorithms, graph theory, substitution decomposition, modular decomposition, cycle-free poset, transitive orientation, graph partitioning, cardinality lexicographic ordering, graph recognition

AMS subject classifications. 68Q25, 68R10

PII. S0097539792224814

1. Notation.

All graphs in this paper are simple and have no self-loops. We also assume all graphs are connected. (For the purposes of this paper, the components of a disconnected graph can always be processed independently.) Let $G = (V, E)$ be a graph. An undirected edge between vertices u and v is denoted by uv. A directed edge from u to v is written as (u, v). For undirected graphs, the neighborhood of a vertex v, $N(v)$, is $\{w \in V : vw \in E\}$. For a set S of vertices, $N(S)=\cup_{v\in S}N(v) \setminus S$. The *degree* of a vertex v, *d(v)*, is the cardinality of $N(v)$. The *closed neighborhood* of vertex v, $N[v]$, is $\{v\} \cup N(v)$. Let $m = |E|$, $n = |V|$.

A *chordal graph* is a graph with no induced subgraph isomorphic to a cycle C_k, $k \geq 4$. Chordal graphs have been studied extensively. They are also called *triangulated*, *rigid-circuit*, and *perfect elimination* graphs. There are several subclasses of chordal graphs which have gained a lot of attention, e.g., interval graphs, split graphs, strongly chordal graphs, and chordal comparability graphs. The latter are the comparability graphs of *cycle-free* partial orders.

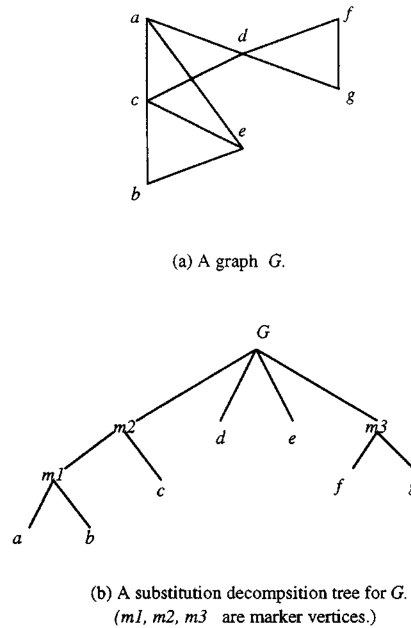
A *module* in an undirected graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that for every vertex $v \in V \setminus S$, v is adjacent to all vertices in S or no vertex in S. A module is *nontrivial* if $1 < |S| < |V|$. A *substitution decomposition* of a graph is the process of substituting a nontrivial module in the graph with a marker vertex and doing the same recursively for the module and the substituted graph. It is also called a *modular decomposition*. The process of a substitution decomposition on a graph leads to a construction of a *decomposition tree*, where each subtree represents a nontrivial module marked by its root. An example of a substitution decomposition is shown in Fig. 1.1.

For general graphs, substitution decomposition takes $O(\min(n^2, m\alpha(m, n)))$ time [15], [21]. In this paper, we call a graph *prime* if it does not contain a nontrivial module.

*Received by the editors January 17, 1992; accepted for publication February 13, 1998; published electronically January 29, 1999. A preliminary version of this paper appeared as *Substitution Decomposition on Chordal Graphs and Applications*, in ISA'91, Algorithms, Taipai, 1991, Lecture Notes in Comput. Sci. 557, Springer-Verlag, Berlin, 1991, pp. 52–60.

http://www.siam.org/journals/sicomp/28-3/22481.html

†Institute of Information Science, Academia Sinica, Nankang, Taipei, Taiwan 11529, Republic of China (hsu@iis.sinica.edu.tw, mada@iis.sinica.edu.tw).



(a) A graph G.

(b) A substitution decompsition tree for G.
($m1$, $m2$, $m3$ are marker vertices.)

FIG. 1.1.

A directed graph $G = (V, E)$ is *transitive* if for all $u, v, w \in V$, $(u, v), (v, w) \in E \Rightarrow (u, w) \in E$. For a transitive graph G, since G has no self-loops, G must be acyclic. An undirected graph is a *comparability graph* if we can give each edge a direction such that the resultant directed graph is transitive. This process is called a *transitive orientation*. The complement of a comparability graph is called a *co-comparability graph*.

A *clique* is a set completely connected vertices. A clique is *maximal* if it is not an induced subgraph of any larger clique.

2. Introduction. A linear-time algorithm for the substitution decomposition on chordal graphs is given in this paper, which results in linear-time recognition algorithms for chordal comparability graphs and interval graphs. In the next section, we present an $O(n + m)$ algorithm for substitution decomposition on chordal graphs. Our algorithm uses a special ordering to force vertices in the same module to occur consecutively in this ordering. This algorithm decomposes chordal graphs into prime components. A prime graph has the following properties: (i) if it is a comparability graph, there is a unique transitive orientation [19] (up to the reversal of the directions of all edges); (ii) if it is an interval graph, there is a unique interval representation for the graph [13] (by which we mean there is a unique linear maximal clique arrangement up to the reversal of the interval model).

A *chordal comparability graph* is a graph which is both a chordal graph and a comparability graph. The fastest algorithm [20] for recognizing a comparability graph involves two stages. First, the input graph is transitively oriented, which can be done in $O(n^2)$ time. Then we test whether this directed graph is transitive. The fastest algorithm for the latter problem takes time proportional to that of multiplying two $n \times n$ Boolean matrices, which is currently $O(n^{2.376})$ [3]. Recently, an $O(n + m)$ algorithm has been developed to test whether a directed chordal graph is transitive

[16]. This brings the complexity of recognizing chordal comparability graphs down to $O(n^2)$. The new bottleneck is the transitive orientation of chordal graphs. In section 4, we present an algorithm which transitively orients a prime chordal comparability graph in $O(n+m)$ time. Combined with previous results, this yields an $O(n+m)$ algorithm for chordal comparability graph recognition.

A graph $G = (V, E)$ is called an *interval graph* if it is the intersection graph of a set F of closed intervals on the real line [14]. In other words, there is a one-to-one mapping between the vertices in G and the intervals in F such that two vertices are adjacent iff their corresponding intervals overlap. Interval graphs are exactly the chordal co-comparability graphs [10]. This class of graphs has a wide range of applications (cf. [11]).

Booth and Lueker [1] devised the first linear-time algorithm to recognize interval graphs using a complicated data structure called a PQ-tree. Korte and Möhring [11] simplified the operations on a PQ-tree by carrying out an incremental algorithm based on a lexicographic ordering. In section 5, we present a linear-time algorithm for recognizing prime interval graphs without using a PQ-tree. Combined with the decomposition algorithm, this yields a linear-time algorithm for interval graph recognition. We consider our algorithm to be much simpler than previous ones since there is no complicated data structure involved and the approach is intuitively appealing.

3. Substitution decomposition on chordal graphs. A vertex is *simplicial* if its neighbors form a clique. A necessary and sufficient condition for a graph to be chordal is that it admits a *perfect elimination scheme*, which is a linear ordering of the vertices such that, for each vertex v, the neighbors of v that are ordered after v form a clique. A perfect elimination scheme of a chordal graph can be obtained by taking the reverse of a *lexicographic ordering*, which can be carried out in linear time [18].

A lexicographic ordering can be considered a special kind of breadth-first ordering. Imagine there is a label of n digits, initially filled with zeros, associated with each vertex. After the ith vertex in the ordering is chosen, put a "1" into the ith digit of the labels of the neighbors of the vertex. The $(i+1)$th vertex is then chosen among the unchosen vertices with the greatest label (with the first digit being the most significant digit). This ordering guarantees that if x is ordered before y and there is a vertex ordered before x which is a neighbor of one but not both of x, y, the first vertex added to the ordering with such property must be a neighbor of x. The linear-time algorithm is built upon a partitioning procedure. One implementation of a lexicographic ordering is shown in Fig. 3.1.

The output ordering π is the reverse of a *perfect elimination scheme* iff the input graph is chordal [18]. Perfect elimination schemes play a central role in most algorithms concerning chordal graphs.

A *cardinality lexicographic ordering* is just a lexicographic ordering with the vertices sorted by their degrees in descending order prior to the partitioning. This extra step ensures that when more than one vertex is eligible to be included to the ordering, the tie is broken by choosing a vertex with maximum degree. Since we can count the degrees and bucket sort the vertices of a graph in linear time, cardinality lexicographic ordering can also be done in linear time.

LEMMA 3.1. *Let S be a module in a chordal graph $G = (V, E)$. Either S is a clique or $N(S)$ is a clique.*

Proof. Suppose neither S nor $N(S)$ is a clique. There are $w, x \in S$, $y, z \in N(S)$, $wx, yz \notin E$. Since S is a module, both w and x are adjacent to y and z. Therefore, w, x, y, z form a C_4, which contradicts the assumption that G is chordal. ∎

```
procedure Lexicographic(G);
    create a list L of sets with V as the only set in L;
    /* each set in L is kept as a doubly linked list */
    for i := 1 to n do
        begin
            v := the first element of the first set in L;
            remove v;
            π(i) := v;
            split and replace each L_j ∈ L into N(v) ∩L_j and L_j \ N(v);
                /* put N(v) ∩L_j in front of L_j \ N(v) */
            discard empty sets;
        end;
end Lexicographic;
```

Fig. 3.1.

If module S is a clique, every vertex in S has the same closed neighborhood. Such a module is called a type I module. All type I modules can be located in $O(n+m)$ time by partitioning the vertices using the closed neighborhoods of all vertices. By Lemma 3.1, if there is a set with more than one vertex at the end of the partitioning process, it is a type I module.

Lemma 3.2. *If $N[u] \subset N[v]$, then $\pi^{-1}(u) > \pi^{-1}(v)$ in every cardinality lexicographic ordering.*

Proof. Initially, v is ordered before u since v has greater degree than u. Since $N[u] \subset N[v]$, no partitioning can pull u in front of v. Therefore, v will be included in an ordering before u. $\square$

We call a module type II if it is connected but not type I. After all type I modules are removed, a cardinality lexicographic ordering will put the vertices in a type II module consecutively and the neighborhood of the module will be ordered before the module.

Lemma 3.3. *Let S be a connected module in a chordal graph G with no type* I *module. If π is a cardinality lexicographic ordering on G, then*

(i) *$\pi^{-1}(v) < \pi^{-1}(u)$ $\forall v \in N(S)$, $u \in S$, and*

(ii) *all vertices in S are ordered consecutively in π.*

Proof. (i) Since S is not type I, $N(S)$ is a clique. If $v \in N(S)$ and $u \in S$, then $N[v] \supseteq S \cup N(S) \supseteq N[u]$. $N[v] \neq N[u]$, otherwise u, v form a type I module. By Lemma 3.2, $\pi^{-1}(v) < \pi^{-1}(u)$.

(ii) Suppose there exists $\pi^{-1}(x) < \pi^{-1}(y) < \pi^{-1}(z), y \notin S, x, z \in S$, and $\pi^{-1}(x)$, $\pi^{-1}(y)$ are smallest possible. Since every vertex in $N(S)$ is ordered before x, and x must be the first element of S in π, when x is selected, it has a lexicographic value caused by $N(S)$. At the same moment, y, as well as all vertices in S, are in the first set in L. While we are adding vertices in S into the ordering, y is never placed into a set in front of a set containing a vertex in S since (by part (i)) $y \notin N(S)$. Since S is connected, we have a pair of adjacent vertices $x', z' \in S$, $\pi^{-1}(x') < \pi^{-1}(y) < \pi^{-1}(z')$. However, when x' is selected, z' will be placed into a set before y. Therefore, $\pi^{-1}(z') < \pi^{-1}(y)$, a contradiction. $\square$

After getting the cardinality lexicographic ordering π, we scan the ordering from the last position. By Lemma 3.3, if there is a type II module, the vertices in the module must be in consecutive positions with all neighbors ordered before the module.

The algorithm to discover all type II modules in π tries to find the existence of such configurations. We use a "stack of stacks" to store scanned vertices. Each stack can be viewed as a candidate for a module. There are two conditions we have to enforce. First, no vertex in a stack has a neighbor in another previously created stack, since by Lemma 3.3, all neighbors of a type II module will be ordered before the module. Second, the neighborhoods of vertices in the stack should agree outside the stack. Each time a new vertex v is scanned, we try to start a new stack. If there is an edge extended from the top stack down to a lower stack, all boundaries between these two stacks must be broken. With each stack, we store the size of the stack, the common neighborhood of vertices in the stack, and the minimum $\pi^{-1}(w)$, where w is the neighbor of some but not all vertices in the stack. For a stack to be eligible to be a module, w must be included in the stack. After v is processed, if the size of the top stack is greater than 1 and the neighborhoods of vertices in the top stack agree on all the unscanned vertices, we conclude this stack forms a module.

Each time a module is reported, a minimal module is found. We then replace the top stack by a marker vertex, whose neighborhood is the CommonNeighbors of the stack. Afterwards, this marker vertex is treated the same as all other vertices. Hence, all modules will be reported in a recursive fashion. A pseudocode implementation of the algorithm is shown in Fig. 3.2. An example for the algorithm is shown in Fig. 3.3.

LEMMA 3.4. *ChordalSubstDecomp correctly finds all type* II *modules in a chordal graph in* $O(n+m)$ *time.*

Proof. Correctness: Suppose there is a type II module. By Lemma 3.3, all its vertices are in consecutive position in π; let v be the one with the greatest $\pi^{-1}(v)$ and u be the one with the smallest $\pi^{-1}(u)$ of the module. Since no vertex in this module has a neighbor after v in the ordering, after u is merged into the top stack v will still be the bottom vertex. After u is merged into this top stack, the MinDisagree of this stack is greater than or equal to $\pi^{-1}(u)$ since their neighborhoods agree on all vertices before u in π. This stack will be reported as a module.

Conversely, whenever a stack S is declared to be a module, no vertex is S has a neighbor beyond the bottom of S; otherwise, the stack would have been merged with that vertex. Moreover, since we check that the MinDisagree value is greater than or equal to the top of the stack, all vertices yet to be processed must agree on all elements of the stack. Hence the stack is in fact a module.

Time complexity: ChordalSubstDecomp steps through π and spends at constant on each vertex if we ignore the time for subroutine calls. (Since MergeTopTwoStacks can be called at most $O(n)$ times, the total cost of entering the while-loop is bounded by $O(n)$.) Each call of CreateStack(v) costs $O(|N(v)|)$ units of time. Overall, CreateStack takes $O(n+m)$ time. We assume the neighborhoods of each vertex are sorted by their indices in π (also the CommonNeighbors). The cost of MergeTopTwoStacks is proportional to the sizes of the CommonNeighbors of the top two stacks. Since the size of the CommonNeighbors of a stack is never larger than the degree of any member in the stack, we can charge this cost to the neighborhoods of the bottom vertex in the top stack and the top vertex in the second top stack. After a merge, the bottom of the top stack will never be a bottom and the top of the second top stack will never be a top. Therefore, the neighborhood of each vertex will never be charged more than twice. The total cost for MergeTopTwoStacks is then $O(n+m)$. Since all the costs are bounded by $O(n+m)$, the complexity of this algorithm is $O(n+m)$. ∎

```
ChordalSubstDecomp(G, π);
    i := 0; /* the number of stacks; an index */
    for j := n to 2 do
        begin
            v := π(j);
            CreateStack(v);
            while v has a neighbor in a lower stack do
                MergeTopTwoStacks;
            if the size of STACK(i) > 1 and STACK(i).MinDisagree ≥ j then
                report that vertices in STACK(i) form a module
        end;
end ChordalSubstDecomp;

CreateStack(v);
    i := i + 1;
    STACK(i).MinDisagree := n + 1;
        /* the value n + 1 indicates there is no disagreed neighbor */
    STACK(i).CommonNeighbors := N(v);
end CreateStack;

MergeTopTwoStacks;
    S := STACK(i).CommonNeighbors ∩ STACK(i − 1).CommonNeighbors;
    STACK(i − 1).MinDisagree := min( STACK(i).MinDisagree,
        STACK(i − 1).MinDisagree,
        (π^{-1}(v), v ∈ STACK(i).CommonNeighbors ∪
            STACK(i − 1).CommonNeighbors, v ∉ S) );
    STACK(i − 1).CommonNeighbors := S;
    i := i − 1
end MergeTopTwoStacks;
```

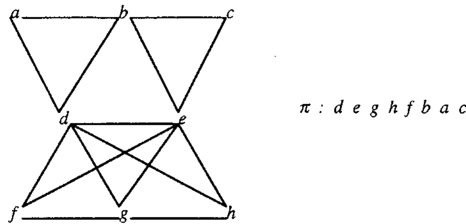
FIG. 3.2.

After we replace every type I and II module by a marker vertex, all remaining modules must be independent sets with the same neighborhoods. These modules, call them type III modules, can be found in linear time by partitioning the vertex set using their neighborhoods (as we did for finding type I modules). In conclusion, we have the following theorem.

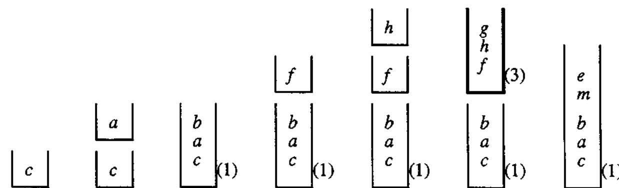
THEOREM 3.5. *The substitution decomposition on a chordal graph can be carried out in $O(n + m)$ time.*

Proof. We find type I, II, and III modules in stages. Each stage takes $O(n + m)$ time. We have to show only that at each stage, if a module of another type is generated, it will also be found.

During stage III, if a type I or II module is generated after we replace a type III module by a marker, then we must have had a connected module before the type III module is replaced by a marker. This module would have been detected in previous stages. After stage II, if a type I module S is generated, there must be some vertices in S which are markers of type II modules, otherwise S would have been detected in stage I. Suppose $v \in S$ is a marker of the last of these type II module. The module



(a) Input graph G and a cardinality lexicographic ordering.



(b) The changes of the stack configurations when π is being scanned backward. The number in the parenthesis for each nontrivial stack is its MinDisagree. Note that the stack for g,h,f is a module and is replaced by a marker vertex m.

FIG. 3.3. *An example for ChordalSubstDecomp.*

replaced by v together with $S \setminus \{v\}$ form a type II module. Therefore, by Lemma 3.4, S should be reported as a module at stage II. □

The decomposition tree can be easily constructed along with the decomposition process. Whenever a module S is reported, we replace S by a marker v with the same neighborhood as that of S. For the decomposition tree, create a tree rooted at v whose children are those vertices in S.

4. Transitive orientation on chordal graphs. In this section, we present a linear-time algorithm to transitively orient a prime chordal graph. Since a prime comparability graph admits a unique transitive orientation [19], the uniqueness provides us extra power to improve the efficiency. Together with the linear-time algorithms for substitution decomposition and transitive verification [16] on chordal graphs, we can thus recognize chordal comparability graphs in linear time.

We call $P = (X, R)$ a *partially ordered set (poset)* if X is a set and R is an irreflexive transitive binary relation on X. We say x *dominates* y if $(x, y) \in R$. If (u, v) or $(v, u) \in R$, we say that u, v are comparable. A poset $P = (X, R)$ can be viewed as a transitive graph $G = (V, E)$ by taking X as the vertex set V, R as the edge set E. Chordal comparability graphs, when transitively oriented, become a class of poset called *cycle-free* posets. For more about characteristics on cycle-free posets, see [5], [16].

A poset can be expressed by its Hasse diagram, which is an undirected graph with a minimum number of edges where there is an upward path from a to b iff a dominates b. A *chain* of a poset is a path on its diagram whose vertices are pairwise comparable. For brevity, all chains mentioned hereafter are assumed maximal unless

otherwise stated. The chains of a poset correspond to the maximal cliques of its comparability graph. Therefore, in a diagram, the vertices which appear in exactly one chain are simplicial in the poset's comparability graph.

To better understand the algorithm, the readers should keep in mind an imaginary diagram—call it the *target* diagram—which represents the poset resulting from the unique orientation of the input graph. Unlike the traditional transitive orientation algorithms, our algorithm initially does not actually orient the edges of the input graph. Instead, we explore the relative positions of vertices in the target diagram. In the end, each edge is then oriented according to the relative positions of its two endpoints.

We call a vertex w in a diagram *high* (resp., *low*) if it is dominated by (resp., dominates) a pair of incomparable vertices x, y. In a chordal comparability graph, a simplicial vertex is neither high nor low and a nonsimplicial vertex must be either high or low but not both, since if w is made high by u and v, and low by x and y, then $\{u, v, x, y\}$ induces a cycle of length 4. Thus, it can be observed that on a chain in the diagram of a cycle-free poset, the high (resp., low) vertices are above (resp., below) all simplicial and all low (resp., high) vertices. We say vertex x is higher (resp., lower) than y when there exists a chain containing both x and y where x is above (resp., below) y.

One of the most widely used characterizations of chordal graphs is that the maximal cliques of a chordal graph can be connected to form a tree T such that for each vertex v, the subgraph induced on T by the maximal cliques containing v is connected [2], [8]. (Call it the *connectivity property*.) Our algorithm applies a partitioning technique on the clique tree structure for the input chordal graph. At the end, all nonsimplicial vertices are marked either high or low and a topological sort for the target diagram is generated to provide the basis for a transitive orientation.

In our following discussion, we assume that all graphs under investigation are prime. Therefore, for any two vertices x and y, $N[x] \neq N[y]$ and $N(x) \neq N(y)$. The basic idea of our algorithm is to take a confirmed high (or low) vertex x in a certain chain and try to force the common vertices in a neighboring chain (i.e., another chain which has nonempty intersection with this one) to be low (or high). Formally, we claim the following.

LEMMA 4.1. *Let C_i and C_j be two chains with intersection S. If x is highest (resp., lowest) in C_i, $x \notin S$, then every vertex in S must be low (resp., high).*

Proof. Suppose v is the highest vertex in S. Since $x \notin C_j$, there exists $y \in C_j$, x, y are incomparable, y higher than v. Therefore, v is low and so is every vertex in S. □

To take advantage of the property of Lemma 4.1, we need to carry out our partitioning by the order where "extreme" vertices are considered first. This requirement can be easily met by the following observation.

LEMMA 4.2. *For two adjacent high (resp., low) vertices x, y, $N[x] \supset N[y]$ iff x is higher (resp., lower) than y.*

Proof. ($\Rightarrow$) Suppose $N[x] \supset N[y]$ but x is lower than y. (Note that since the graph is prime, $N[x] \neq N[y]$.) There must be a vertex z which is adjacent to x but not y. z must be higher than x since otherwise transitivity and the fact that y is higher than x would imply that y is higher than z, contradicting the fact that y and z are incomparable. y, z together with the two vertices u, v which make x high form a C_4, a contradiction.

```
procedure CliqueTree(G, π)
    create a clique C_1=π(1);
    for i := 2 to n do /* assuming the graph is connected */
        begin
            v := π(i);
            find u ∈ N(v) with maximum π^{-1} and π^{-1}(u) < i;
                /* since G is connected, u must exist. */
            let C_u be the clique generated or expanded when u is processed;
            if v and C_u form a clique, then expand C_u by adding v to it;
            else create a new clique C_j containing v and its neighbors with
                less π^{-1}, and link C_j to C_u;
        end
end CliqueTree;
```

Fig. 4.1.

($\Leftarrow$) If x is higher than y, but there exists a $z \in N[y]$ which is incomparable to x, z must be higher than y. Again, this implies the existence of a C_4. $\square$

For an input graph G, a clique tree representation can be constructed by the algorithm CliqueTree (see Fig. 4.1). We assume that the input graph is connected. The algorithm goes through a lexicographic ordering starting from the first picked vertex. Each time a new vertex v (it must be simplicial in the subgraph induced by the previously processed vertices) is processed, we either add it to an existing clique or create a new clique for v and link the new clique to an existing one.

Theorem 4.3. *CliqueTree creates a clique tree representation (i.e., links all maximal clique to form a tree T such that for each vertex v, the maximal cliques containing v induce a connected subtree) in $O(m+n)$ time.*

Proof. If we keep the record of the size of each clique, checking if v and C_u form a clique can be done in $|N(v)|$ time. Since each step can be done in $|N(v)|$ time as v is processed, CliqueTree can be done in $\mathrm{O}(m+n)$ time.

For correctness, the following three conditions must be satisfied.

(i) All maximal cliques will be generated.

This can be proved by induction. The hypothesis is that the vertices $\pi(i)\pi(i-1)...\pi(1)$ processed by CliqueTree will generate all maximal cliques among them. This is trivially true when $i = 1$. When $\pi(i+1)$ is then processed, since it is simplicial, it can be in only one maximal clique. This clique is either created or expanded from an old one.

(ii) These maximal cliques are connected as a tree.

Since each new clique (except the first one) will link to exactly one existing clique, a tree structure of all existing maximal cliques is always kept.

(iii) The tree constructed has the connectivity property.

Again, we use induction. Suppose the tree constructed on $\pi(i)\pi(i-1)...\pi(1)$ is a legitimate clique tree. If $\pi(i+1)$ is added to an existing clique, there is no problem. Suppose a new clique C_p is created by $p = \pi(i+1)$, and C_p is linked to C_q which contains a neighbor q with maximum π^{-1} (which is smaller than $i+1$) and C_q is created or expanded when q is processed. Note that all the neighbors of q with smaller π^{-1} will appear in C_q. If there exists a vertex r which is in C_p and C_r but not in C_q, since $\pi^{-1}(r) < \pi^{-1}(q)$, and q and r must be adjacent (as they are both neighbors of the simplicial vertex p), r must in C_q. Therefore, the connectivity on the clique containing any vertex is kept after p is processed. $\square$

```
procedure Orientation(G, T)
    v := a vertex of largest degree in G;
    Q := a queue containing only v;
    mark(v) := 1;
    /* a vertex u is high if it is marked 1, low if marked −1
        initially all vertices are marked 0 */
    while Q is not empty do
        begin
            remove the first vertex v from Q;
            for all tree edges e between an element of T_v and T \ T_v do
            begin
                    L := an empty list;
                    for all vertices w in e do
                        if mark(w) = 0 then
                            begin
                                mark(w) := -mark(v);
                                add w to L;
                            end;
                    sort L into order of decreasing degree;
                    append the elements of L, in order, to Q;
                    remove e from T;
            end;
        end;
end Orientation;
```

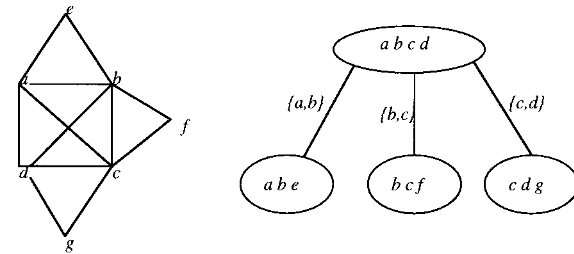
FIG. 4.2.

An example of the construction is included in Fig. 5.3. We also record the intersection of two adjacent maximal cliques on the edge connecting these maximal cliques. All these can be obtained in linear time. With this information, we are ready to perform a transitive orientation on a chordal comparability graph. The input is a chordal graph G with one of its clique tree T. Let T_v be the subtree induced by all maximal cliques containing v on T. We also assume each edge of T points to a set of vertices which are in both maximal cliques connected by that edge.

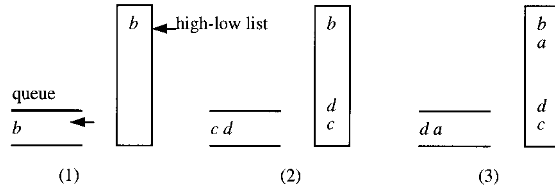
To avoid wasting time looking into an empty edge, we remove an edge on T when everything recorded in that edge is processed. This will not affect the correctness of the algorithm since our subsequent traversal is not going to pass beyond this edge anyway.

An ordering to be imposed on the vertices is constructed during our algorithm by filling in an array of length n from both ends. Whenever a vertex enters the queue, if it is marked high, it is inserted into the highest-indexed empty position in the array. Otherwise, it is inserted into the lowest-indexed empty position. Once procedure Orientation (see Fig. 4.2) has completed, we insert the remaining vertices, which are simplicial, arbitrarily into the remaining empty positions in the array. We can then orient an edge from the endpoint with a lower position to the endpoint with a higher position on the list. If the input graph is transitive, this orientation yields a partial order whose diagram is exactly characterized by the high-low relations generated by our algorithm. An example is shown in Fig. 4.3.

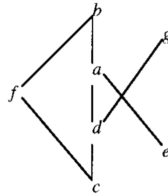
We are now ready to state the main theorem of this section.



(a) A chordal graph G and its clique tree representation. The intersections of adjacent cliques are shown in braces.



(b) The contents in the queue and the high-low list after: (1) b is selected as the initial extreme vertex and marked high; (2) b is processed (b is deleted from the braces $\{a,b\}$ and $\{b,c\}$); (3) c is processed.



(c) The target diagram constructed from the high-low list.

FIG. 4.3. *An example for the transitive orientation of a chordal graph.*

THEOREM 4.4. *A prime chordal graph, $G = (V, E)$, can be transitively oriented in $O(n + m)$ time.*

Proof. Throughout Orientation, we claim that our high-low assignments admit a feasible transitive orientation if the input graph is comparable; and the vertex x at the front of the queue must be maximal (i.e., no other vertex is higher (resp., lower) than x if x is high (resp., low)) among all unprocessed vertices. By Lemma 4.2, we know these conditions hold when we enter the while-loop. When a high vertex x is moved into the queue, all vertices higher than x are either already in the queue or they are entering the queue at the same time since, by Lemma 4.2, they exist on the same edge of the clique tree when x is detected. Since each vertex being processed is maximal among the unmarked vertices, by Lemma 4.1 and the sorting before putting vertices into the queue, we know these two conditions still hold after each operation.

The main point we need to prove is that if G is a comparability graph, every nonsimplicial vertex gets a high-low assignment when the algorithm ends. Suppose this is not true. Let x be a vertex which is high in the unique orientation after we fix the first vertex, but which is not marked high by the algorithm. Let S be the connected component containing x in the graph induced by $V \setminus M$, where M is the set of vertices marked by the algorithm. We claim that S contains more than one vertex.

Assume for a contradiction that S contains only one vertex. Then of the two vertices y, z which make x high, one (say, y) must be marked before z (y and z cannot be both simplicial, otherwise a module containing y and z can be found). There is a maximal clique containing x, y and another containing x, z, and they are connected by a path in the clique tree. When y is being processed, this path is traversed. There must be a time when y is not shared by two adjacent maximal cliques while x is. Therefore, x would have been marked opposite to y at that moment. This contradiction shows that S has at least two vertices. We claim S is a module. Suppose $u \in M$, which is not simplicial, is adjacent to v but not w, $v, w \in S$. (u must exist; otherwise $V \setminus M$ is a module.) We can find $v', w' \in S$ such that u is adjacent to v' but not w' and v', w' are adjacent. Again, there is a path in the clique tree connecting a maximal clique containing u, v' and another containing v', w'. Similar to what we have observed on x, y, z earlier, this implies that v' should have been marked opposite to u when u is used for partitioning. This contradicts $v' \in S$. Since G is prime, S cannot exist and every nonsimplicial vertex must be marked by the algorithm.

The $O(n + m)$ complexity comes from an amortized analysis [22]. The clique tree and the intersection of all pairs of adjacent maximal cliques can be obtained in linear time. When we traverse the clique tree, whenever an edge is passed, every vertex recorded in that edge will be processed (either ignored or moved to a list) in constant time and the edge of the clique tree is then deleted. Therefore, the cost of the traversal can be charged to the number of vertices stored in the edges, which is $O(n + m)$. When a set of vertices is put into the queue, we charge the cost of sorting to the edges in the clique induced by this set of vertices. Since there are quadratically many edges in a clique, we have plenty of time for sorting. After all nonsimplicial vertices are marked, the orientation can be performed in constant time per edge. The overall complexity is thus $O(n + m)$. ∎

COROLLARY 4.5. *Chordal comparability graphs can be transitively oriented in $O(n + m)$ time.*

Proof. We can incorporate the orientation algorithm with the substitution decomposition tree for a chordal graph. It is well known that the orientation within a module is independent to the orientation outside that module [9]. Formally, a graph is comparability iff each of its modules (including the prime graph represented at the root of a decomposition tree) can be transitively oriented. Since each module of a chordal graph is also chordal, our linear-time algorithm for prime chordal graph can be applied to each module when it is found. The overall time complexity is still linear since each edge is oriented once. ∎

Note that during the execution of our algorithm, we never check if an illegal orientation occurs. We only make sure that if the input graph is comparability, it will be transitively oriented correctly. With the linear-time algorithm for transitive verification for chordal graphs [16], we have the following result.

COROLLARY 4.6. *Chordal comparability graphs can be recognized in $O(m + n)$ time.*

5. Interval graph recognition.

Interval graphs have been a very useful model for many applications. Interested readers are referred to [6], [9]. The fastest algorithm to recognize interval graphs relies on the following property.

THEOREM 5.1 (See [7]). *A graph G is an interval graph iff its maximal cliques can be linearly ordered such that, for each vertex v, the maximal cliques containing v occur consecutively.*

This linear ordering of the maximal cliques actually admits a clique tree which

is a path. Unfortunately, although we can generate a clique tree for a chordal graph in linear time, the clique tree constructed for an interval graph by the algorithm is not necessarily a path. Booth and Lueker [1] created a data structure called PQ-tree to capture the consecutive property of a set of intervals. Based on this data structure, a linear-time algorithm is devised to recognize interval graphs. However, their algorithm is quite involved. Korte and Möhring [11] devised a simpler algorithm to recognize interval graphs which also runs in linear time. They observed that if the input vertices follow a lexicographic ordering, the operations on the PQ-tree can be simplified.

Hsu [13] proved that an interval graph has a unique maximal clique arrangement if the graph is prime. In this section, we provide a linear-time algorithm to find the linear maximal clique arrangement of a prime interval graph. Together with the linear-time substitution decomposition algorithm, we have yet another linear-time algorithm for interval graph recognition. Our algorithm uses only basic techniques such as graph partitioning and lexicographic ordering; no special data structure such as PQ-tree is required. We consider it to be much simpler than the previous algorithms. However, we want to remind the readers that Booth and Lueker's algorithm is much more flexible in the sense that it can test the consecutive 1's property of a Boolean matrix and can operate in an "on-line" fashion. In contrast, our algorithm deals only with interval graphs and must operate in an "off-line" fashion. The readers are encouraged to make comparisons among these algorithms.

In order to better understand our algorithm, we ask the readers to keep in mind a geometric model which is the unique linear maximal clique arrangement for the input graph. Our algorithm is based on a graph partitioning idea. Initially, we obtain the clique tree representation of the input chordal graph as we did in the last section. Partition the maximal cliques into two sets such that there is a linear maximal clique arrangement (if G is an interval graph) where all the maximal cliques in one set are at the left of the maximal cliques in the other set. We then further refine our partition based on the following observation.

LEMMA 5.2. *Let A and B be two sets of maximal cliques where A is at the left of B in a linear arrangement. Suppose vertex v is shared by $C_A, C_B, C_A \in A, C_B \in B$. If X is a set of maximal cliques at the right of A, all maximal cliques in X containing v must be at the left of those not containing v. Symmetrically, if Y is a set of maximal cliques at the left of B, all maximal cliques in Y containing v must be at the right of those not containing v.*

The proof to this lemma, which is omitted here, can be observed from the geometric model of an interval graph. The refinement process is repeatedly picking a vertex which is in two maximal cliques in different sets of a partition. We can then further partition these sets by Lemma 5.2. To start the partitioning process, we need an initial partition which admits a feasible linear maximal clique arrangement if the input graph is interval. The following lemma provides an easy way to find such a configuration.

Before we prove Lemma 5.3, we investigate some basic behaviors on a lexicographic ordering. A lexicographic ordering is a breadth-first ordering. Therefore, the traversed vertices always form a connected component. Moreover, the traversal always adds vertices of a particular maximal clique until it is completely traversed. For a lexicographic ordering π on G, we say that maximal clique C_a is traversed before C_b if every vertex in C_a is included in π before the last ordered vertex in C_b.

If we focus on interval graphs with a particular geometric model, a lexicographic

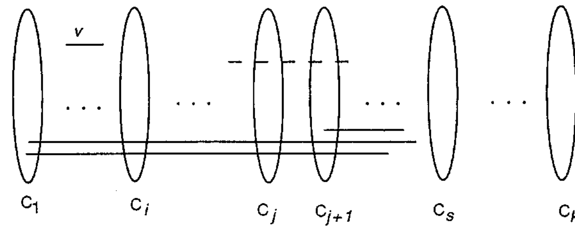


FIG. 5.1. *The scheme for proving Lemma* 5.3. *The dashed line represent an interval which cannot exist.*

ordering always picks a maximal clique to traverse first, then picks another which has the greatest lexicographic value defined by the traversed vertices, and so on. Intuitively, the traversal starts at a maximal clique and then expands toward both ends of a linear maximal clique arrangement. The maximal cliques might not be traversed consecutively as it is possible that there are more than one maximal clique with vertices which are tied in their lexicographic value.

LEMMA 5.3. *For any lexicographic ordering on a prime interval graph G, the maximal clique containing the last vertex (which is simplicial and thus contained by only one maximal clique) included in the ordering must be leftmost or rightmost on the linear maximal clique arrangement for G.*

Proof. Suppose G has k maximal cliques. Since G is prime, these maximal cliques have a unique ordering (up to reversal) $C_1, C_2, ..., C_k$, from left to right. (See Fig. 5.1.) Let C_s be the first traversed maximal clique. We claim that either C_1 or C_k will be the last traversed maximal clique. It suffices to prove that if $s > 1$, C_1 will be the last traversed maximal clique among $C_1, C_2, ..., C_s$. Suppose C_i, instead of C_1, is the last traversed maximal clique, $1 < i < s$. Keep in mind that during a lexicographic ordering, the lexicographic value of each vertex (as an interval in the linear maximal clique arrangement) is decided by its connection to the intervals that had been ordered. Let v be the first vertex which lies to the left of C_i being included in the ordering with lexicographic value α. It is clear that every unordered vertex in C_i also has the same lexicographic value. Let C_j be the maximal clique with greatest index value such that every unordered vertex in C_j also has a lexicographic value α. There is no interval connecting C_j and C_{j+1} except those that extend all the way to C_1. Otherwise, such a vertex will be ordered before v. Therefore, the set of vertices M whose interval representations end before C_{j+1} form a module. M is not trivial. Since C_1 has a simplicial vertex, and since C_i is maximal, there must be a vertex in C_i but not in C_1. This contradicts our assumption that G is prime. ☐

The clique tree structure provides enough information for us to partition a prime interval graph into a unique linear maximal clique arrangement. We now present a detailed description of the algorithm in Fig. 5.2. An example for this algorithm is presented in Fig. 5.3.

We now prove the main theorem of this section.

THEOREM 5.4. *Given a prime interval graph G, a linear maximal clique arrangement of G can be obtained in $O(m + n)$ time.*

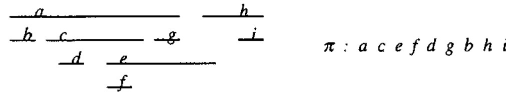
Proof. Lemmas 5.2 and 5.3 assure that LinearMaxCliqueArrange never partitions the maximal cliques incorrectly. What we need to show is that when the execution is over, a linear maximal clique arrangement can be trivially derived from the partitioning. In other words, each set of maximal cliques contains exactly one

```
procedure LinearMaxCliqueArrangement(G)
    find a clique tree of G;
    create a initial partition P which consists all maximal cliques in G;
    partition the maximal clique containing the last vertex of a lexicographic
        ordering from the rest of the maximal cliques;
    move (and delete from the clique tree) the edges in the clique tree crossing
        these two sets of maximal cliques into CrossingEdge;
    while CrossingEdge ≠ ∅ do begin
        pick an edge C_iC_j from CrossingEdge and remove it from CrossingEdge;
        for each unprocessed v ∈ C_i ∩ C_j do
            begin
                for all sets S_i in P, partition S_i into maximal cliques containing
                    v, S'_i, and maximal cliques not containing v, S''_i;
                if S'_i ≠ ∅ and S''_i ≠ ∅ then do
                    begin
                        replace S_i by S'_i, S''_i according to Lemma 5.2;
                        for each edge e in the clique tree between a member of S'_i
                            and a maximal clique not in S'_i, remove e from the
                                clique tree and add it to CrossingEdge;
                    end;
            end;
    end; {while}
end Partitioning;
```

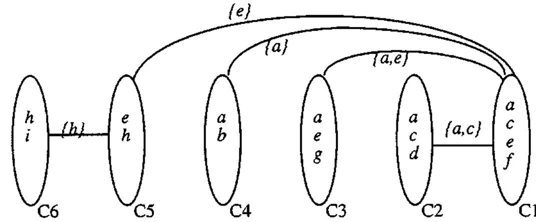
Fig. 5.2.

maximal clique. If we suppose this is not the case, then we have more than one maximal cliques in the same set S after every edge in CrossingEdge has been processed. Let V_S denote the union of vertices in the maximal cliques in S. We claim that $M = \{x | N(x) \subset V_S, x \in V_S\}$ is a module. To show this, we show that, for any vertex $u \notin M$, on the interval model, u either meets all the maximal cliques in S or none of them. If this is not true, there must exist a vertex v, $v \in C_x, C_z; v \notin C_y; C_x, C_y \in S; C_z \notin S$. From the property of a clique tree, there is a path on the subtree induced by v connecting C_x and C_z. Traversing on the path from C_x, we can find an edge on the clique tree, $C'_xC'_z$, such that $C'_x \in S$, $C'_z \notin S$. When C'_x and C'_z were first partitioned into different sets, this edge must have been put into CrossingEdge and v would have been used to partition C_x, C_y into different sets before the algorithm ends. Thus M is a module. To see that it is nontrivial, note that S must contain two distinct maximal cliques C and C', so there must be a vertex $x \in C \setminus C'$ and a vertex $y \in C' \setminus C$. Since all vertices outside M meet all or none of S, x and y must be in M, so M is a nontrivial module, which contradicts the fact that G is prime. Therefore, no set contains more than one maximal clique when the partitioning is done.

A bipartite graph H connecting the vertex set and the maximal cliques can be constructed by scanning all the maximal cliques once, such that vertex v is connected to C iff $v \in C$. The size of this graph is in $O(n+m)$. To find a clique tree and the intersections of all adjacent maximal cliques on the tree can be done in $O(n+m)$ time. The partitioning caused by vertex v can be performed in $N_H(v)$ time since all we need to do is to mark the maximal cliques containing v and split a set into



(a) The interval model of input graph G and a lexicographic ordering.



(b) The clique tree representation for G built from π.

The initial configuration:
[C6] [C5 C4 C3 C2 C1] CrossEdge = { C6C5 }

after h on C6C5 is used for partitioning...
[C6] [C5] [C4 C3 C2 C1] CrossEdge = { C5C1 }

after e on C5C1 is used for partitioning...
[C6] [C5] [C3 C1] [C4 C2] CrossEdge = { C4C1, C2C1 }

after a,c on C2C1 are used for partitioning...
[C6] [C5] [C3] [C1] [C2] [C4] CrossEdge = { C4C1, C3C1 }

FIG. 5.3. *An example for LinearMaxCliqueArrangement.*

two if necessary. To decide the left-right relation of two sets, we maintain a counter in each partitioned set recording the number of maximal cliques to the right of the set. Whenever a partition is caused by v, we examine whether v is connected to a left maximal clique (one with greater counter number) or to a right maximal clique. Since whenever an edge is moved into CrossingEdge, the edge is deleted from the clique tree, the total time needed for these movements is bounded by the number of edges in the graph. In our traversal on the maximal cliques of S'_i in order to find the crossing edges, the time we wasted on traversing noncrossing edges when a partition is caused by vertex v is bounded by $N_H(v)$. Therefore, the overall complexity is in $O(n + m)$. □

An interval model for the input graph G can be constructed from the linear maximal clique arrangement. It's easy to test in linear time whether this model is consistent with G. Therefore, we have a linear-time algorithm to recognize prime interval graphs. We can incorporate this algorithm with the substitution decomposition algorithm for chordal graphs to get a linear-time algorithm for recognizing interval graphs. Whenever a module S is identified, we test whether S is an interval graph. By Lemma 3.1, we can always replace the interval of a marker vertex v by the interval model of the module it represents and still have an interval model. If all the modules (including the last prime graph representing the input graph G) are interval graphs, G is an interval graph. We have thus proved the following corollary.

COROLLARY 5.5. *An interval graph can be recognized in linear time.*

Remark. After we submitted our paper [12], McConnell and Spinrad [17] have developed a linear-time algorithm to perform modular decomposition for general graphs. They also discussed the transitive orientation on a number of problems. Their time bound for recognizing chordal comparability graphs is $O(n + mlogn)$.

Yet another linear-time algorithm for recognizing interval graph without using PQ-tree is developed by Corneil, Olariu, and Stewart [4].

Acknowledgment. The authors thank the referees who made many helpful suggestions and a clearer presentation for procedure Orientation and the paper. We also want to thank Spinrad who pointed out an error on the clique tree construction in an earlier manuscript.

REFERENCES

[1] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.
[2] P. Buneman, *A characterization of rigid-circuit graphs*, Discrete Math., 9 (1974), pp. 205–212.
[3] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual ACM Symposium on the Theory of Computation, ACM, New York, 1987, pp. 1–6.
[4] D. Corneil, S. Olariu, and L. Stewart, *The ultimate interval graph recognition algorithm*?, in Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, Philadelphia, 1998, pp. 175–180.
[5] D. Duffus, I. Rival, and P. Winkler, *Minimizing setups for cycle-free ordered sets*, Proc. Amer. Math. Soc., 85 (1982), pp. 509–513.
[6] P. C. Fishburn, *Interval Orders and Interval Graphs*, Wiley, New York, 1985.
[7] D. R. Fulkerson and O. A. Gross, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.
[8] F. Gavril, *The intersection graphs of subtrees in trees are exactly the chordal graphs*, J. Combin. Theory B, 16 (1974), pp. 47–56.
[9] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
[10] P. C. Gilmore and J. J. Hoffman, *A characterization of comparability graphs and of interval graphs*, Canad. J. Math., 16 (1964), pp. 539–548.
[11] N. Korte and R. H. Möhring, *An incremental linear-time algorithm for recognizing interval graphs*, SIAM J. Comput., 18 (1989), pp. 68–81.
[12] W. L. Hsu and T. H. Ma, *Substitution decomposition on chordal graphs and applications*, in ISA '91, Algorithms: 2nd International Symposium on Algorithms, Lecture Notes in Comput. Sci. 557, Springer-Verlag, Berlin, 1991, pp. 52–60.
[13] W. L. Hsu, *$O(m \cdot n)$ algorithms for the recognition and isomorphism problems on circular-arc graphs*, SIAM J. Comput., 24 (1995), pp. 411–439.
[14] C. G. Lekkerkerker and J. Boland, *Representation of a finite graph by a set of intervals on the real line*, Fund. Math., 51 (1962), pp. 45–64.
[15] J. H. Muller and J. Spinrad, *Incremental modular decomposition*, J. Assoc. Comput. Mach., 36 (1989), pp. 1–19.
[16] T. H. Ma and J. Spinrad, *Cycle-free partial orders and chordal comparability graphs*, Order, 8 (1991), pp. 175–183.
[17] R. McConnell and J. Spinrad, *Linear-time modular decomposition and efficient transitive orientation of comparability graphs*, in Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, SIAM, Philadelphia, pp. 536–545.
[18] 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.
[19] L. N. Shevrin and N. D. Filippov, *Partially ordered sets and their comparability graphs*, Siberian Math. J., 11 (1970), pp. 497–509.
[20] J. Spinrad, *On comparability and permutation graphs*, SIAM J. Comput., 14 (1985), pp. 658–670.
[21] J. Spinrad, *P_4 trees and substitution decomposition*, Disc. Appl. Math., 39 (1992), pp. 263–291.
[22] R. E. Tarjan, *Amortized computational complexity*, SIAM J. Alg. Disc. Math., 6 (1985), pp. 306–318.