Fully Dynamic Algorithms for Chordal Graphs and Split Graphs

40

LOUIS IBARRA

DePaul University

Abstract. We present the first dynamic algorithm that maintains a clique tree representation of a chordal graph and supports the following operations: (1) query whether deleting or inserting an arbitrary edge preserves chordality; and (2) delete or insert an arbitrary edge, provided it preserves chordality. We give two implementations. In the first, each operation runs in $O(n)$ time, where n is the number of vertices. In the second, an insertion query runs in $O(\log^2 n)$ time, an insertion in $O(n)$ time, a deletion query in $O(n)$ time, and a deletion in $O(n \log n)$ time. We also present a data structure that allows a deletion query to run in $O(\sqrt{m})$ time in either implementation, where m is the current number of edges. Updating this data structure after a deletion or insertion requires $O(m)$ time.

We also present a very simple dynamic algorithm that supports each of the following operations in $O(1)$ time on a general graph: (1) query whether the graph is split, and (2) delete or insert an arbitrary edge.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*; E.1 [**Data**]: Data Structures—*Graphs and networks; trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Dynamic graph algorithms, chordal graphs, clique trees, split graphs

ACM Reference Format:

Ibarra, L. 2008. Fully dynamic algorithms for chordal graphs and split graphs. ACM Trans. Algor. 4, 4, Article 40 (August 2008), 20 pages. DOI = 10.1145/1383369.1383371 http://doi.acm.org/10.1145/1383369.1383371

1. *Introduction*

1.1. DYNAMIC GRAPH ALGORITHMS. A dynamic graph algorithm updates a solution to a graph problem as incremental changes are made in the input graph, for example, as single edges are deleted or inserted. The dynamic algorithm must update

Author's address: L. Ibarra, School of Computing, DePaul University, 243 S. Wabash Ave., Chicago, IL 60604, e-mail: ibarra@cs.depaul.edu.

the solution faster than a static algorithm, which computes the solution without any previously computed information. Typically, the algorithm has a preprocessing step in which it computes a solution for the initial graph, possibly with some auxiliary information, before any changes are made.

A *fully dynamic graph algorithm* supports the following operations: (1) a *query*, which is a question about the solution being maintained, such as "Are vertices u, v connected?" or "Is the graph bipartite?"; and (2) an *update*, which is the deletion or insertion of an edge. The operations must be performed *online*; that is, the algorithm must perform each operation without any information about future operations. A *deletions-only* or *insertions-only* dynamic algorithm, respectively, supports only edge deletions or only edge insertions. Dynamic algorithms for problems on weighted graphs also support updates that change an edge weight.

Fully dynamic algorithms have been developed for numerous problems on undirected graphs, including connectivity, biconnectivity, 2-edge connectivity, bipartiteness, minimum spanning trees, and planarity [Eppstein et al. 1998; Feigenbaum and Kannan 1999]. Hell et al. [2001] developed a fully dynamic algorithm that maintains a representation of a proper interval graph and allows arbitrary updates, provided the graph remains at the proper interval, and queries to decide whether an edge update is allowed.

1.2. CHORDAL GRAPHS. Let $G = (V(G), E(G)) = (V, E)$ be an undirected graph without loops or multiple edges and let $n = |V|$ and $m = |E|$. For brevity, we write $v \in G$ and $\{u, v\} \in G$ instead of $v \in V$ and $\{u, v\} \in E$, and we define $G - \{u, v\} = (V, E - \{u, v\})$ and $G + \{u, v\} = (V, E + \{u, v\})$.

Let $S \subseteq V$. The subgraph of G *induced by* S is $G[S] = (S, E[S])$, where $E[S] = \{\{u, v\} \in E \mid u, v \in S\}$. Let $G - S = G[V - S]$. A *clique* (*independent set*) of G is a set of pairwise adjacent (nonadjacent) vertices of G. The *clique number* (*independence number*) of G is the size of the largest clique (independent set) of G. The *clique cover number* (*chromatic number*) of G is the smallest k such that there is a partition of V into k cliques (independent sets). A graph G is *perfect* if, for every induced subgraph H of G, the clique number of H is equal to the chromatic number of H.

A *chord* of a path or cycle is an edge joining nonconsecutive vertices of the path or cycle. A graph G is *chordal* (or *triangulated*) if every cycle of length 4 or more has a chord. Every chordal graph is perfect [Golumbic 1980]. See Blair and Peyton [1993], Golumbic [1980], and McKee and McMorris [1999] for background. The text by Golumbic [1980] emphasizes the classical characterizations; the excellent primer by Blair and Peyton [1993] emphasizes the clique tree characterization and its applications in sparse matrix computation. Chordal graphs also have applications in biology, databases, statistics, and facility location problems [McKee and McMorris 1999].

Chordal graphs are quite tractable. There are linear-time algorithms that decide whether a graph is chordal and, if so, compute a perfect elimination ordering of its vertices [Rose et al. 1976; Tarjan and Yannakakis 1984]. There are efficient parallel algorithms to solve the same problem, such as Klein [1996]. Each of the problems CLIQUE, INDEPENDENT SET, PARTITION INTO CLIQUES, and CHROMATIC NUMBER is NP-complete on general graphs, but solvable in linear time on chordal graphs [Gavril 1972].

1.3. RESULTS. We present the first dynamic algorithm that maintains a clique tree representation of a chordal graph G and supports the following operations.

—*Delete-Query*(u, v). This returns "yes" if $G - \{u, v\}$ is chordal and "no" otherwise.

—*Delete*(u, v). This deletes $\{u, v\}$ from G, provided $G - \{u, v\}$ is chordal.

—*Insert-Query*(u, v). This returns "yes" if $G+\{u, v\}$ is chordal and "no" otherwise.

—*Insert*(u, v). This inserts $\{u, v\}$ into G, provided $G + \{u, v\}$ is chordal.

The algorithm also maintains solutions to the problems CLIQUE and CHROMATIC NUMBER. We give two implementations. In the first, each operation runs in $O(n)$ time. This implementation is very simple and uses no complex data structures. In the second, the Delete-Query operation runs in $O(n)$ time, Delete in $O(n \log n)$, Insert-Query in $O(\log^2 n)$ time, and Insert in $O(n)$ time. We also present a data structure that speeds-up Delete-Query on sparse graphs. The data structure may be used with either implementation and allows Delete-Query to run in $O(\sqrt{m})$ time. Updating the data structure after Delete or Insert requires $O(m)$ time and no updating is needed after Delete-Query or Insert-Query. All implementations improve the $O(m + n)$ time obtained by running a static algorithm (e.g., Rose et al. [1976]) for each operation. (All of the bounds in this article are worst case unless specified otherwise.)

Optionally, the algorithm can maintain the connected components of G. It supports the query "Are vertices u, v connected?" in $O(1)$ time with $O(n)$ time per update. If this query is not needed, its ET-tree data structure [Henzinger and King 1999] is easily omitted. The connected components of an arbitrary graph can be maintained with $O(1)$ time per query and $O(\sqrt{n})$ time per update, using topology trees [Frederickson 1985] combined with sparsification [Eppstein et al. 1997], but the resulting data structure is much more complicated than an ET-tree.

Our dynamic algorithm has been used to improve a result for minimal filled graphs. (Informally, a filled graph of G is G with edges added to make it chordal; see Blair et al. [2001] or Rose et al. [1976] for formal definitions. Given an arbitrary graph G and a filled graph G^+ of G, the problem is to remove fill edges from G^+ to obtain a minimal filled graph of G that is also a subgraph of G^+. The algorithm in Blair et al. [2001] solves this problem in $O(f(m + f))$ time, where $m = |E(G)|$ and $f = |E(G^+)| - m$, which is the number of fill edges in G^+. Using the first implementation of our algorithm, Heggernes and Telle have improved the running time to $O(nf + m)$ time [Heggernes 1999]. Minimal filled graphs are desirable in sparse matrix computation, as well as in other areas of computer science. Since f is $O(n)$ for many practical matrices [Blair et al. 2001], $O(nf+m)$ compares favorably with the best-known running time for computing *any* minimal filled graph of G, which is $O(nm)$ [Rose et al. 1976].

We also present a very simple dynamic algorithm that supports the following operations on a general graph: (1) query whether the graph is split, (2) delete or insert an arbitrary edge. Each operation runs in $O(1)$ time. Whenever the current graph is split, the algorithm produces solutions to CLIQUE, INDEPENDENT SET, PARTITION INTO CLIQUES, and CHROMATIC NUMBER in $O(1)$ time.

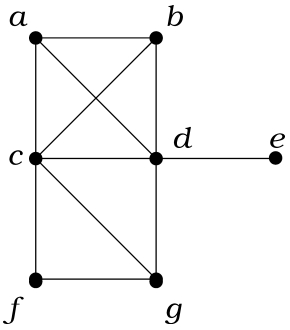

FIG. 1. A chordal graph G.

2. *Clique Trees*

Let $G = (V, E)$ be any graph. A clique K is *maximal* if K is not properly contained in another clique or, equivalently, if no vertex in $V - K$ is adjacent to every vertex in K. Let $\mathcal{K}_G$ be the set of maximal cliques of G. Let $\mathcal{K}_G(v)$ be the set of maximal cliques of G that contain $v \in V$.

A *clique tree* of G is a tree T on $\mathcal{K}_G$ with the *induced subtree property*: For any $v \in V$, the subgraph of T induced by $\mathcal{K}_G(v)$ is a tree; see Figures 1 and 2. This is equivalent to the *clique intersection property*: For any two cliques $K, K' \in \mathcal{K}_G$, the set $K \cap K'$ is contained in every clique on the path in T between K and K'. Buneman [1974], Gavril [1974], and Walter [1978] independently discovered the same characterization: G is chordal if and only if G has a clique tree.

The *weighted clique intersection graph* W_G of G is the weighted graph on $\mathcal{K}_G$, where cliques $K, K' \in \mathcal{K}_G$ are adjacent exactly when $K \cap K' \neq \emptyset$, and edge $\{K, K'\}$ is weighted by $|K \cap K'|$. Moreover, W_G is connected if and only if G is connected. For any connected graph G, a tree on $\mathcal{K}_G$ is a maximum-weight spanning tree of W_G if and only if it has the induced subtree property [Bernstein and Goodman 1981].

In summary, the following are equivalent for a tree T on $\mathcal{K}_G$.

—T is a clique tree of G.

—T has the induced subtree property.

—T has the clique intersection property.

—T is a maximum-weight spanning tree of W_G (if G is connected).

Figure 1 shows a chordal graph G and Figure 2 shows a clique tree of G. Replacing the edge $\{K_y, K_w\}$ with $\{K_x, K_w\}$ gives a different clique tree for G, which shows that a graph's clique tree may not be unique.

We use the following conventions. We refer to the *vertices* of G and the *nodes* of T and use s, t, u, v as vertex names and w, x, y, z as node names. A node $x \in T$ corresponds to maximal clique K_x of G. An edge $\{x, y\} \in T$ has weight $w(x, y) = |K_x \cap K_y|$.

Let $S \subset V$. If $u, v \in V$ are in the same connected component of G and in different connected components of $G - S$, then S is a *uv-separator* of G. A uv-separator is *minimal* if it does not properly contain another uv-separator. If $u, v \in V$ and S is a minimal uv-separator, then S is a *minimal vertex separator* of G. Ho and Lee [1989] and Lundquist [1990] independently discovered the following property of clique trees.

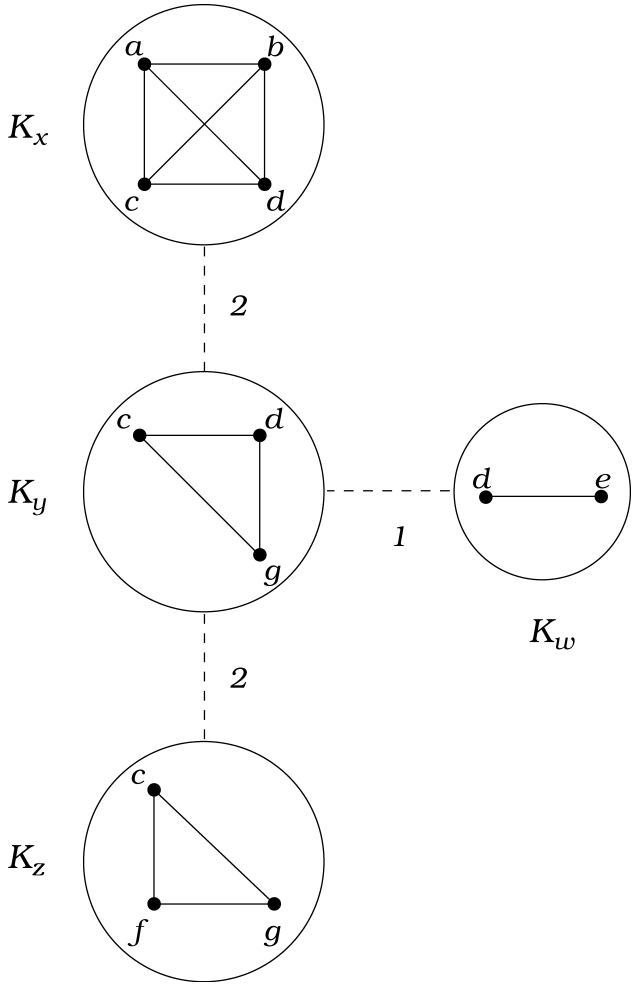

FIG. 2. A clique tree T of the graph in Figure 1, with T's edges shown by dashed lines.

THEOREM 1 ([HO AND LEE 1989; LUNDQUIST 1990]). *Let G be a connected chordal graph. Let T be any clique tree of G.*

(1) *A set S is a minimal vertex separator of G if and only if $S = K_x \cap K_y$ for some $\{x, y\} \in E(T)$.*
(2) *If $S = K_x \cap K_y$ for $\{x, y\} \in E(T)$, then S is a minimal uv-separator for any $u \in K_x - S$ and $v \in K_y - S$.*

Thus, any clique tree of G has nodes and edges that respectively correspond to the maximal cliques and minimal vertex separators of G. For example, consider the clique tree in Figure 2. If $S = K_x \cap K_y = \{c, d\}$, then S is a uv-separator for every $u \in V_x = \{a, b\}$ and $v \in V_y = \{e, f, g\}$, and S is a minimal uv-separator for every $u \in K_x - S$ and $v \in K_y - S$.

A chordal graph has at most n maximal cliques, so its clique tree has at most n nodes. There are numerous algorithms that decide whether a graph G is chordal and, if so, compute the maximal cliques of G and a clique tree T of G. Tarjan

and Yannakakis [1984] implicitly gave the first $O(m+n)$-time algorithm in their paper on acyclic hypergraphs for relational databases. Lewis et al. [1989] gave the first explicit $O(m+n)$-time algorithm in the context of sparse matrix computation. Blair and Peyton [1993] describe an $O(m+n)$-time algorithm in a graph theory context.

Most discussions of chordal graphs, such as Blair and Peyton [1993] and Lewis et al. [1989], assume a connected graph, since disconnected graphs are typically handled by applying the results to each connected component. To maintain the graph's clique tree in a unified way in our dynamic algorithm, we extend the definitions as follows. Let G be a disconnected chordal graph and let $T_1, T_2, \ldots, T_k$ be clique trees of G's connected components, $k > 1$. Let x and y be nodes of different T_i's, so that K_x and K_y are contained in different components of G. Since $K_x \cap K_y = \emptyset$ and $\{x, y\}$ is not an edge of W_G, we call $\{x, y\}$ a *dummy edge* with weight $w(x, y) = 0$. We join $T_1, T_2, \ldots, T_k$ with arbitrary dummy edges, to form a tree T which satisfies the induced subtree and clique intersection properties. We extend W_G by adding all dummy edges to it, so that T is a maximum-weight spanning tree of W_G. Thus T satisfies all three clique tree properties. We will use clique tree to refer to T and *strict clique tree* to refer to the T_i's, which are the subtrees of T corresponding to the connected components of G.

3. *The Dynamic Algorithm for Chordal Graphs*

Throughout, G denotes the current chordal graph and T denotes the current clique tree of G.

3.1. DELETE-QUERY.

We show how to decide whether $G - \{u, v\}$ is chordal for $\{u, v\} \in E$.

LEMMA 2 ([ROSE ET AL. 1976]). *Let G be a chordal graph with edge $\{u, v\}$. Then either $G - \{u, v\}$ is chordal or $G - \{u, v\}$ has a chordless cycle of length* 4.

LEMMA 3 ([GRONE ET AL. 1984]). *A graph H has no chordless cycle of length* 4 *if and only if for all distinct vertices s, t with $\{s, t\} \notin E(H)$, $H + \{s, t\}$ has exactly one maximal clique containing $\{s, t\}$.*

By Lemmas 2 and 3, $G - \{u, v\}$ is chordal if and only if for all distinct vertices s, t with $\{s, t\} \notin E(G - \{u, v\})$, $G - \{u, v\} + \{s, t\}$ has exactly one maximal clique containing $\{s, t\}$. Thus, a necessary condition for $G - \{u, v\}$ to be chordal is that G has exactly one maximal clique containing $\{u, v\}$. In fact, this is also a sufficient condition. Although sufficiency does not follow immediately from Lemma 3, its proof is very similar to Lemma 3's proof in Grone et al. [1984].

THEOREM 4. *Let G be a chordal graph with edge $\{u, v\}$. Then $G - \{u, v\}$ is chordal if and only if G has exactly one maximal clique containing $\{u, v\}$.*

PROOF. ($\Longrightarrow$) By Lemmas 2 and 3. ($\Longleftarrow$) Suppose $G - \{u, v\}$ is not chordal. By Lemma 2, $G - \{u, v\}$ has a chordless cycle of length 4, say (u, s, v, t). Since $\{u, v\}$ is an edge of G and $\{s, t\}$ is not an edge of G, it follows that $\{u, v, s\}$ and $\{u, v, t\}$ are two cliques of G that cannot be contained in the same maximal clique of G. Thus, $\{u, v\}$ is contained in at least two distinct maximal cliques of G. 2 □

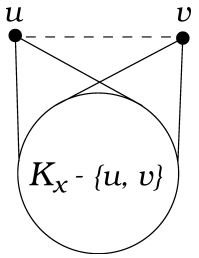

FIG. 3. Deleting $\{u, v\}$ from K_x.

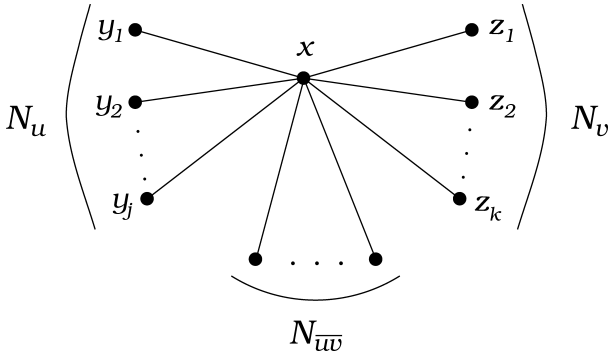

FIG. 4. Before deleting $\{u, v\}$.

Delete-Query(u, v)
For every node x of T, test whether $\{u, v\} \in K_x$. If there is exactly one such node, return "yes", and otherwise, return "no".
End Delete-Query

3.2. DELETION. We next show how to update T for $G - \{u, v\}$, given that T has a unique node x such that $\{u, v\} \in K_x$. Let $K_x^u = K_x - \{v\}$, $K_x^v = K_x - \{u\}$. In $G - \{u, v\}$, every clique K_y, $y \neq x$ is maximal but K_x has split into the cliques K_x^u, K_x^v, which may not be maximal; see Figure 3.

We decide whether K_x^u, K_x^v are maximal in $G - \{u, v\}$, as follows. Partition the set $N(x)$ of x's neighbors into

$$N_u = \{y \in N(x) \mid u \in K_y\}$$
$$N_v = \{z \in N(x) \mid v \in K_z\}$$
$$N_{\overline{uv}} = \{w \in N(x) \mid u, v \notin K_w\}$$

(see Figure 4). Let $k = |K_x|$. Since T is a clique tree, for any $y \in N(x)$, $w(x, y) \leq k - 1$.

Then

K_x^u is not maximal in $G - \{u, v\}$
iff $\exists y \in T$, $y \neq x$, $K_x^u \subset K_y$
iff $\exists y \in N_u$, $K_x^u \subset K_y$
(by the clique intersection property)
iff $\exists y \in N_u$, $K_x \cap K_y = K_x^u$
iff $\exists y \in N_u$, $w(x, y) = k - 1$.

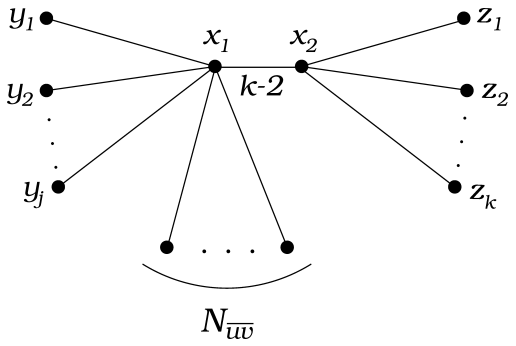

FIG. 5. Deleting $\{u, v\}$.

Similarly, K_x^v is not maximal if and only if $\exists z \in N_v$, $w(x, z) = k - 1$.

Delete(u, v)

1. Find the unique node x of T such that $\{u, v\} \in K_x$. If there is more than one such node, reject the deletion. Otherwise, for every $y \in N(x)$, test whether $u \in K_y$ or $v \in K_y$ and whether $w(x, y) = k-1$.

2. (We modify T as if K_x^u and K_x^v were maximal and then modify T again if they are not. It is straightforward to rewrite the operation so that T is modified once.)

Replace node x with new nodes x_1 and x_2 respectively representing K_x^u and K_x^v and add edge $\{x_1, x_2\}$ with $w(x_1, x_2) = k - 2$. In the following, each new edge's weight is the (replaced) old edge's weight. If $y \in N_u$, replace $\{x, y\}$ with $\{x_1, y\}$. If $z \in N_v$, replace $\{x, z\}$ with $\{x_2, z\}$. If $w \in N_{\overline{uv}}$, replace $\{x, w\}$ with $\{x_1, w\}$ or $\{x_2, w\}$, choosing arbitrarily. Observe that T has the induced subtree property. (See Figure 5.)

3. If K_x^u and K_x^v are both maximal in $G - \{u, v\}$, stop. If K_x^u is not maximal because $K_x^u \subset K_{y_i}$ for some $y_i \in N_u$, choose one such y_i arbitrarily, contract $\{x_1, y_i\}$, and replace x_1 with y_i. This maintains the induced subtree property, even if $K_x^u \subset K_{y_{i'}}$ for more than one $y_{i'} \in N_u$. Similarly, if K_x^v is not maximal because $K_x^v \subset K_{z_i}$ for some $z_i \in N_v$, choose one such z_i arbitrarily, contract $\{x_2, z_i\}$, and replace x_2 with z_i. Thus, x has been replaced with 0, 1, or 2 new nodes.

In every case, the edge added between x_1 and x_2 is not contracted and this edge's endpoints (which may have changed) contain u and v. We will subsequently use this observation.

End Delete

Delete(u, v) updates T so that there is a bijection between the nodes of T and the maximal cliques of $G - \{u, v\}$. Furthermore, T has the induced subtree property and the weight of any edge $\{y, z\} \in T$ is $|K_y \cap K_z|$. Hence T is a clique tree for $G - \{u, v\}$.

We can readily decide whether $\{u, v\}$ is a *cut edge* of G, that is, whether deleting $\{u, v\}$ disconnects a connected component of G, as follows. If $\{u, v\}$ is a cut edge, it must be a maximal clique. If $\{u, v\}$ is not a cut edge, then G has a cycle containing $\{u, v\}$ and the shortest such cycle has length 3 or else G is not chordal, which implies $\{u, v\}$ is not a maximal clique. Thus, $\{u, v\}$ is a cut edge of G if and only if $\{u, v\}$ is a maximal clique of G, that is, $|K_x| = 2$.

3.3. INSERT-QUERY. We show how to decide whether $G + \{u, v\}$ is chordal for $\{u, v\} \notin E$. We will implicitly use the fact that if T_u and T_v are the subtrees of T respectively induced by $\mathcal{K}_G(u)$ and $\mathcal{K}_G(v)$, then T_u and T_v do not intersect (because no clique contains $\{u, v\}$).

THEOREM 5. *Let G be a chordal graph without edge $\{u, v\}$. Then $G + \{u, v\}$ is chordal if and only if there exists a clique tree T of G such that $u \in K_x$, $v \in K_y$ for some $\{x, y\} \in T$.*

PROOF. ($\Longrightarrow$) Since $G' = G + \{u, v\}$ is chordal, G' has a clique tree T'. By the observation at the end of Delete(u, v), if $\{u, v\}$ is deleted from G', the algorithm produces a clique tree T of G with $u \in K_x$, $v \in K_y$ for some $\{x, y\} \in T$. ($\Longleftarrow$) If $\{x, y\}$ is a dummy edge of T, then u and v are in different connected components of G and $G + \{u, v\}$ is certainly chordal. Otherwise, u and v are in the same connected component G'', which has a strict clique tree T''. We apply Theorem 1 to G'' and T''. Let $I = K_x \cap K_y \neq \emptyset$. Since $\{u, v\}$ is not an edge, $u \notin K_y$, $v \notin K_x$ and thus $u \in K_x - I$, $v \in K_y - I$; see Figure 7. By Theorem 1, I is a uv-separator.

To show that $G + \{u, v\}$ is chordal, it suffices to show that $G'' + \{u, v\}$ is chordal. Let C be any cycle in $G'' + \{u, v\}$ with length 4 or more such that C contains $\{u, v\}$. Let $P = C - \{u, v\}$, so that P is a u–v path of length 3 or more. Since I is a uv-separator, P must contain a vertex $s \in I$. Then either $\{s, u\}$ or $\{s, v\}$ is a chord of P, which means that C has a chord. Hence $G'' + \{u, v\}$ is chordal. □

Suppose T is a clique tree of G. If T does not satisfy the condition in Theorem 5, there may exist another clique tree T' of G that does satisfy the condition. The following theorem specifies when such a clique tree T' exists.

THEOREM 6. *Let G be a chordal graph without edge $\{u, v\}$. Let T be a clique tree of G and let x, y be the closest nodes in T such that $u \in K_x$, $v \in K_y$. Assume $\{x, y\} \notin T$.*

There exists a clique tree T' of G with $u \in K_{x'}$, $v \in K_{y'}$ and $\{x', y'\} \in T'$ if and only if the minimum-weight edge e on the x–y path in T satisfies $w(e) = w(x, y)$.

PROOF. ($\Longleftarrow$) Since $T'' = T - e + \{x, y\}$ and T have the same weight, T'' is a clique tree of G. Moreover, $u \in K_x$, $v \in K_y$ and $\{x, y\} \in T''$.

($\Longrightarrow$) We first consider T. Let T_u and T_v be the subtrees of T respectively induced by $\mathcal{K}_G(u)$ and $\mathcal{K}_G(v)$. Then $x, x' \in T_u$ and $y, y' \in T_v$ (see Figure 6). By assumption, we have $\{x', y'\} \notin T$. Furthermore, x, y are on the x'–y' path P' in T, which implies that the x–y path P in T is contained in P'. By the clique intersection property, $K_{x'} \cap K_{y'} \subseteq K_x \cap K_y$ and so $w(x, y) \geq w(x', y')$. Similarly, for any edge $f \in P$,

$$w(f) \geq w(x, y) \geq w(x', y'). \tag{1}$$

We now consider T'. Let $V_{x'}$, $V_{y'}$ be the node sets of the trees of $T' - \{x', y'\}$, where $x' \in V_{x'}$ and $y' \in V_{y'}$. Consider the cut $[V_{x'}, V_{y'}]$ of the weighted clique intersection graph W_G. Since $P' + \{x', y'\}$ is a cycle of W_G containing an edge crossing $[V_{x'}, V_{y'}]$ (i.e., $\{x', y'\}$), it must contain an edge $e \in P'$ crossing $[V_{x'}, V_{y'}]$. Since $\{x', y'\} \in T'$, it follows that $e \notin T'$. Then $w(e) \leq w(x', y')$, or else $T' - \{x', y'\} + e$ is a tree with greater weight than T', a contradiction.

We claim that $e \notin P' - P$. Every edge in $P' - P$ is contained in either $V(T_u)$ or $V(T_v)$. Moreover, $x' \in T_u$ and $y' \in T_v$ implies $V(T_u) \subseteq V_{x'}$ and $V(T_v) \subseteq V_{y'}$. But e crosses $[V_{x'}, V_{y'}]$, which means $e \notin P' - P$. Therefore $e \in P$. Then, by Eq. (1), $w(e) \geq w(x, y) \geq w(x', y')$. Since $w(e) \leq w(x', y')$, it follows that $w(e) = w(x, y) = w(x', y')$. Thus e is a minimum-weight edge on P and $w(e) = w(x, y)$. □

Theorem 6 holds even if $\{u, v\}$ joins different connected components of G. In this case, $\{x, y\}$ is a dummy edge joining different strict clique trees. Then the minimum-weight edge e on the x–y path in T must also be a dummy edge and so

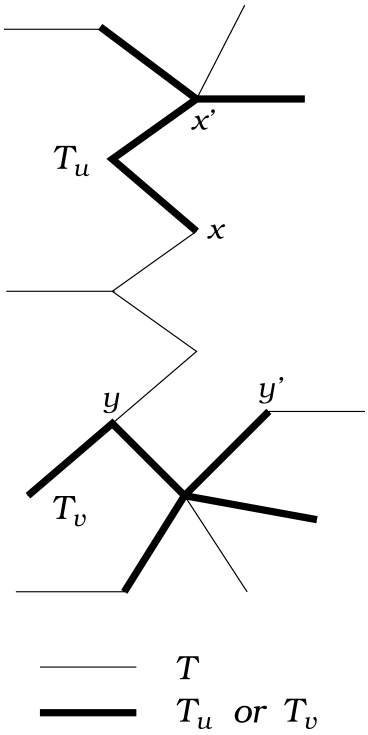

FIG. 6. Clique tree T.

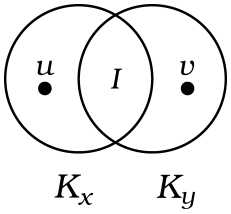

FIG. 7. Inserting $\{u, v\}$.

$w(e) = w(x, y) = 0$. Then, $T - e + \{x, y\}$ is a clique tree satisfying the condition in Theorem 5.

Insert-Query(u, v)

Find the closest nodes $x, y \in T$ such that $u \in K_x$, $v \in K_y$. If $\{x, y\} \in T$, return "yes". Otherwise, find the minimum weight edge e on the x–y path in T. If $w(e) = w(x, y)$, return "yes", and if $w(e) > w(x, y)$, return "no".

End Insert-Query

3.4. INSERTION. We next show how to update T for $G + \{u, v\}$, given that $u \in K_x$, $v \in K_y$ and $\{x, y\} \in T$. Let $I = K_x \cap K_y$. Then $K = I \cup \{u, v\}$ is a clique in $G + \{u, v\}$; see Figure 7.

We claim that K is maximal in $G + \{u, v\}$. Otherwise, there is a vertex of $V - K$ adjacent to every vertex in K, which means u and v are connected in $G - I$. However, by Theorem 1, I is a uv-separator of G, a contradiction. Thus K is a maximal clique of $G + \{u, v\}$. (Moreover, Theorem 4 implies that K is the unique

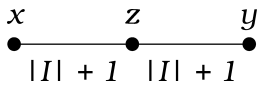

FIG. 8. Adding node z to T.

maximal clique of $G + \{u, v\}$ that contains $\{u, v\}$.) Since K is not a clique in G, we must add a new node z to T with $K_z = K$. Furthermore, if $K_w \subset K_z$ for some $w \in T$, then K_w is not maximal in $G + \{u, v\}$ and we must remove node w from T.

First, we consider whether K_w is maximal in $G + \{u, v\}$ for some $w \neq x, y$. Suppose $K_w \subset K_z$. Since K_w doesn't contain $\{u, v\}$, either $K_w \subset I \cup \{u\} \subseteq K_x$ or $K_w \subset I \cup \{v\} \subseteq K_y$, a contradiction. Thus, K_w is a maximal clique in $G + \{u, v\}$ for every $w \neq x, y$.

Second, we consider whether K_x, K_y are maximal in $G + \{u, v\}$. Since $v \notin K_x$, it follows that $K_x \subset K_z$ iff $K_x = I \cup \{u\}$ iff $|K_x| = |I| + 1$. Thus K_x is maximal in $G + \{u, v\}$ iff $|K_x| > |I| + 1$. Similarly, K_y is maximal in $G + \{u, v\}$ iff $|K_y| > |I| + 1$. Therefore, we can decide whether K_x or K_y is maximal in $G + \{u, v\}$ by comparing $|K_x|$, $|K_y|$, and $w(x, y) = |I|$.

Insert(u, v)

1. Find the closest nodes $x, y \in T$ such that $u \in K_x, v \in K_y$. If $\{x, y\} \in T$, go to Step 2. Otherwise, find the minimum-weight edge e on the x–y path in T. If $w(e) = w(x, y)$, replace e with $\{x, y\}$ in T, and if $w(e) > w(x, y)$, reject the insertion.

2. (We modify T as if K_x and K_y were maximal and then modify T again if they are not.)

Replace edge $\{x, y\}$ in T with new node z representing $K_z = I \cup \{u, v\}$ and add edges $\{x, z\}$, $\{y, z\}$, each with weight $|I| + 1$. (See Figure 8.) Determine whether K_x, K_y are maximal in $G + \{u, v\}$ by comparing $|K_x|$, $|K_y|$, and $w(x, y)$. If K_x and K_y are both maximal, stop. If K_x is not maximal, contract $\{x, z\}$ and replace x with z. If K_y is not maximal, contract $\{y, z\}$ and replace y with z. Thus, x and y have been replaced with 1, 2, or 3 nodes.

End Insert

Insert(u, v) updates T so that there is a bijection between the nodes of T and the maximal cliques of $G + \{u, v\}$. Furthermore, T has the induced subtree property and the correct edge weights. Hence, T is a clique tree for $G + \{u, v\}$.

Note that $\{u, v\}$ joins different connected components of G if and only if $\{x, y\}$ is a dummy edge of T, i.e., $w(x, y) = 0$. In this case, $K_z = \{u, v\}$.

4. *First Implementation*

In this simple implementation, we represent each maximal clique with a characteristic vector, so that membership testing requires $O(1)$ time.

The preprocessing step runs in $O(m+n)$ time, as follows. Compute the maximal cliques of G and build a clique tree T of G, which can be represented with adjacency lists. Label each edge $\{x, y\} \in T$ with its weight $w(x, y)$ and each node $x \in T$ with a pointer to the characteristic vector for K_x.

We implement Delete-Query by searching T. We implement Delete by searching T to find node x and then examining x's neighbors.

We implement Insert-Query by searching T to find x, y. If $\{x, y\} \notin T$, we compute $w(x, y) = |K_x \cap K_y|$ by comparing K_x and K_y's characteristic vectors. We implement Insert by searching T to find x, y. We compute z's characteristic vector as follows. If both of K_x, K_y are maximal, compute the AND vector of K_x and K_y's vectors and add u and v to it. If exactly one of K_x, K_y is maximal, say K_y, add v to x's vector. If neither of K_x, K_y is maximal, add v to x's vector or, equivalently, add u to y's vector.

Since T has at most n nodes and each vector has length n, Delete-Query, Delete, Insert-Query, and Insert each run in $O(n)$ time.

4.1. CONNECTED COMPONENTS. We can readily maintain the connected components of G by using a degree-d ET-tree [Henzinger and King 1999] to represent a spanning tree for each component. If $d = n^{\epsilon}, 0 < \epsilon < 1$, then each ET-tree has $O(1)$ depth and deciding whether two vertices are in the same ET-tree requires $O(1)$ time. Thus, each "Are vertices u, v connected?" query runs in $O(1)$ time. The descriptions of Delete and Insert show, respectively, how to decide whether deleting an edge disconnects a component and whether inserting an edge connects two components. Since splitting an ET-tree or joining two ET-trees requires $O(n^{\epsilon})$ time, each update runs in $O(n)$ time. All of these running times are worst case.

The only difficulty is when an edge $\{u, v\}$ to be deleted is contained in its component's spanning tree T_{sp}, but is not a cut edge of its component. Then deleting $\{u, v\}$ splits T_{sp} into the trees T_{sp}^u, T_{sp}^v, respectively containing u, v, and we must find a *replacement edge*, which is an edge of the component that connects T_{sp}^u and T_{sp}^v. Let K_x be the unique maximal clique containing $\{u, v\}$ before the deletion. Note $|K_x| > 2$ because $\{u, v\}$ is not a cut edge. Since any vertex $t \in K_x - \{u, v\}$ is adjacent to both u and v, it follows that t is in the same component as u, v and so $t \in T_{sp}$. If $t \in T_{sp}^u$, then $\{t, v\}$ is a replacement edge, and if $t \in T_{sp}^v$, then $\{t, u\}$ is a replacement edge. We join T_{sp}^u and T_{sp}^v with the replacement edge, restoring the component's spanning tree.

5. *Second Implementation*

In this implementation, we reduce the time for Insert-Query by representing the clique tree T with a Sleator-Tarjan dynamic tree [Sleator and Tarjan 1983]. Building the initial dynamic tree requires $O(n \log n)$ time. Each of the following operations requires $O(\log n)$ time on a dynamic tree with $O(n)$ nodes: rerooting the tree, finding the ith node on a path to the root, finding the minimum-weight edge on a path to the root, and splitting the tree by deleting an edge. Joining two dynamic trees with an edge requires $O(\log n)$ time, where each tree has $O(n)$ nodes.

We represent each maximal clique with a characteristic vector, as before. For every vertex $v \in V$, we maintain a pointer to any node $z \in T$ such that $v \in K_z$.

Insert-Query(u, v)

Find nodes $w, z \in T$ such that $u \in K_w, v \in K_z$. Reroot T at w. Let P be the w–z path in T and use binary search to find the lowest node $x \in P$ such that $u \in K_x$ and the highest node $y \in P$ such that $v \in K_y$. This yields the closest nodes $x, y \in T$ such that $u \in K_x, v \in K_y$.

If $\{x, y\} \in T$, return "yes". Otherwise, reroot T at x and find the minimum weight edge e on the x–y path in T. If $w(e) = w(x, y)$, return "yes"; if $w(e) > w(x, y)$, return "no".

End Insert-Query

There are a constant number of dynamic tree operations except for the binary search, which requires $O(\log n)$ "Find the ith node on a path to the root" operations. Therefore, Insert-Query requires $O(\log^2 n)$ time plus the time to compute $w(x, y) = |K_x \cap K_y|$, which we discuss in the next section.

Since Delete makes $O(n)$ changes to T, each of which is a split or join dynamic tree operation, Delete now requires $O(n \log n)$ time. The operations Delete-Query and Insert still require $O(n)$ time. Therefore, we have reduced the time for Insert-Query to $O(\log^2 n)$ at the expense of increasing the time for Delete to $O(n \log n)$.

5.1. COMPUTING THE WEIGHTS. In this section, we show how to compute $w(x, y) = |K_x \cap K_y|$ in $O(1)$ time by maintaining a $n \times n$ matrix W such that $W(x, y) = w(x, y)$. We update W in $O(n)$ time after Delete or Insert, as follows.

Deleting $\{u, v\}$ from maximal clique K_x splits it into cliques K_x^u and K_x^v, which are represented by new nodes x_1 and x_2. If K_x^u or K_x^v is not maximal, then it is contained in an existing maximal clique and we have no need to compute its corresponding W entries. Otherwise, since $K_x^u = K_x - \{v\}$ and $K_x^v = K_x - \{u\}$, for any $w \in T$, we can readily compute $W(w, x_1)$ and $W(w, x_2)$ from $W(w, x)$ in $O(1)$ time. Thus, updating W after Delete requires $O(n)$ time.

We now consider Insert. After Insert(u, v) adds node z to T and creates z's characteristic vector, we must compute $W(w, z)$ for all $w \in T$; the other entries in W are unchanged. We have $W(x, z) = W(y, z) = |I| + 1$ (see Figure 7). For any $w \neq x, y, z$, we will show how to compute $W(w, z)$ from $W(w, x)$ and $W(w, y)$.

Suppose K_x is not maximal in $G + \{u, v\}$. Then $K_x = I \cup \{u\}$ and $K_z = K_x \cup \{v\}$. Therefore, if $v \in K_w$, then $W(w, z) = W(w, x) + 1$, and otherwise $W(w, z) = W(w, x)$. We proceed similarly if K_y is not maximal in $G + \{u, v\}$.

Suppose both K_x and K_y are maximal in $G + \{u, v\}$. Consider T immediately before Insert(u, v) and let T_x, T_y be the trees of $T - \{x, y\}$, where $x \in T_x$ and $y \in T_y$. Suppose $w \in T_x$; the other case is symmetric. By the clique intersection property, $K_w \cap K_y \subseteq I$, which implies that $K_w \cap (K_y - I) = \emptyset$ (see Figures 7 and 8). Then $K_w \cap K_z = K_w \cap (I \cup \{u\}) = (K_w \cap I) \cup (K_w \cap \{u\}) = (K_w \cap K_y) \cup (K_w \cap \{u\})$. Therefore, if $u \in K_w$, then $W(w, z) = W(w, y) + 1$, and otherwise $W(w, z) = W(w, y)$. In every case, we compute $W(w, z)$ in $O(1)$ time for each w. Thus, updating W after Insert requires $O(n)$ time.

Lastly, we consider the preprocessing time to compute W for the initial chordal graph G. We can compute W in $O(n^3)$ time by comparing the characteristic vectors for every pair of maximal cliques. Alternatively, we can begin with an empty graph (with n vertices and no edges) and insert each edge of G using Insert, which requires $O(mn)$ total time. Since the dynamic algorithm maintains a clique tree, we must ensure that each partial subgraph is chordal, as follows.

A graph H is chordal if and only if H has a *perfect elimination ordering* [Fulkerson and Gross 1965], which is an ordering $v_1, v_2, \ldots, v_n$ of $V(H)$ such that for every i, the set of vertices $\{v_j \mid i < j \text{ and } \{v_i, v_j\} \in E(H)\}$ is a clique of H. We compute a perfect elimination ordering $v_1, v_2, \ldots, v_n$ of G in $O(m+n)$ time [Rose et al. 1976; Tarjan and Yannakakis 1984]. Then for each $i = 1, 2, \ldots, n$ in decreasing order, we insert the edges incident to v_i in any order. A straightforward induction proof shows that after each edge is inserted, $v_1, v_2, \ldots, v_n$ is a perfect elimination ordering of the current subgraph, which proves the subgraph is chordal. Thus we can compute W in $O(mn)$ time. This is an improvement for sparse graphs, which we consider in the next section.

6. *Faster Delete-Query*

In this section, we present a data structure to speed-up Delete-Query on sparse graphs, for example, planar graphs, which have $O(n)$ edges. The data structure may be used with either implementation and it allows Delete-Query to run in $O(\sqrt{m})$ time. Updating the data structure after Delete or Insert requires $O(m)$ time and no updating is needed after Delete-Query or Insert-Query.

6.1. INTRODUCTION. We use Yellin's algorithm for dynamic sets of elements from a totally ordered universe [Yellin 1994]. The algorithm supports the following operations, among others.

—*Create()*. This returns the index i of a new set S_i.

—*Delete(x, i)*. This deletes element x from set S_i.

—*Insert(x, i)*. This inserts element x into set S_i.

—*Intersect(i, j)*. This returns true if $S_i \cap S_j \neq \emptyset$ and false otherwise.

The intersect operation is readily extended so that it returns the size of the intersection; we use this extended operation.

For each nonisolated vertex v, we represent $\mathcal{K}_G(v)$ as a dynamic set with index v. By Theorem 4, $G-\{u, v\}$ is chordal if and only if $|\mathcal{K}_G(u) \cap \mathcal{K}_G(v)| = 1$. Therefore, we implement Delete-Query(u, v) by returning "yes" if *intersect*$(u, v) = 1$ and "no" if *intersect*$(u, v) > 1$.

Yellin gives two implementations of his algorithm. We use the second, in which each operation runs in $O(\sqrt{s_t} \log s_t)$ amortized time, where $s_t = \sum_i |S_i|$ at the time t of the operation. In our case, we have $s_t = \sum_G$, where $\sum_G = \sum_{v:d(v)>0} |\mathcal{K}_G(v)|$. For example, if G is a tree, then $\sum_G = 2(n-1)$, and if G is a clique, then $\sum_G = n$.

LEMMA 7. *If G is a chordal graph with m edges, then $\sum_G \leq 2m$.*

PROOF. If G is not connected, then applying the lemma to each connected component of G yields the lemma for G. Suppose G is connected. Let v be the first vertex in a perfect elimination ordering of G and let $k > 0$ be the degree of v. Since G is chordal, $G' = G - v$ is chordal. (Every induced subgraph of a chordal graph is chordal.) By induction, $\sum_{G'} \leq 2(m-k)$. Since v is in a unique maximal clique of G, it follows that $|\mathcal{K}_G(v)| = 1$ and for each neighbor u of v, $|\mathcal{K}_G(u)| \leq |\mathcal{K}_{G'}(u)| + 1$. Then $\sum_G \leq \sum_{G'} + k + 1 \leq 2(m-k) + k + 1 = 2m - k + 1 \leq 2m$. □

Using Yellin's algorithm, each set operation runs in $O(\sqrt{m} \log n)$ amortized time, where m is the current number of edges. Then Delete-Query runs in $O(\sqrt{m} \log n)$ amortized time. Delete or Insert may create a maximal clique with $O(n)$ vertices, say K_z, and for each $v \in K_z$, element K_z must be inserted into set $\mathcal{K}_G(v)$ using *insert*(K_z, v). Thus, updating the data structure after Delete or Insert requires $O(n\sqrt{m} \log n)$ amortized time, which is excessive.

We improve the running time by observing that our set problem is more restricted than the one solved by Yellin's algorithm: We note that a chordal graph has at most n vertices (corresponding to sets) and at most n maximal cliques (corresponding to elements). Next, we use this observation to modify Yellin's algorithm so that the intersect operation runs in $O(\sqrt{m})$ time, which means Delete-Query runs in $O(\sqrt{m})$ time. Then, we show that updating the modified data structure after Delete or Insert requires $O(m)$ time. Both bounds are worst case.

6.2. MODIFIED YELLIN'S ALGORITHM. The main difference between this algorithm and Yellin's is that we assume the universe of elements and set indices is $\{1, 2, \ldots, n\}$ and we represent the sets in the data structure with characteristic vectors, instead of balanced search trees. We describe the modified algorithm with the same notation used to present Yellin's set operations.

The main idea in Yellin's algorithm is to maintain $x.member = \{k \mid x \in S_k\}$ for each element x in some set S_i and a count $C_{i,j} = |S_i \cap S_j|$ for every pair of sets S_i, S_j. To insert x into S_i, we add i to $x.member$ and $C_{i,j}$ is incremented for each $j \in x.member$. To do this quickly, the size of $x.member$ is bounded. The modified data structure follows.

—We have a universe set U, where $x \in U$ if and only if $x \in S_i$ for some i.

—Each $x \in U$ has a set $x.member \subseteq \{k \mid x \in S_k\}$ represented by a characteristic vector and by an unsorted doubly linked list; each nonzero element in the vector has a pointer to the corresponding element in the linked list. The algorithm maintains the invariant that $|x.member| \leq \lceil \sqrt{p} \rceil$, where p is a parameter.

—Each $x \in U$ has a set $x.overflow \subset \{k \mid x \in S_k\}$ represented by a characteristic vector and by an unsorted doubly linked list; each nonzero element in the vector has a pointer to the corresponding element in the linked list. The algorithm maintains the invariant that $x.member$ and $x.overflow$ partition $\{k \mid x \in S_k\}$ and that $|x.overflow| > 0$ only if $|x.member| = \lceil \sqrt{p} \rceil$.

—We have a doubly linked list, Overflows, of the nonempty overflow sets. The invariant implies that $|\text{Overflows}| \leq s_t/\sqrt{p}$.

—Finally, we have $n \times n$ array COMMON, where $\text{COMMON}[i, j] = |\{x \mid i, j \in x.member\}| \leq |S_i \cap S_j|$. (We assume COMMON$[i, j]$ and COMMON$[j, i]$ are the same element.)

In Yellin's algorithm, each of the sets, except for Overflows, is represented by a balanced search tree and the bound on $|x.member|$ is a function close to $\sqrt{s_t}$, so that $|\text{Overflows}| = O(\sqrt{s_t})$.

The modified operations follow. To implement *intersect*(i, j), scan through Overflows and count the number of elements x such that: (1) $i \in x.member$ and $j \in x.overflow$; or (2) $i \in x.overflow$ and $j \in x.member$; or (3) $i \in x.overflow$ and $j \in x.overflow$. Return the sum of this count and COMMON$[i, j]$. Since examining each element requires $O(1)$ time, the intersect operation runs in $O(s_t/\sqrt{p})$ time.

To implement *delete*(x, j), test whether $j \in x.member$. *Case* 1: $j \in x.member$. Delete j from $x.member$ and decrement COMMON$[i, j]$ for each $i \in x.member$, which requires $O(\sqrt{p})$ time. If $x.overflow$ is nonempty, move an arbitrary set index k from $x.overflow$ to $x.member$ and increment COMMON$[i, k]$ for each $i \in x.member$. Remove $x.overflow$ from Overflows if it is empty. Remove x from U if $x.member$ is empty. *Case* 2: $j \notin x.member$. Delete j from $x.overflow$ and remove $x.overflow$ from Overflows if it is empty.

To implement *insert(x, j)*, insert x into U if it is not already present. If $|x.member| < \lceil \sqrt{m} \rceil$, insert j into $x.member$ and increment COMMON$[i, j]$ for each $i \in x.member$, which requires $O(\sqrt{p})$ time. Otherwise, insert j into $x.overflow$ and insert $x.overflow$ into Overflows if it not already present. Thus, both operations delete and insert will run in $O(\sqrt{p})$ time.

Lastly, we consider the preprocessing step. Building all member and overflow sets requires $O(s_t)$ time, where $s_t = \sum_i |S_i|$ at the initial time t. Initializing COMMON requires $O(n^2)$ time and incrementing COMMON for every pair of elements in one member set requires $O(p)$ time. Thus, the preprocessing time is $O(s_t + n^2 + pn)$.

6.3. UPDATING THE DATA STRUCTURE. We now apply our version of Yellin's algorithm with the parameter p set to the current number m of edges in G. Since $s_t \leq 2m$, each set operation runs in $O(\sqrt{m})$ time, which means Delete-Query runs in $O(\sqrt{m})$ time. It remains to show how to update the modified data structure after Delete or Insert.

The key observation is that instead of maintaining lists of the elements in each set, Yellin's algorithm maintains lists of the sets containing each element. In our problem, vertices correspond to sets and maximal cliques correspond to elements. Therefore, if Delete or Insert creates a new maximal clique, we can update the modified data structure by creating a new element (maximal clique) with its set indices (vertices) stored in its member and overflow sets.

Delete(u, v) has three cases. Recall from Section 3.1 that $u, v \in K_x$ and $K_x^u = K_x - \{v\}$, $K_x^v = K_x - \{u\}$. *Case* 1: Both of K_x^u, K_x^v are maximal, so Delete replaces K_x with K_x^u and K_x^v in the clique tree T. Create element K_x^u in U and copy its member and overflow sets from element K_x. Since any clique has at most m vertices, this requires $O(m)$ time. For every $s, t \in K_x^u.member$, increment COMMON$[s, t]$. Rename K_x to be K_x^v. Call *delete*(K_x^u, v) and *delete*(K_x^v, u) to delete set index v, u from element K_x^u, K_x^v, respectively. *Case* 2: Exactly one of K_x^u, K_x^v is maximal, say K_x^u, so Delete replaces K_x with K_x^u in T. Rename K_x to be K_x^u and call *delete*(K_x^u, v). *Case* 3: Neither of K_x^u, K_x^v is maximal, so Delete removes K_x from T. Delete K_x from U by decrementing COMMON$[s, t]$ for every $s, t \in K_x.member$ and then deleting K_x from U.

If the member bound does not hold for the new (smaller) value of m, do the following for each element $K \in U$ such that $|K.member| > \lceil\sqrt{m}\rceil$. Move an arbitrary set index t from *K.member* to *K.overflow* and for each $s \in K.member$, decrement COMMON$[s, t]$. Insert *K.overflow* into Overflows if it is not already present. Since there are at most $2\sqrt{m}$ nonempty overflow sets and every member set has size at most $\lceil\sqrt{m}\rceil$, it follows that the delete operation runs in $O(m)$ time.

Insert(u, v) also has three cases. Recall from Section 3.2 that $u \in K_x$, $v \in K_y$ and $K_z = (K_x \cap K_y) \cup \{u, v\}$. *Case* 1: Both of K_x, K_y are maximal, so Insert adds K_z to T. Create element K_z in U and initialize its member and overflow sets with $\{u, v\}$ and the vertices in $K_x \cap K_y$, computed using the characteristic vectors and linked lists of K_x and K_y. Since any clique has at most m vertices, this requires $O(m)$ time. For every $s, t \in K_z.member$, increment COMMON$[s, t]$. *Case* 2: Exactly one of K_x, K_y is maximal, say K_x, so Insert replaces K_y with $K_z = K_y \cup \{u\}$ in T. Rename K_y to be K_z and call *insert*(K_z, u). *Case* 3: Neither of K_x, K_y is maximal, so Insert replaces K_x, K_y with $K_z = K_x \cup \{v\} = K_y \cup \{u\}$ in T. Rename K_x to be K_z and call *insert*(K_z, v). Delete K_y from U by decrementing COMMON$[s, t]$ for every $s, t \in K_y.member$ and then deleting K_y from U.

If the member bound does not hold for the new (larger) value of m, restore the invariant in a similar way as in Delete. Thus, updating the modified data structure after Delete or Insert requires $O(m)$ time.

7. *Summary of Running Times*

The following table lists the running times of the four possible versions of the dynamic algorithm for chordal graphs. A_1 and A_2 represent the first and second

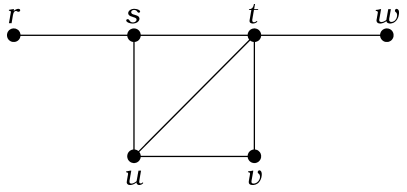

FIG. 9. A split graph.

implementations, respectively. A_1^+ and A_2^+ represent the first and second implementations with the faster Delete-Query data structure, respectively.

	Delete-Query	Delete	Insert-Query	Insert	preprocessing
A_1	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(m+n)$
A_2	$O(n)$	$O(n \log n)$	$O(\log^2 n)$	$O(n)$	$O(m+n \log n)$
A_1^+	$O(\sqrt{m})$	$O(m+n)$	$O(n)$	$O(m+n)$	$O(mn+n^2)$
A_2^+	$O(\sqrt{m})$	$O(m+n \log n)$	$O(\log^2 n)$	$O(m+n)$	$O(mn+n^2)$

8. *The Dynamic Algorithm for Split Graphs*

A graph is *split* if its vertex set can be partitioned into a clique K and an independent set I; there may be edges between vertices in K and vertices in I. Figure 9 shows a split graph. Split graphs can be recognized in linear time, and recognized in parallel in constant time [Nikolopoulos 1995]; see Golumbic [1980] and McKee and McMorris [1999] for background.

The *complement* of a graph $G = (V, E)$ is the graph $\bar{G} = (V, \bar{E})$, where $e \in \bar{E}$ if and only if $e \notin E$. A graph G is split if and only if G and $\bar{G}$ are chordal [Földes and Hammer 1977]. Therefore, we can obtain a dynamic algorithm for split graphs by using our dynamic algorithm for chordal graphs to maintain the clique trees of G and $\bar{G}$. Using the first implementation, each of the operations Delete-Query, Delete, Insert-Query, and Insert for split graphs (analogous to the operations for chordal graphs) runs in $O(n)$ time. By using an alternate characterization of split graphs, we will obtain an algorithm that supports each of the following operations on a general graph in $O(1)$ time.

—*Is-Split*. This returns "yes" if G is split and "no" otherwise.

—*Delete*(u, v). This deletes the edge $\{u, v\}$ from G.

—*Insert*(u, v). This inserts the edge $\{u, v\}$ into G.

The algorithm relies on the following characterization.

THEOREM 8 ([HAMMER AND SIMEONE 1981]). *Let G be a graph with degree sequence $d_1 \geq d_2 \geq \cdots \geq d_n$ and let $j = \max\{i \mid d_i \geq i - 1\}$. Then G is a split graph if and only if*

$$\sum_{i=1}^{j} d_i = j(j-1) + \sum_{i=j+1}^{n} d_i. \tag{2}$$

Furthermore, if Eq. (2) *holds, then G's clique number is j.*

Given a degree sequence $d_1 \geq d_2 \geq \cdots \geq d_n$, if some d_l satisfies (3), then every $d_k, k < l$, satisfies (3).

$$d_i \geq i - 1 \tag{3}$$

Given a graph G, we compute the d_i's in $O(m + n)$ time and sort them in nonincreasing order in $O(n)$ time. We build a doubly linked list $L = (d_1, d_2, \ldots, d_n)$ and an array whose ith element is a pointer to the first and last elements of L equal to i. We compute j and the sums sum_l, sum_r of the elements in $L_l = (d_1, \ldots, d_j)$, $L_r = (d_{j+1}, \ldots, d_n)$ and decide whether Eq. (2) holds. This preprocessing step requires $O(m + n)$ time. We next show how to update j, sum_l, sum_r in $O(1)$ time when an edge is inserted or deleted.

To update the data structure after an edge insertion into G, we must increment two elements of L. Consider the data structure immediately before inserting $\{u, v\}$ into G. Observe that we may choose any element of L equal to $d(u)$ ($d(v)$) as the element corresponding to u (v). If $d(u) = d(v)$, we choose the first two elements equal to $d(u)$; if $d(u) \neq d(v)$, we choose the first element equal to $d(u)$ and the first element equal to $d(v)$. Let d_{i_1}, d_{i_2} be the elements chosen to correspond to u, v, where $i_1 < i_2$. Since either $d_{i_1-1} > d_{i_1} = d_{i_2}$ or $d_{i_1-1} > d_{i_1} \geq d_{i_2-1} > d_{i_2}$, it follows that when d_{i_1}, d_{i_2} are each incremented, the nonincreasing order of L is preserved. However, some elements of L_r may now satisfy (3) and we must move them to L_l and increase j accordingly.

Consider which elements of L_r may satisfy (3) when incremented. Immediately before the insertion, $d_j \geq j-1$ and $d_{j+1} \leq j-1$. If $d_{j+1} = j-1$, then incrementing d_{j+1} will cause it to satisfy (3). Consider d_{j+k} for $k \geq 2$. Since $j-1 \geq d_{j+1} \geq d_{j+k}$, incrementing d_{j+k} cannot cause it to satisfy (3). Therefore, d_{j+1} is the only element in L_r that may satisfy (3) when incremented.

To implement Insert(u, v), we choose d_{i_1}, d_{i_2} as before (using the array of pointers) and increment each of them. If either d_{i_1} or d_{i_2} is in L_r and satisfies (3), we move it to L_l and increment j. Both of d_{i_1}, d_{i_2} cannot be in L_r and also satisfy (3), so j is incremented at most once. We then update sum_l, sum_r and decide whether Eq. (2) holds in $O(1)$ time.

To implement Delete(u, v), we proceed in the analogous way, that is, we choose d_{i_1}, d_{i_2}, decrement each, and decide whether we must decrease j. Thus, the operations Is-Split, Delete, and Insert run in $O(1)$ time.

Figure 9 shows a split graph G with degree sequence 4, 3, 3, 2, 1, 1 and $j = 3$. Then $4+3+3 = (3 \cdot 2)+2+1+1$. Inserting $\{v, w\}$ into G yields a nonsplit graph G' with degree sequence 4, 3, 3, 3, 2, 1 and $j = 4$. Inserting $\{s, v\}$ into G' yields a split graph G'' with degree sequence 4, 4, 4, 3, 2, 1 and $j = 4$. All three graphs are chordal.

Finally, since chordal graphs are perfect [Golumbic 1980], then by the last part of Theorem 8, we have G's clique number and chromatic number whenever G is a split graph. If we maintain this data structure for both G and $\bar{G}$, then we also have G's independence number and clique cover number. These numbers correspond to the problems CLIQUE, CHROMATIC NUMBER, INDEPENDENT SET, and PARTITION INTO CLIQUES, respectively.

9. *Conclusions and Open Problems*

Since a chordal graph is perfect, its clique number equals its chromatic number. Therefore, a graph's clique tree immediately provides solutions to Clique and

Chromatic Number on the graph. We might also want to maintain solutions to Independent Set and Partition into Cliques, as well as a minimum coloring, maximum independent set, and a minimum clique cover. However, the only known nontrivial algorithm for these problems on chordal graphs [Gavril 1972] requires a perfect elimination ordering of the vertices, which appears difficult to maintain under edge deletions and insertions.

Most classical algorithms for chordal graphs are based on perfect elimination orderings. In contrast, our dynamic algorithm for chordal graphs is based on clique trees. This suggests that clique trees may become as useful in dynamic algorithms for chordal graphs as they have been in sparse matrix computation.

ACKNOWLEDGMENTS. I thank V. King for support and encouragement and C. Johnson for suggesting using clique trees.

REFERENCES

BERNSTEIN, P. A., AND GOODMAN, N. 1981. Power of natural semijoins. *SIAM J. Comput. 10*, 751–771.

BLAIR, J. R. S., HEGGERNES, P., AND TELLE, J. A. 2001. A practical algorithm for making filled graphs minimal. *Theor. Comput. Sci. 250*, 1-2, 125–141.

BLAIR, J. R. S., AND PEYTON, B. 1993. An introduction to chordal graphs and clique trees. In *Graph Theory and Sparse Matrix Computation*. IMA Volumes in Mathematics and its Applications, vol. 56. Springer, New York, 1–29.

BUNEMAN, P. 1974. A characterization of rigid circuit graphs. *Discr. Math. 9*, 205–212.

EPPSTEIN, D., GALIL, Z., AND ITALIANO, G. F. 1998. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook*, M. J. Atallah, ed. CRC Press, Boca Raton, FL, Chapter 8.

EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND NISSENZWEIG, A. 1997. Sparsification—A technique for speeding up dynamic graph algorithms. *J. ACM 44*, 5, 669–696.

FEIGENBAUM, J., AND KANNAN, S. 1999. Dynamic graph algorithms. In *Handbook of Discrete and Combinatorial Mathematics*, K. H. Rosen, ed. CRC Press, Boca Raton, FL, Chapter 17.

FÖLDES, S., AND HAMMER, P. L. 1977. Split graphs. *Congr. Numer. 19*, 311–315.

FREDERICKSON, G. N. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput. 14*, 4, 781–798.

FULKERSON, D., AND GROSS, O. 1965. Incidence matrices and interval graphs. *Pacific J. Math. 15*, 3, 835–855.

GAVRIL, F. 1972. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput. 1*, 2, 180–187.

GAVRIL, F. 1974. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combinatorial Theory B 16*, 47–56.

GOLUMBIC, M. C. 1980. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York.

GRONE, R., JOHNSON, C. R., SÁ, E. M., AND WOLKOWICZ, H. 1984. Positive definite completions of partial Hermitian matrices. *Linear Algeb. Appl. 58*, 109–124.

HAMMER, P. L., AND SIMEONE, B. 1981. The splittance of a graph. *Combinatorica 1*, 275–284.

HEGGERNES, P. 1999. Personal communication, May.

HELL, P., SHAMIR, R., AND SHARAN, R. 2001. A fully dynamic algorithm for recognizing and representing proper interval graphs. *SIAM J. Comput. 31*, 289–305.

HENZINGER, M. R., AND KING, V. 1999. Randomized dynamic graph algorithms with polylogarithmic time per operation. *J. ACM 46*, 4, 502–516.

HO, C., AND LEE, R. C. T. 1989. Counting clique trees and computing perfect elimination schemes in parallel. *Inf. Process. Lett. 31*, 61–68.

KLEIN, P. 1996. Efficient parallel algorithms for chordal graphs. *SIAM J. Comput. 25*, 4, 797–827.

LEWIS, J. G., PEYTON, B. W., AND POTHEN, A. 1989. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM J. Comput. 10*, 6, 1146–1173.

LUNDQUIST, M. 1990. Zero patterns, chordal graphs, and matrix completions. Ph.D. thesis, Department of Mathematical Sciences, Clemson University.

MCKEE, T. A., AND MCMORRIS, F. R. 1999. *Topics in Intersection Graph Theory*. Society for Industrial and Applied Mathematics, Philadelphia, PA. Monograph.

NIKOLOPOULOS, S. D. 1995. Constant-Time parallel recognition of split graphs. *Inf. Process. Lett. 54*, 1–8.
ROSE, D. J., TARJAN, R. E., AND LUEKER, G. S. 1976. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput. 5*, 2, 266–283.
SLEATOR, D. D., AND TARJAN, R. E. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci. 26*, 362–391.
TARJAN, R. E., AND YANNAKAKIS, M. 1984. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput. 13*, 3, 566–579.
WALTER, J. R. 1978. Representations of chordal graphs as subtrees of a tree. *J. Graph Theory 2*, 265–267.
YELLIN, D. 1994. An algorithm for dynamic subset and intersection testing. *Theor. Comput. Sci. 129*, 397–406.

RECEIVED MARCH 2005; ACCEPTED MAY 2007