INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Dynamic Algorithms for Chordal and Interval Graphs

by

Louis Walter Ibarra
B.S., Johns Hopkins University, 1988
M.S., University of Virginia, 1990

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. Valerie King, Supervisor (Dept. of Computer Science)

Dr. John Ellis, Departmental Member (Dept. of Computer Science)

Dr. Michael Fellows, Departmental Member (Dept. of Computer Science)

Dr. Jing Huang, Outside Member (Dept. of Mathematics)

Dr. Pavol Hell, External Examiner (School of Computing Science, Simon Fraser University)

Supervisor: Dr. Valerie King

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, (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 introduce the clique-separator graph representation of a chordal graph, which provides significantly more information about the graph's structure than the well-known clique tree representation. We present fundamental properties of the clique-separator graph and additional properties when the input graph is interval. We then introduce the train tree representation of interval graphs and use it to decide whether there is a certain linear ordering of the graph's maximal cliques. This yields a fully dynamic algorithm to recognize interval graphs in $O(n \log n)$ time per edge insertion or deletion. The clique-separator graph may lead to dynamic algorithms for every proper subclass of chordal graphs, and the train tree may lead to fast dynamic algorithms for problems on interval graphs.

Examiners:

Dr. Valerie King, Supervisor (Dept. of Computer Science)

Dr. John Ellis, Departmental Member (Dept. of Computer Science)

Dr. Michael Fellows, Departmental Member (Dept. of Computer Science)

Dr. Jing Huang, Outside Member (Dept. of Mathematics)

Dr. Pavol Hell, External Examiner (School of Computing Science, Simon Fraser University)

Contents

List of Figures

Chapter 1

Introduction

1.1 Dynamic graph algorithms

A *dynamic graph algorithm* maintains a solution to a graph problem as the graph undergoes a series of small changes, such as single edge deletions or insertions. For every change, the algorithm updates the solution faster than recomputing the solution from scratch, i.e., with no previously computed information. For example, the World Wide Web is a dynamic graph whose vertices and edges represent network nodes and links that unpredictably may gain or lose capability as equipment fails, new equipment is introduced, or the network becomes congested. Typically, a dynamic graph algorithm has a preprocessing step to compute a solution, with some auxiliary information, for the initial graph. For example, our dynamic graph algorithm in Chapter 3 builds a data structure for the initial graph in $O(n^3)$ time and then updates it in $O(n \log n)$ time per edge insertion or deletion, whereas computing the solution from scratch (without previously computed information) requires $O(m + n)$ time, where m and n are respectively the number of edges and vertices in the graph.

A dynamic graph algorithm supports two operations: *query*, a question about the solution being maintained, e.g., "Are vertices u, v connected?" or "Is the graph bipar-

tite?", and *update*, the insertion or deletion of an edge. Dynamic graph algorithms for problems on weighted graphs also support updates that change an edge weight. Some algorithms support updates that delete or insert a vertex, e.g., [CH78, KS99, SP75]. An *insertions-only* or *deletions-only* dynamic graph algorithm respectively allows only edge insertions or only edge deletions. A *fully* dynamic graph algorithm allows both insertions and deletions. With some problems, maintaining a solution is easier when only insertions or only deletions are allowed. For example, a simple and fast insertions-only algorithm can be readily obtained for connectivity using disjoint-set union [Tar75], whereas fast fully dynamic algorithms are considerably more involved [HK99, HT98, HdT98].

Fully dynamic algorithms have been developed for numerous problems on undirected graphs, including connectivity [EGIN97, Fre85, HK99, HT98, HdT98], biconnectivity [EGIN97, HK95, Rau94], 2-edge connectivity [Fre97, HK99], bipartiteness [EGIN97, HK99, HK97], minimum spanning trees [CH78, EGIN97, Fre85, HK97, HdT98, SP75], and planarity [BT96, EGIS96, ILR93]. There are also algorithms for problems on plane graphs [EGIS96, EIT+92, Fre85, HRS94, Rau94]. Some of these results are described in survey papers [EGI98, FK99].

Hell, Shamir, and Sharan [HSS99] developed a fully dynamic algorithm to recognize proper interval graphs that maintains a representation of the proper interval graph. Their algorithm allows updates that result in a proper interval graph and supports queries to decide whether an edge update is allowed. In this thesis, we will present analogous algorithms for chordal graphs and interval graphs, which are classes that contain the proper interval graph class.

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 $m = |E|$. We write $H \subseteq G$ if H is a subgraph of G and $H \subset G$ if H is a proper subgraph of G. The subgraph of G *induced by* $U \subseteq V$ is $G[U] = (U, E[U])$, where $E[U] = \{\{u, v\} \in E \mid u, v \in U\}$. Given $S \subset V$, let $G - S$ denote $G[V - S]$. Let $G - \{u, v\}$ denote $(V, E - \{\{x, y\}\})$ and let $G + \{x, y\}$ denote $(V, E \cup \{\{x, y\}\})$.

A graph G is *chordal* (or *triangulated*) if every cycle of length 4 or more has a *chord*, which is an edge joining two nonconsecutive vertices of the cycle. Equivalently, G is chordal if no induced subgraph of G is a cycle of length greater than 3. Since every induced subgraph of a chordal graph is chordal, some authors describe being chordal as a *hereditary* property, e.g., [Gol80]. A graph G is *perfect* if for every induced subgraph H of G, the chromatic number of H is equal to the clique number of H. A graph G is the *intersection graph* of a family of sets $\mathcal{F} = \{S_1, \ldots, S_k\}$ if $\mathcal{F}$ is the vertex set of G and $\{S_i, S_j\}$ is an edge of G exactly when $i \neq j$ and $S_i \cap S_j \neq \emptyset$. Every chordal graph is perfect [Gol80] and every chordal graph is the intersection graph of a family of subtrees of a tree [Bun74, Gav74b, Wal72, Wal78], where the subtrees are viewed as sets. Golumbic [Gol80] and McKee and McMorris [MM99] discuss chordal graphs in the context of perfect graphs and intersection graphs, respectively.

There are linear time algorithms to recognize chordal graphs [RTL76, TY84] and to solve various problems on chordal graphs that are NP-complete on general graphs: CLIQUE, CHROMATIC NUMBER, INDEPENDENT SET, and PARTITION INTO CLIQUES [Gav72]. Chordal graphs have applications in biology, databases, statistics, facility location problems, and especially in sparse matrix computation [BP93, Gol80, MM99].

We will discuss several characterizations of chordal graphs, for which we require

some definitions. A set $U \subseteq V$ satisfying a property is *minimal* if there is no $U' \subseteq V$ satisfying the property such that $U' \subset U$, and *maximal* if there is no $U' \subseteq V$ satisfying the property such that $U' \supset U$. A *clique* of G is a set of pairwise adjacent vertices of G. An *independent set* of G is a set of pairwise nonadjacent vertices of G.

Let $S \subset V$. If two vertices are connected in G and not connected in $G - S$, then S is a *separator* of G. If $u, v \in V$ are connected in G and not connected in $G - S$, then S is a *uv-separator* of G. If $u, v \in V$ and S is a minimal uv-separator, then S is a *minimal vertex separator* of G. If $V_1, V_2 \subset V$ and S is a uv-separator of G for every $u \in V_1$ and $v \in V_2$, then S *separates* V_1, V_2. In Figure 1.1, $\{u, x, y\}, \{v, x, y\}$ are maximal cliques, $\{x, y\}, \{y\}$ are minimal vertex separators, and $\{y\}$ is a minimal separator.

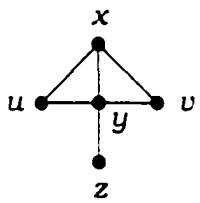


Figure 1.1: A graph.

We will discuss the characterizations of chordal graphs that are the most well known and most useful for our purposes. Other, less useful, characterizations of chordal graphs include [Duc84, Shi84, BCD+90, BJ96].

A graph G is chordal if and only if any of the following statements holds.

1. Every minimal vertex separator of G is a clique [Dir61].
2. G has a perfect elimination ordering [FG65].
3. G has a clique tree [Bun74, Gav74b, Wal72, Wal78].

Characterization 1

Using this characterization, we can give an algorithm that decides whether a graph is chordal: for every pair u, v of vertices, find every minimal uv-separator and test

whether it is a clique. Since there are $\Theta(n^2)$ vertex pairs and each test requires $O(n^2)$ time, not including the time to find the separators, this is a very inefficient algorithm when compared to the linear time algorithms we will subsequently describe. This characterization is important, however, for the clique-separator graph we define in Chapter 3.

Characterization 2

A vertex v is *simplicial* if $Adj(v)$ is a clique, where $Adj(v)$ is the set of neighbors of v. For example, the complete graph on n vertices has n simplicial vertices and a path has exactly 2 simplicial vertices. A *perfect elimination ordering (PEO)* of G is an ordering of its vertices $(v_1, v_2, \ldots, v_n)$ such that for each v_i, the set $Adj(v_i) \cap \{v_j \mid i < j\}$ is a clique. Equivalently, each v_i is a simplicial vertex in the subgraph induced by $\{v_i, \ldots, v_n\}$. For example, the graph in Figure 1.1 has perfect elimination orderings (u, v, x, y, z) and (z, u, x, y, v), which shows that the PEO may not be unique.

Every chordal graph has a simplicial vertex [Dir61]. This fact leads to a short proof [FG65] that a graph G is chordal if and only if G has a PEO: ($\Rightarrow$) Let v be any simplicial vertex of G. Since chordality is hereditary, then $G - v$ is chordal. By induction, $G - v$ has a PEO, which when appended to v yields a PEO of G. ($\Leftarrow$) Let C be a cycle of length 4 or more. Let v_i be the vertex of C with smallest index in the PEO. Since v_i is simplicial in the subgraph induced by $\{v_i, \ldots, v_n\}$, then v's two neighbors in C are connected by an edge, which is a chord of C.

Fulkerson and Gross [FG65] gave the first algorithm to compute a PEO: repeatedly find and delete any simplicial vertex until no vertices remain. The algorithm's correctness follows from the proof that a graph is chordal if and only if it has a PEO. Since a simplicial vertex can be found in $O(n^3)$ time, this algorithm runs in $O(n^4)$ time. There is an algorithm to recognize chordal graphs in $O(n^{\log_2 7})$ time using fast matrix multiplication [Gav74a], but it is superseded by algorithms that compute a

PEO in linear time. Rose, Tarjan, and Lueker [RTL76] gave the first linear time algorithm based on a graph search called Lexicographic BFS (LEX-BFS):

> Number the vertices from n to 1 in decreasing order, as follows. A vertex's label is the list of its numbered neighbors in decreasing order. Initially all labels are empty. Number the vertex with the largest label in lexicographic order, with ties broken arbitrarily, and repeat.

Every ordering produced by LEX-BFS could be produced by Breadth-First Search. Rose, Tarjan, and Lueker show that if a graph is chordal, then every ordering produced by LEX-BFS is a PEO. They implement LEX-BFS in linear time and give a linear time algorithm to test whether a given vertex ordering is a PEO. Thus, this algorithm recognizes chordal graphs in $O(m + n)$ time. Tarjan and Yannakakis [TY84] later gave another linear time algorithm using a simpler graph search called Maximum Cardinality Search (MCS):

> Number the vertices from n to 1 in decreasing order, as follows. Number the vertex with the most numbered neighbors, with ties broken arbitrarily, and repeat.

Tarjan and Yannakakis show that if a graph is chordal, then every ordering produced by MCS is a PEO. Using a heap, MCS runs in $O(m \log n + n \log n)$ time. Tarjan and Yannakakis implement MCS to run in $O(m + n)$ time and they use the algorithm from LEX-BFS to test whether a given vertex ordering is a PEO. Thus, this algorithm also recognizes chordal graphs in $O(m + n)$ time.

MCS can produce orderings that cannot be produced by LEX-BFS and vice versa, and there are PEOs that are cannot be produced by MCS or LEX-BFS. Shier [Shi84] gives an algorithm that can produce any PEO, although it does not run in linear time. Panda [Pan96] shows that Depth-First Search guided by MCS (the next vertex to visit

is the neighbor of the current vertex with the most numbered neighbors) computes a PEO of a chordal graph in linear time, as does Breadth-First Search guided by MCS.

A PEO is required for the linear time algorithms for CLIQUE, CHROMATIC NUMBER, INDEPENDENT SET, and PARTITION INTO CLIQUES on chordal graphs [Gav72]. A PEO is also important for several algorithms that recognize interval graphs, which we will discuss in the next section.

Characterization 3

Buneman [Bun74], Gavril [Gav74b], and Walter [Wal72, Wal78] independently discovered that a graph is chordal if and only if it is the intersection graph of a family of subtrees of a tree. In particular, they showed that a graph G is chordal if and only if G has a *clique tree*, which is a tree T on the maximal cliques of G with the *clique intersection property*: for any two maximal cliques K, K', the set $K \cap K'$ is contained in every maximal clique on the K-K' path in T.

Let $\mathcal{K}_G$ be the set of maximal cliques of G and let $\mathcal{K}_G(v)$ be the set of maximal cliques of G containing vertex v. It is straightforward to show that the clique intersection property is equivalent to the *induced subtree property*: for any vertex v, the subgraph of T induced by $\mathcal{K}_G(v)$ is a tree. Let T_v be the subtree of T induced by $\mathcal{K}_G(v)$. Since $\{u, v\}$ is an edge of G if and only if u, v are both contained in a maximal clique of G, then $\{u, v\}$ is an edge of G if and only if T_u and T_v intersect. Thus, G is the intersection graph of the family of subtrees $\{T_v \mid v \in V\}$.

The *weighted clique intersection graph* W_G of G is the weighted graph on the maximal cliques of G where $\{K, K'\}$ is an edge if and only if $K \cap K' \neq \emptyset$, and each edge $\{K, K'\}$ has weight $|K \cap K'|$. W_G is connected if and only if G is connected. Bernstein and Goodman [BG81] showed that 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.

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

- T is a clique tree of G.
- T has the clique intersection property.
- T has the induced subtree property.
- T is a maximum-weight spanning tree of W_G (if G is connected).

Ho and Lee [HL89] and Lundquist [Lun90] independently discovered the following property of clique trees, which we will use extensively, especially in the interval graph algorithms in Chapter 3.

Theorem 1.1 *Let G be a connected chordal graph with clique tree T.*

1. *A set S is a minimal vertex separator of G if and only if $S = K \cap K'$ for some $\{K, K'\} \in E(T)$.*
2. *If $S = K \cap K'$ for $\{K, K'\} \in E(T)$, then S separates $K - S$ and $K' - S$.*

Fulkerson and Gross [FG65] showed that a chordal graph with n vertices has at most n maximal cliques. By Theorem 1.1.1, a chordal graph with n vertices has at most $n-1$ minimal vertex separators. Also, if S satisfies the premise of Theorem 1.1.2, then S is a minimal uv-separator for any $u \in K - S$ and $v \in K' - S$ because every vertex of S is adjacent to every vertex of $K - S$ and $K' - S$, which means S is a minimal vertex separator.

Buneman, Gavril, and Walter showed how to construct a clique tree of a chordal graph in polynomial time, for example, Gavril's algorithm runs in $O(n^4)$ time. Subsequently, others developed linear time algorithms to compute a clique tree of a chordal graph, as well as its maximal cliques and minimal vertex separators. Tarjan and Yannakakis [TY84] implicitly gave the first linear time algorithm in the context of acyclic

hypergraphs for relational databases. Lewis, Peyton, and Pothen [LPP89] gave the first explicit linear time algorithm in the context of sparse matrix computation. Blair and Peyton [BP93] gave a simpler algorithm in a graph theory context. Each of these linear time algorithms uses MCS.

The text by Golumbic [Gol80] describes the PEO and the clique tree, but does not include more recent clique tree results like Theorem 1.1. The excellent primer by Blair and Peyton [BP93] emphasizes the clique tree and its applications in sparse matrix computation. Although both characterizations have been very useful in developing algorithms for chordal graphs, both may not be equally useful in developing dynamic algorithms for chordal graphs. Since the PEO is computed with a graph search such as Lex-BFS or MCS, a single edge update could yield a very different PEO. There are properties that imply a given ordering is a PEO, for example Property P in [TY84], but again, a single edge update could yield a very different ordering. Thus, the PEO appears to be difficult to maintain under edge updates. In contrast, in Chapter 2, we present a simple algorithm that maintains a chordal graph's clique tree in $O(n)$ time per update.

1.3 Interval graphs

A graph is *interval* if it is the intersection graph of a set of intervals on the real line. There are as many characterizations of interval graphs as there are of chordal graphs. We will briefly discuss the most well known characterizations, for which we require three definitions. An *asteroidal triple* is a set of three vertices such that between any two of the vertices, there is a path that does not contain a neighbor of the third vertex. An asteroidal triple is an independent set. The complement of a graph $G = (V, E)$ is the graph $\bar{G} = (V, \bar{E})$, where $\{u, v\} \in \bar{E}$ if and only if $\{u, v\} \notin E$. A graph is *comparability* (or *transitively orientable*) if there is an orientation of its edges such

that if $(u, v), (v, w)$ are directed edges, then (u, w) is a directed edge.

A graph G is interval if and only if any of the following statements holds.

1. G is chordal and G has no asteroidal triple [LB62].

2. G is chordal and $\bar{G}$ is a comparability graph [GH64].

3. There is a linear ordering of the maximal cliques of G such that the maximal cliques containing any given vertex are consecutive in the ordering [GH64].

The linear ordering in the third characterization is a clique tree of G that is a path, which is a *clique path* of G. It follows that a graph is interval if and only if it is the intersection graph of a family of subpaths of a path. Therefore, the interval graph class is a subclass of the chordal graph class. It is a proper subclass, as shown by the net, which is the graph formed by attaching an edge to each corner of a triangle to form a graph with 6 vertices. Since not every clique tree of an interval graph is a clique path, a clique tree algorithm may not produce a clique path when applied to an interval graph.

Interval graphs often model problems involving a linear order. Consequently, interval graphs have even more applications than chordal graphs, including applications in archeology, biology, genetics, psychology, and scheduling [Gol80, MM99]. Furthermore, most well known NP-complete problems can be solved in polynomial time on interval graphs, including HAMILTONIAN CIRCUIT, DOMINATING SET, and GRAPH ISOMORPHISM [Joh85]. Bodlaender [Bod89] showed that ACHROMATIC NUMBER is NP-complete for interval graphs, which is one of the few known NP-completeness results for interval graphs [Joh85].

There are numerous algorithms to recognize interval graphs, including the following linear time algorithms. Booth and Lueker [BL76] gave the first (linear time) algorithm and introduced the PQ-tree, which represents a set of permutations. Their

algorithm computes the maximal cliques and then uses a complicated set of templates to construct a PQ-tree that represents the set of valid orderings of the maximal cliques. Korte and Möhring [KM89] gave an algorithm that constructs an MPQ-tree, which is a simpler version of the PQ-tree. This construction uses a LEX-BFS ordering and depends on LEX-BFS properties that MCS does not have. Habib, Paul, and Viennot [HMPV00] gave an algorithm that computes a clique tree and then uses a LEX-BFS ordering to manipulate the clique tree into a clique path. Subsequently, others have developed algorithms that do not compute a linear ordering of the maximal cliques: Hsu and Ma [HM99] gave an algorithm that uses a substitution decomposition computed with Lex-BFS, and Corneil, Olariu, and Stewart [COS98] gave a particularly simple algorithm that uses four sweeps of Lex-BFS, with ties broken in a particular way.

1.4 Other classes

We describe one subclass of interval graphs. A graph is *proper interval* (or *unit interval*) if it is the intersection graph of a set of intervals on the real line such that no interval is properly contained in another. The subclass is proper, as shown by the claw ($K_{1,3}$), which is the graph consisting of one vertex adjacent to three nonadjacent vertices. There are linear time algorithms to recognize proper interval graphs derived from the Booth and Lueker algorithm and the Korte and Möhring algorithm. There are also simpler linear time algorithms that do not use PQ-trees or MPQ-trees, including algorithms by Corneil, Kim, Natarajan, Olariu, and Sprague [CKN+95], Deng, Hell, and Huang [DHH96], and de Figueiredo, Meidanis, and de Mello [dFMdM95]. Hell, Shamir, and Sharan [HSS99] gave a fully dynamic algorithm to recognize and represent proper interval graphs in $O(\log n)$ time per edge insertion or deletion.

We describe one more subclass of chordal graphs. A graph is *split* if its vertex set can be partitioned into a clique K and an independent set I, with no restriction on edges between vertices in K and vertices in I. A graph G is split if and only if G and its complement $\bar{G}$ are chordal [FH77]. Split graphs and interval graphs are incomparable classes.

Johnson [Joh85] discusses these and other graph classes with "broad algorithmic significance", including *undirected path graphs*, *directed path graphs*, and *strongly chordal graphs*. The Golumbic [Gol80] text and the McKee and McMorris monograph [MM99] discuss many of the same classes in the context of perfect graphs and intersection graphs, respectively.

1.5 Results

In this thesis, we present the first dynamic algorithms that recognize chordal graphs and interval graphs. Both algorithms maintain a graph representation with $O(n)$ size. In Chapter 2, we present the dynamic algorithm that recognizes a chordal graph by maintaining its clique tree. The algorithm allows updates that result in a chordal graph and supports queries to decide whether an edge update is allowed. In a very simple implementation, each operation runs in $O(n)$ time, and in a more complicated implementation, one operation runs in $O(\log^2 n)$ time, another in $O(n \log n)$ time, and the others in $O(n)$ time.

In Chapter 3, we introduce the *clique-separator graph* representation of a chordal graph, which is a graph on its maximal cliques and minimal vertex separators. We prove fundamental properties of the clique-separator graph and additional properties when the input graph is interval. We then introduce the *train tree* representation of interval graphs and use it to decide whether there is a clique path on the graph's maximal cliques. This yields the dynamic algorithm to recognize interval graphs in

$O(n \log n)$ time per edge insertion or deletion. The algorithm allows updates that result in an interval graph and supports queries to decide whether an edge update is allowed. The clique-separator graph and the train tree are representations that provide significantly more information about the graph's structure than the clique tree representation.

Chapter 2

A dynamic algorithm for chordal graphs

2.1 Introduction

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) returns "yes" if $G - \{u, v\}$ is chordal and "no" otherwise.
- *Delete*(u, v) deletes $\{u, v\}$ from G, provided $G - \{u, v\}$ is chordal.
- *Insert-Query*(u, v) returns "yes" if $G + \{u, v\}$ is chordal and "no" otherwise.
- *Insert*(u, v) 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, *Delete-Query* 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. Both implementations improve

the $O(m+n)$ time obtained by running a static algorithm (e.g. [RTL76]) for each operation. All of the bounds in this paper 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 [HK99] 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 [Fre85] combined with sparsification [EGIN97], but the resulting data structure is much more complicated than ET-trees.

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 [BHT01] or [RTL76] 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 [BHT01] 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 [Heg]. Minimal filled graphs are desirable in sparse matrix computation, as well as other areas of computer science. Since f is $O(n)$ for many practical matrices [BHT01], $O(nf+m)$ compares favorably with the best known running time for computing *any* minimal filled graph of G, which is $O(nm)$ [RTL76].

2.2 Clique trees

By Theorem 1.1.1, a clique tree of G has nodes and edges that correspond to the maximal cliques and minimal vertex separators of G, respectively. Figure 2.1 shows a chordal graph G and Figure 2.2 shows a clique tree of G. The set $S = K_x \cap K_y = \{c, d\}$

is a minimal uv-separator for every $u \in K_x - S = \{a, b\}$ and $v \in K_y - S = \{g\}$. Also, 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 will refer to the *vertices* of G and the *nodes* of T and will usually use s, t, u, v as vertex names and w, x, y, z as node names. In this chapter only, node $x \in T$ corresponds to maximal clique K_x of G and an edge $\{x, y\} \in T$ has weight $w(x, y)$.

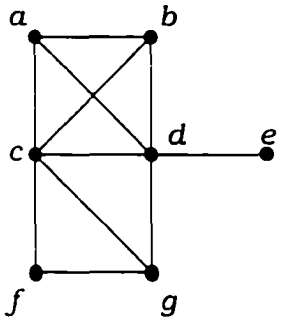


Figure 2.1: A chordal graph G.

Most discussions of chordal graphs, such as [BP93, LPP89], 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.

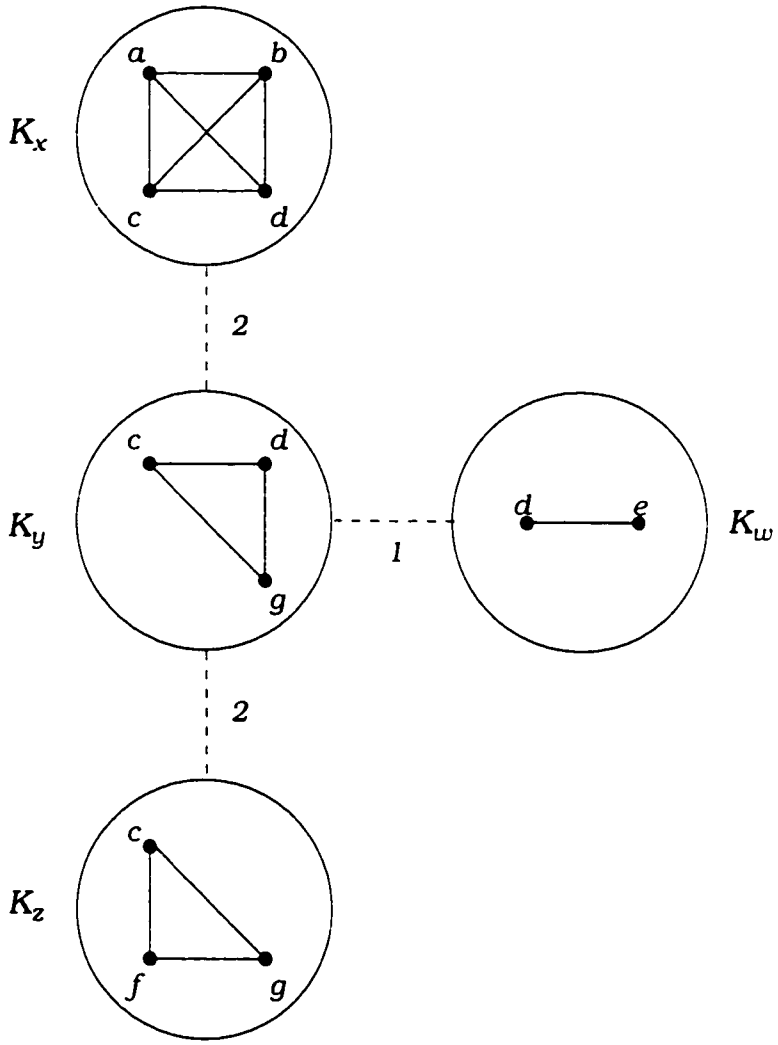


Figure 2.2: A clique tree T of the graph in Figure 1.1.

2.3 The dynamic algorithm for chordal graphs

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

2.3.1 Delete-Query

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

Lemma 2.1 ([RTL76]) *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 2.2 ([GJSW84]) *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: 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 [GJSW84].

Theorem 2.3 *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 and v.*

Proof ($\Rightarrow$) By Lemmas 2 and 3. ($\Leftarrow$) 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, then $\{u, v, s\}$ 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. □

Delete-Query(u, v)

For every node x of T, test whether $\{u, v\} \subseteq K_x$. If there is exactly one such node, return "yes", and otherwise, return "no".

End Delete-Query

2.3.2 Delete

We next show how to update T for $G - \{u, v\}$, given that T has a unique node x such that $\{u, v\} \subseteq K_x$. 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 2.3.)

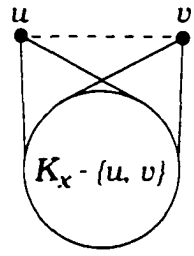


Figure 2.3: Deleting $\{u, v\}$ from K_x.

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

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

(See Figure 2.4.) Let $k = |K_x|$. Since T is a clique tree, then for any $y \in N(x)$, $w(x, y) \leq k - 1$.

Then

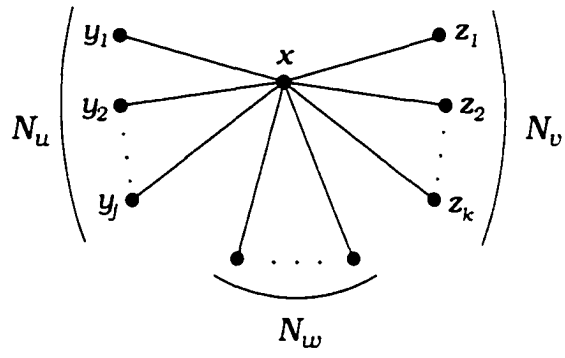


Figure 2.4: Before deleting $\{u, v\}$.

K_x^u is not maximal in $G - \{u, v\}$

if and only if $\exists y \in T, y \neq x, K_x^u \subset K_y$

if and only if $\exists y \in N_u, K_x^u \subset K_y$ (by the clique intersection property)

if and only if $\exists y \in N_u, K_x \cap K_y = K_x^u$

if and only if $\exists y \in N_u, w(x, y) = k - 1$

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 $\{u, v\} \subseteq K_x$. 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 representing K_x^u and K_x^v, respectively, 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 $\{x_1, y\}$. For each $z \in N_v$, replace $\{x, z\}$ with $\{x_2, z\}$. For each $w \in N_w$, replace $\{x, w\}$ with $\{x_1, w\}$ (or $\{x_2, w\}$). Observe that T has the induced subtree property. (See Figure 2.5.)

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 $y_i \in N_u$, contract $\{x_1, y_i\}$ and replace x_1 with y_i. This

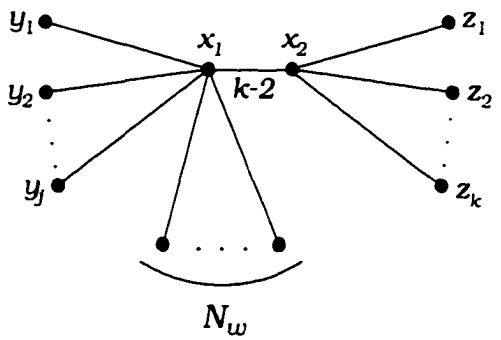


Figure 2.5: Deleting $\{u, v\}$.

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 $z_i \in N_v$, contract $\{x_2, z_i\}$ and replace x_2 with z_i. Thus, x has been replaced with 0, 1, or 2 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, i.e., 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, i.e., $K_x = \{u, v\}$ for some $x \in T$.

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 induced by $\mathcal{K}_G(u)$ and $\mathcal{K}_G(v)$, respectively, then T_u and T_v do not intersect (because no clique contains $\{u,v\}$).

Theorem 2.4 *Let G be a chordal graph without edge $\{u,v\}$. Then $G+\{u,v\}$ is chordal if and only if G has a clique tree T with $u \in K_x, v \in K_y$ for some $\{x,y\} \in E(T)$.*

Proof ($\Rightarrow$) Since $G' = G+\{u,v\}$ is chordal, then 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 $\{x,y\} \in E(T)$. ($\Leftarrow$) 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, then $u \notin K_y, v \notin K_x$ and thus $u \in K_x - I, v \in K_y - I$. (See Figure 2.7.) 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 C. Hence, $G''+\{u,v\}$ is chordal. □

Suppose G has a clique tree T that does not satisfy the condition in Theorem 5. The following theorem shows that if G has another clique tree T' that does satisfy the condition, then T' differs from T by one edge.

Theorem 2.5 *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 $\{x,y\} \notin E(T)$.*

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

Proof ($\Leftarrow$) Since $T'' = T - e + \{x, y\}$ and T have the same weight, then T'' is a clique tree of G. Moreover, $u \in K_x, v \in K_y$ and $\{x, y\} \in E(T'')$. ($\Rightarrow$) 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 2.6.) By assumption, we have $\{x', y'\} \notin T$. Furthermore, x, y are on the x'-y' path P' in T, which implies 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 $f \in P$, $w(f) \geq w(x, y) \geq w(x', y')$.

We now consider T'. Let V_1, V_2 be the node sets of the trees of $T' - \{x', y'\}$, where $x' \in V_1$ and $y' \in V_2$. Consider the cut $[V_1, V_2]$ of W_G. Since $P' + \{x', y'\}$ is a cycle containing an edge crossing $[V_1, V_2]$ (namely, $\{x', y'\}$), it must contain an edge $e \in P'$ crossing $[V_1, V_2]$. Since $\{x', y'\} \in T'$, then $e \notin T'$. 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 $V(T_u) \subseteq V_1$ and $V(T_v) \subseteq V_2$. But e crosses $[V_1, V_2]$, which means $e \notin P' - P$. Therefore, $e \in P$. Then $w(e) \geq w(x, y) \geq w(x', y')$. Since $w(e) \leq w(x', y')$, then $w(e) = w(x, y) = w(x', y')$. 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 $w(e) = w(x, y) = 0$. Then $T - e + \{x, y\}$ is a clique tree satisfying the condition in Theorem 5.

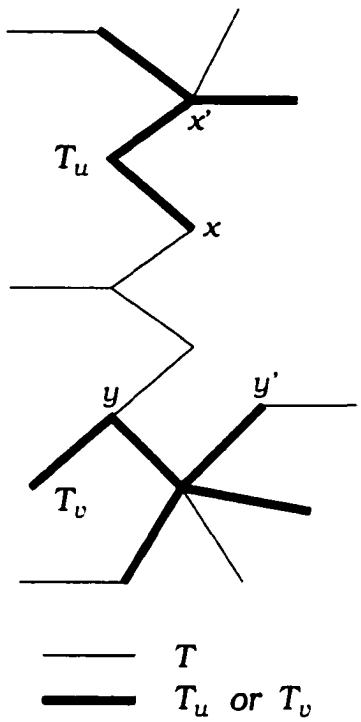


Figure 2.6: Clique tree T.

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". (We cannot have $w(e) < w(x, y)$ because T is a clique tree.)

End Insert-Query

2.3.4 Insert

We next show how to update T for $G + \{u, v\}$, given that $u \in K_x, v \in K_y$ and $\{x, y\} \in E(T)$. Let $I = K_x \cap K_y$. Then $K = I \cup \{u, v\}$ is a clique in $G + \{u, v\}$. (See Figure 2.7.)

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 $G - I$. But by

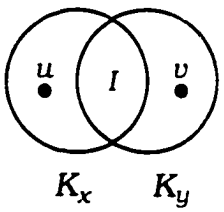


Figure 2.7: Inserting $\{u, v\}$.

Theorem 1, I is a uv-separator of G, a contradiction. Thus, K is a maximal clique in $G + \{u, v\}$. 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 does not contain the edge $\{u, v\}$, 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 $v \notin K_x$, then $K_x \subset K_z$ if and only if $K_x = I \cup \{u\}$ if and only if $|K_x| = |I| + 1$. Similarly, $K_y \subset K_z$ if and only if $K_y = I \cup \{v\}$ if and only if $|K_y| = |I| + 1$. Thus, we decide whether K_x or K_y is maximal in $G+\{u, v\}$ by comparing $|K_x|, |K_y|$, and $|I| = w(x, y)$.

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 2.8.) 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. Otherwise, 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

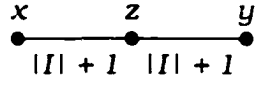


Figure 2.8: Adding node z to T.

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, $K_z = \{u, v\}$.

2.4 First implementation

In this simple implementation, we represent each maximal clique with a characteristic vector, i.e., an array of n bits. Then 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 [BP93]. 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* and *Insert-Query* by searching T. Since T has at most n nodes, each operation runs in $O(n)$ time.

We implement *Delete* by searching T to find node x and then examining x's neighbors. Since there are $O(n)$ changes to T, *Delete* runs in $O(n)$ time. We implement *Insert* by searching T to find x and y. We must then compute z's characteristic vector. If K_x and K_y are both maximal, we compute the intersection vector of x

and y's vectors and add u, v to it. Otherwise, we copy x or y's vector and add the appropriate vertices (u or v or both) to it. Since each vector has length n, *Insert* requires $O(n)$ time.

2.4.1 Connected components

We can readily maintain the connected components of G by using a degree-d ET-tree [HK99] to represent a spanning tree for each component, where $d = n^{\epsilon}, 0 < \epsilon < 1$. Deciding whether two vertices are in the same ET-tree requires $O(1)$ time, so each "Are vertices u, v connected?" query runs in $O(1)$ time. The descriptions of *Delete* and Insert show how to decide whether deleting an edge disconnects a component and whether inserting an edge connects two components. Since splitting or joining the ET-trees requires $O(n^{\epsilon})$ time, each update runs in $O(n)$ time.

The only difficulty occurs when an edge $\{u, v\}$ to be deleted is contained in its component's spanning tree T_{sp} but it 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, then 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.

2.5 Second implementation

In this implementation, we represent each maximal clique with a characteristic vector and the clique tree T with a Sleator-Tarjan dynamic tree [ST83]. For every vertex $v \in V$, we maintain a pointer to any node $z \in T$ such that $v \in K_z$. We implement

Insert-Query(u, v) as follows.

Let $u \in K_w$ and $v \in K_z$. Reroot the dynamic tree for T at w. Use binary search on the path in T from z to the root to 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 by rerooting at x and finding the minimum weight edge from y to the root. If $w(e) = w(x, y)$, return "yes"; if $w(e) > w(x, y)$, return "no".

Each of the following dynamic tree operations requires $O(\log n)$ time: rerooting the tree, finding the ith node on a path to the root, finding the minimum weight edge on a path to the root, splitting the tree by deleting an edge or joining two trees with an edge. Thus, *Insert-Query* requires $O(\log^2 n)$ time. Since *Delete* makes $O(n)$ changes to T, it 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 $O(n \log n)$.

2.5.1 Computing the weights

In the first implementation of *Insert-Query*, we computed $w(x, y) = |K_x \cap K_y|$ in $O(n)$ time by comparing the characteristic vectors for K_x and K_y. In the second implementation, we compute $w(x, y)$ in $O(1)$ time by maintaining a $n \times n$ matrix W with $W(x, y) = w(x, y)$. The preprocessing time is dominated by the $O(n^3)$ time to build W. We update W in $O(n)$ time after *Insert* or *Delete*, as follows.

After *Insert*(u, v) adds node z to T and creates z's characteristic vector, $K_z = I \cup \{u, v\}$, 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 2.8.) Let $w \neq x, y, z$. We 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 $w \in T_x$. By the clique intersection property, $K_w \cap K_y \subseteq K_x$. (See Figures 7 and 8.) Moreover, $K_w \cap K_y = K_w \cap I$, which implies $|K_w \cap K_z| = |K_w \cap K_y| + |K_w \cap \{u\}|$. Therefore, if $u \in K_w$, then $W(w,z) = W(w,y) + 1$, and otherwise, $W(w,z) = W(w,y)$. We proceed similarly if $w \in T_y$. In every case, we compute $W(w,z)$ in $O(1)$ time for each w. Thus, we update W in $O(n)$ time after *Insert*.

We now consider *Delete*. Deleting edge $\{u,v\}$ 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 the corresponding W entries. Otherwise, since $K_x^u = K_x - \{v\}$ and $K_x^v = K_x - \{u\}$, then 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, we update W in $O(n)$ time after *Delete*.

2.6 Conclusions and open problems

Since a chordal graph is perfect [Gol80], 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 [Gav72] requires a perfect elimination ordering of the vertices, which appears to be difficult to maintain under edge deletions and insertions.

Most classical algorithms for chordal graphs are based on perfect elimination or-

derings. 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.

Chapter 3

A dynamic algorithm for interval graphs

3.1 Introduction

In spite of the large literature on chordal graphs and clique trees, we are not aware of any papers on the characteristics of clique trees for any chordal graph subclass. This is probably because restricting the graph does not restrict its clique tree in general. For example, if G is a star ($K_{1,n}$), then every tree on its maximal cliques is a clique tree of G, even though G is chordal, interval, and split. The structure of a clique tree provides little information about the structure of the graph, even given the sizes of the maximal cliques, and very different graphs may have the same clique tree. For example, if G is interval and each maximal clique of G has size 2, then G may be a path (which has exactly one clique tree) or a star (which has all possible clique trees) or some combination of the two.

Given a chordal graph G, we will define G's *clique-separator graph* $\mathcal{G}$ to be a graph whose nodes represent the maximal cliques and minimal vertex separators of G and whose arcs (directed edges) and edges (undirected edges) represent the con-

tainment relations between the sets corresponding to the nodes. The clique-separator graph is unique, unlike the clique tree and the PEO, and it has numerous structural properties. Moreover, when the graph G is interval, proper interval, split, etc., its clique-separator graph $\mathcal{G}$ has additional structural properties. Consequently, the clique-separator graph may lead to recognition and dynamic algorithms for every well known subclass of chordal graphs.

In Section 3.2, we present fundamental properties of the clique-separator graph $\mathcal{G}$, including the following. We show that (a) the nodes of $\mathcal{G}$ are partitioned into trees that have the clique intersection property and that contain all the edges and none of the arcs of $\mathcal{G}$, (b) the graph formed by contracting each tree into a single node is a directed acyclic graph, which means there is a topological sort of the trees, (c) in any topological sort of the trees, if $\mathcal{H} \subset \mathcal{G}$ consists of a "tail" of the topological sort and a cut node separates two nodes in $\mathcal{H}$, then the cut node separates the same two nodes in $\mathcal{G}$, (d) the nodes of $\mathcal{G}$ separated by a cut node S of $\mathcal{G}$ specify the vertices of G separated by the minimal vertex separator corresponding to S. (The last property is analogous to Theorem 1.1.2, but is much stronger.)

In Section 3.3, we present properties of $\mathcal{G}$ when G is interval. We show that (a) after removing the uninteresting leaves, every tree of $\mathcal{G}$ is a path P such that any clique path of G orders the maximal cliques in the same order as P, (b) the minimal vertex separators contained in a maximal clique can be partitioned into at most two chains, where each separator is contained in its successor in the chain. The first property is used by the train tree algorithm and the second by the update algorithm.

In Section 3.4, we describe $\mathcal{G}$'s *train tree*, which is a PQ-tree [BL76] representing all valid orderings of the maximal cliques *and* minimal vertex separators of G. In the Booth and Lueker algorithm to recognize interval graphs [BL76], the PQ-tree represents valid orderings of the maximal cliques only and it is built with a complicated set of templates. In contrast, the train tree is built with a much simpler algorithm

that uses none of the PQ-tree templates. The advantage of including the minimal vertex separators is that the intersection of any two cliques is contained in some minimal vertex separator "between" the two cliques in the train tree, which reduces the difficulty of computing a valid ordering.

In Section 3.5, we present the *train tree algorithm*, which builds a train tree of $\mathcal{G}$ in $O(n)$ time. This algorithm builds the train tree incrementally by examining the trees of $\mathcal{G}$ in reverse topological order and merging each tree into the current train tree. At each step, the current train tree contains all trees of $\mathcal{G}$ in a "tail" of the topological sort.

In Section 3.6, we present the dynamic algorithm to recognize interval graphs in $O(n \log n)$ time per edge insertion or deletion. We show that an edge update requires only a small, local change in $\mathcal{G}$ because at most two new maximal cliques or minimal vertex separators are created, each of which differs from an old maximal clique or minimal vertex separator by at most two vertices. The main difficulty is that if S is a new minimal vertex separator, we must find the minimal vertex separators that contain S and that S contains. With edge insertions, these minimal vertex separators on at most two chains and we can readily find them. With edge deletions, we must find the minimal vertex separators using the train tree and a graph search.

In Section 3.7, we give a simple algorithm that builds the clique-separator graph of a chordal graph in $O(n^3)$ time.

3.2 The clique-separator graph of a chordal graph

We require two terms from partially ordered set theory. A collection of sets U is a *chain* if for any distinct elements $S, S' \in U$, $S \subseteq S'$ or $S' \subseteq S$, and an *antichain* if for any distinct elements $S, S' \in U$, $S \not\subseteq S'$ and $S' \not\subseteq S$. In any graph, the set of maximal cliques is an antichain but the set of minimal vertex separators may not be.

In any chordal graph, every minimal vertex separator is contained in some maximal clique(s). In Figure 3.1. $\{u, x, y\}, \{v, x, y\}$ are maximal cliques and $\{x, y\}, \{y\}$ are minimal vertex separators.

Recall that by Theorem 1.1.1. a chordal graph has at most $n - 1$ minimal vertex separators. Also, if S satisfies the premise of Theorem 1.1.2. then S is a minimal uv-separator for any $u \in K - S$ and $v \in K' - S$ because every vertex of S is adjacent to every vertex of $K - S$ and $K' - S$.

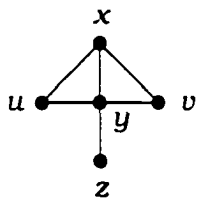


Figure 3.1: A graph.

3.2.1 The clique-separator graph

The *clique-separator graph* $\mathcal{G}$ of G has arcs (directed edges) and edges (undirected edges), as follows.

- Each maximal clique K of G is represented by a *clique node* K of $\mathcal{G}$. Each minimal vertex separator S of G is represented by a *separator node* S of $\mathcal{G}$.
- Each arc is from a separator node to a separator node. $\mathcal{G}$ has arc (S, S'') if $S \subset S''$ and there is no separator node S' such that $S \subset S' \subset S''$.
- Each edge is between a separator node and a clique node. $\mathcal{G}$ has edge $\{S, K\}$ if $S \subset K$ and there is no separator node S' such that $S \subset S' \subset K$.

Figure 3.2 shows a chordal graph G and Figure 3.3 shows its clique-separator graph $\mathcal{G}$; a separator node's superscript is its size. We will refer to the *vertices* of G and the *nodes* of $\mathcal{G}$ and use lowercase variables for vertices and uppercase variables

for nodes. We will usually use "maximal clique" ("minimal vertex separator") and "clique node" ("separator node") interchangeably.

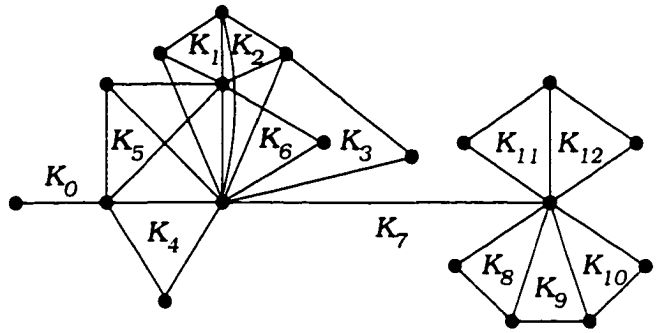


Figure 3.2: A chordal graph G.

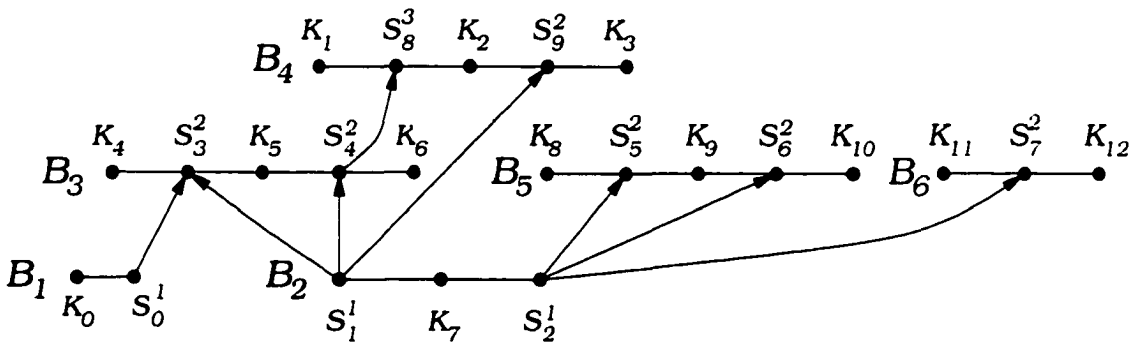


Figure 3.3: The clique-separator graph $\mathcal{G}$ of G. The superscripts indicate the set's size.

Let K be a clique node and let S, S' be distinct separator nodes of $\mathcal{G}$. If $\{S, K\}$ is an edge of $\mathcal{G}$, then S and K are *neighbors* and S is *adjacent* to K. If $\mathcal{G}$ contains a directed path from S to S', then S is a *predecessor* of S', denoted $S \rightsquigarrow S'$. If $\mathcal{G}$ contains arc (S, S'), then S is an *immediate predecessor* of S', denoted $S \rightarrow S'$. We define *successor* and *immediate successor* similarly. Let $degree(S), indegree(S), outdegree(S)$ be the number of neighbors, immediate predecessors, immediate successors of S, respectively. If S is a neighbor of K or a predecessor of a neighbor of K, then S is a *predecessor* of K and K is a *successor* of S, denoted $S \rightsquigarrow K$. Observe that $S \subset S'$ if and only if $S \rightsquigarrow S'$ and that $S \subset K$ if and only if $S \rightsquigarrow K$. Also, the set of immediate

predecessors (or immediate successors) of S is an antichain and the set of neighbors of K is an antichain.

Let $\mathcal{S}$ be a set of nodes of $\mathcal{G}$. A node $X \in \mathcal{S}$ is a *minimal node of* $\mathcal{S}$ if there is no $Y \in \mathcal{S}$ such that $Y \subset X$, and a *maximal node of* $\mathcal{S}$ if there is no $Y \in \mathcal{S}$ such that $Y \supset X$. Given a clique tree T of G, let $\mathcal{S}_T(K)$ be the set of minimal vertex separators $\{K \cap K' \mid \{K, K'\} \in E(T)\}$. The following lemma relates the clique tree and the clique-separator graph.

Lemma 3.1 *Let G be a chordal graph with clique tree T and clique-separator graph $\mathcal{G}$. Let K be a clique node of $\mathcal{G}$.*

1. *$\{S, K\}$ is an edge of $\mathcal{G}$ if and only if S is a maximal node of $\mathcal{S}_T(K)$.*
2. *For any clique node $K' \neq K$, $K' \cap K$ is contained in some neighbor of K in $\mathcal{G}$.*
3. *For any separator node S, $S \cap K$ is contained in some neighbor of K in $\mathcal{G}$.*
4. *For any vertex $v \in K$, K is the unique node of $\mathcal{G}$ containing v if and only if v is not contained in any neighbor of K in $\mathcal{G}$.*

Proof

1. Suppose $\{S, K\}$ is an edge of $\mathcal{G}$. By Theorem 1.1.1, there is a clique node $K' \neq K$ such that $S \subset K'$. Then $S \subseteq K \cap K'$. By the clique intersection property, for some neighbor K'' of K in T, $S \subseteq K \cap K''$. Since $K \cap K''$ is a minimal vertex separator of G by Theorem 1.1.1, if $S \subset K \cap K''$, then $\{S, K\}$ is not an edge of $\mathcal{G}$, a contradiction. Thus, $S = K \cap K'' \in \mathcal{S}_T(K)$. Now if $S \subset S'$ for some $S' \in \mathcal{S}_T(K)$, then again $\{S, K\}$ is not an edge of $\mathcal{G}$, a contradiction. Thus, S is a maximal element of $\mathcal{S}_T(K)$.

   Suppose $\{S, K\}$ is not an edge of $\mathcal{G}$. If $S \not\subset K$, then $S \notin \mathcal{S}_T(K)$. If $S \subset K$, then there is a separator node S' such that $S \subset S' \subset K$ and $\{S', K\}$ is an edge

of $\mathcal{G}$. By the preceding argument, $S' \in \mathcal{S}_T(K)$ and thus S is not a maximal element of $\mathcal{S}_T(K)$.

2. By the clique intersection property, for some neighbor K'' of K in T, $K' \cap K \subseteq K'' \cap K$, which is a minimal vertex separator contained in some maximal element S of $\mathcal{S}_T(K)$. By part 1, S is a neighbor of K in $\mathcal{G}$.

3. By Theorem 1.1.1, $S \subset K'$ for some clique node $K' \neq K$. Then $S \cap K \subseteq K' \cap K$. By part 2, $K' \cap K$ is contained in some neighbor of K in $\mathcal{G}$.

4. By parts 2 and 3. □

3.2.2 Definitions

We require several definitions before presenting the Main Theorem of clique-separator graphs. Let $G = (V, E)$ be a chordal graph with clique-separator graph $\mathcal{G}$. Let $P = (X_0, X_1, \ldots, X_k), k \geq 0$, be a sequence of distinct nodes of $\mathcal{G}$. If (X_i, X_{i+1}) is an arc, $0 \leq i < k$, then P is a *directed path*. If (X_i, X_{i+1}) is an arc or $\{X_i, X_{i+1}\}$ is an edge, $0 \leq i < k$, then P is a *semidirected path*. If P is a semidirected path and (X_k, X_0) is an arc or $\{X_k, X_0\}$ is an edge, $k \geq 2$, then $(X_0, X_1, \ldots, X_k, X_0)$ is a *semidirected cycle*.

Given a set of nodes $\mathcal{N}$, let $V(\mathcal{N}) = \{v \in X \mid X \in \mathcal{N}\}$. Given $\mathcal{H} \subseteq \mathcal{G}$, let $\mathcal{N}(\mathcal{H})$ be the set of nodes of $\mathcal{H}$ and let $V(\mathcal{H}) = V(\mathcal{N}(\mathcal{H}))$. Then $V(\mathcal{G}) = V$. We say $\mathcal{H}$ is *connected* if the underlying undirected graph of $\mathcal{H}$ (obtained by replacing every arc with an edge) is connected. $\mathcal{G}$ is connected if and only if G is connected. If $\mathcal{H}$ is connected, then $G[V(\mathcal{H})]$ is connected, but the converse may not hold, e.g., let $\mathcal{H}$ contain exactly the nodes K_0, K_4 in Figure 3.3.

A *box* of $\mathcal{G}$ is a connected component of the subgraph of $\mathcal{G}$ obtained by deleting all arcs. Then every node of $\mathcal{G}$ is contained in a unique box of $\mathcal{G}$. A box is *isolated* if

it contains exactly one clique node and no separator nodes. A box is isolated if and only if it corresponds to a complete connected component of G. A box is *short* if it contains exactly one separator node and *long* if it contains two or more separator nodes. For example, B_1, B_6 are short and B_2, B_3, B_4, B_5 are long in Figure 3.3.

The *contracted clique-separator graph* $\mathcal{G}^c$ is obtained from $\mathcal{G}$ by contracting each box into a single node and replacing multiple arcs by a single arc. We use the same variable to denote a box of $\mathcal{G}$ and the corresponding node of $\mathcal{G}^c$. The Main Theorem will show $\mathcal{G}$ has no semidirected cycle, equivalently, $\mathcal{G}^c$ is a directed acyclic graph. Then $\mathcal{G}^c$ has a topological sort. Given a topological sort σ (of $\mathcal{G}^c$), we assume the boxes of $\mathcal{G}$ are indexed so that $\sigma = (B_1, B_2, \ldots, B_{b(\mathcal{G})})$, where $b(\mathcal{G})$ is the number of boxes of $\mathcal{G}$. Let $\mathcal{G}(\sigma, i)$ be the subgraph of $\mathcal{G}$ induced by $N(B_i) \cup N(B_{i+1}) \cup \cdots \cup N(B_{b(\mathcal{G})})$.

Let S be a separator node of $\mathcal{G}$ and let $Preds(S) = \{S\} \cup \{S' \mid S' \rightsquigarrow S\}$. Let $components_{\mathcal{G}}(S)$ be the number of connected components of $\mathcal{G} - Preds(S)$ containing a clique node that contains S. If nodes $X, Y \in N(\mathcal{G})$, node sets $N_1, N_2 \subset N(\mathcal{G})$, and subgraphs $\mathcal{H}_1, \mathcal{H}_2 \subset \mathcal{G}$ are contained in the same connected component of $\mathcal{G}$ and in different connected components of $\mathcal{G} - Preds(S)$, then S *divides* X, Y, S *divides* N_1, N_2, and S *divides* $\mathcal{H}_1, \mathcal{H}_2$, respectively. We use "separates" and "divides" when referring to G and $\mathcal{G}$, respectively.

A tree T on a set of nodes of $\mathcal{G}$ has the *clique intersection property* if for any two nodes X, Y of T, the set $X \cap Y$ is contained in every node on the X-Y path in T.

3.2.3 The Main Theorem

Theorem 3.2 *Let G be a chordal graph with clique-separator graph $\mathcal{G}$ and topological sort σ. Let S be a separator node of $\mathcal{G}$.*

1. *Every box of $\mathcal{G}$ is a tree with the clique intersection property and $\mathcal{G}$ has no semidirected cycle. The set of separator nodes in a box is an antichain.*

2. *$\mathcal{G} - Preds(S)$ has connected components $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$ and $G - S$ has connected components $G_1, G_2, \ldots, G_k, k > 1$, such that $V(\mathcal{G}_i) - S = V(G_i), 1 \leq i \leq k$.*

3. *For every $1 \leq i < j \leq b(\mathcal{G})$, if S divides nodes X, Y in $\mathcal{G}(\sigma, j)$, then S divides X, Y in $\mathcal{G}(\sigma, i)$.*

4. *For every $1 \leq i < j \leq b(\mathcal{G})$ and every connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, j)$, at most one separator node of box B_i is a predecessor of any node in $\mathcal{H}$.*

5. *For every $1 \leq j \leq b(\mathcal{G})$ and every connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, j)$, $G[V(\mathcal{H})]$ is a connected chordal graph with clique-separator graph $\mathcal{H}$.*

Proof Since the proof applies to each connected component of G, we assume G is connected. We construct $\mathcal{G}$ inductively. If G is a complete graph, then $\mathcal{G}$ is a single clique node and the theorem is true. Otherwise, let S be a minimal separator of G, which is also a minimal vertex separator of G and therefore a clique. Let $V_1, V_2, \ldots, V_k$ be the vertex sets of the connected components of $G - S$, $k > 1$. Let $G_1, G_2, \ldots, G_k$ be the subgraphs of G respectively induced by $V_1 \cup S, V_2 \cup S, \ldots, V_k \cup S$ (Figure 3.4).

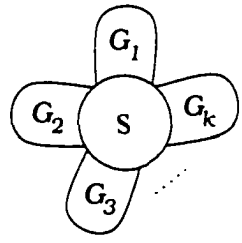


Figure 3.4: G.

Since each of $G_1, G_2, \ldots, G_k$ is connected and chordal, then by induction the theorem holds for each of the corresponding clique-separator graphs $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$. Observe that a separator of any G_i is also a separator of G. Likewise, a minimal vertex separator of any G_i is also a minimal vertex separator of G.

We first show that S is not a maximal clique of any G_i. Otherwise, S is a maximal clique of some G_i and so S is adjacent to some maximal clique K in a clique tree of

G_i, where $\emptyset \neq S \cap K \neq S$. By Theorem 1.1, $S \cap K$ is a minimal vertex separator in G_i and by the observation, $S \cap K$ is a minimal vertex separator in G, contradicting the minimality of S. Thus, S is properly contained in at least one maximal clique of each G_i.

Since S is neither a maximal clique nor a separator of any G_i, then S is not a node in any $\mathcal{G}_i$, which implies the $\mathcal{G}_i$'s are node-disjoint. Now consider $\mathcal{G}_i, \mathcal{G}_j$ for any $i \neq j$. Since a minimal vertex separator of G_i contains at least one vertex of V_i, then $\mathcal{G}$ does not contain an edge $\{S', K'\}$ with S', K' respectively in $\mathcal{G}_i, \mathcal{G}_j$. Similarly, $\mathcal{G}$ does not contain an arc (S', S'') with S', S'' respectively in $\mathcal{G}_i, \mathcal{G}_j$. Therefore, $\mathcal{G}$ has no edge or arc between a node of $\mathcal{G}_i$ and a node of $\mathcal{G}_j$. It follows that we may construct $\mathcal{G}$ from the $\mathcal{G}_i$'s by creating node S and adding edges and arcs between S and nodes in the $\mathcal{G}_i$'s. We do this in two steps, as follows.

Edge Step: For each i such that S is contained in exactly one maximal clique K of G_i, add an edge between S and K and add no arcs from S. (If S is contained in a minimal vertex separator of G_i, then S is contained in at least two maximal cliques of G_i.) Arc Step: For each i such that S is contained in maximal cliques $K_1, K_2, \ldots, K_l$ of $G_i, l > 1$, let T_i be a clique tree of G_i. Let M_i be the set of minimal vertex separators of G_i corresponding to edges on a path in T_i between any two of $K_1, K_2, \ldots, K_l$. By the clique intersection property, S is contained in each element of M_i. Add an arc from S to each minimal element of M_i and add no edges from S. This yields the clique-separator graph $\mathcal{G}$ of G.

Without loss of generality, assume the $\mathcal{G}_i$'s are indexed so that the $\mathcal{G}_i$'s in the Edge Step are $\mathcal{G}_1, \ldots, \mathcal{G}_j$ and the $\mathcal{G}_i$'s in the Arc Step are $\mathcal{G}_{j+1}, \ldots, \mathcal{G}_k$, where $0 \leq j \leq k$ (Figure 3.5). Thus, if $1 \leq i \leq j$, we added exactly one edge from S to a clique node of $\mathcal{G}_i$, and if $j + 1 \leq i \leq k$, we added one or more arcs from S to certain separator nodes of $\mathcal{G}_i$.

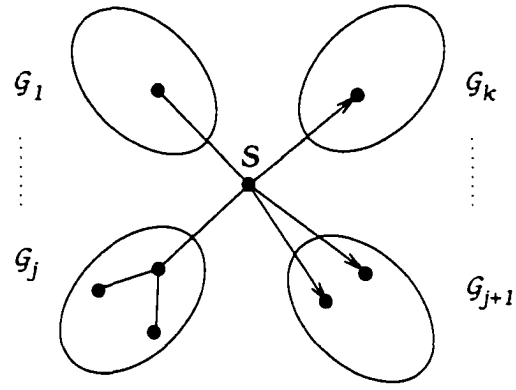


Figure 3.5: $\mathcal{G}$.

1. In the Edge Step, adding an edge between S and one box of each of $\mathcal{G}_1, \ldots, \mathcal{G}_j$ causes these boxes to merge into a box B containing S: since each box is a tree, then B is a tree. (If $j = 0$, then B contains only S.) Since S is contained in each of its neighbors in B, then B has the clique intersection property. By the choice of S, B does not contain separator nodes S', S'' such that $S' \subset S''$. Since every arc added in the Arc Step is an arc from S, then $\mathcal{G}$ has no semidirected cycle.

2. We have defined each G_i to be the subgraph of G induced by $V_i \cup S$, whereas part 2 of the theorem defines each G_i to be the subgraph of G induced by V_i. Thus, S satisfies part 2. Let S' be a separator node of some $\mathcal{G}_i$. By induction, S' satisfies part 2 in $\mathcal{G}_i$. We must show S' satisfies part 2 in $\mathcal{G}$.

Suppose $1 \leq i \leq j$, so that $\mathcal{G}$ has exactly one edge from S to a clique node of $\mathcal{G}_i$. Then whether or not S' is in B, the choice of S implies that S' satisfies part 2 in $\mathcal{G}$. Suppose $j + 1 \leq i \leq k$, so that $\mathcal{G}$ has arcs from S to nodes of $\mathcal{G}_i$ and no edges from S to any node of $\mathcal{G}_i$. Then S' is not in B and whether or not $S \rightsquigarrow S'$, the choice of S implies that S' satisfies part 2 in $\mathcal{G}$.

3. Let $\sigma = (B_1, B_2, \ldots, B_{b(\mathcal{G})})$. Since a separator node of B_1 has no predecessors, it is a minimal separator of G. Therefore, we may assume S is a separator node of B_1 without affecting the proof of parts 1 and 2. (Since $\mathcal{G}$ is connected, then each box of $\mathcal{G}$ has at least one separator node.) Then each of $B_2, \ldots, B_{b(\mathcal{G})}$ is a box of some $\mathcal{G}_i$.

For each $1 \leq i \leq k$, let σ_i be the restriction of σ to $\mathcal{G}_i$. In other words, σ_i is the subsequence of σ that contains every box of $\mathcal{G}_i$; if $1 \leq i \leq j$, then the box of $\mathcal{G}_i$ contained in B_1 is identified with B_1. Then σ_i is a topological sort of $\mathcal{G}_i^{\rightarrow}$. Observe that if $\mathcal{H}$ is a connected component of $\mathcal{G}(\sigma, j')$, $j' > 1$, then $\mathcal{H}$ is a subgraph of some $\mathcal{G}_i$ and thus $\mathcal{H}$ is a connected component of $\mathcal{G}_i(\sigma_i, j')$.

We next show that if S' divides nodes X, Y in $\mathcal{G}(\sigma, j')$, then S' divides X, Y in $\mathcal{G}(\sigma, i')$, $1 \leq i' < j' \leq b(\mathcal{G})$. By the observation, S' divides X, Y in $\mathcal{G}_i(\sigma_i, j')$ for some i. By induction, S' divides X, Y in $\mathcal{G}_i(\sigma_i, i')$. If $i' > 1$, then every connected component of $\mathcal{G}_i(\sigma_i, i')$ is a connected component of $\mathcal{G}(\sigma, i')$ and thus S' divides X, Y in $\mathcal{G}(\sigma, i')$. Suppose $i' = 1$. Then S' divides X, Y in $\mathcal{G}_i$ and we must show S' divides X, Y in $\mathcal{G}$.

If $1 \leq i \leq j$, then the construction of $\mathcal{G}$ implies that S' divides X, Y in $\mathcal{G}$. If $j + 1 \leq i \leq k$, then suppose $\mathcal{G}$ has arcs from S to nodes S_1, S_2 in different connected components of $\mathcal{G}_i - Preds(S')$. (If $\mathcal{G}$ has no such arcs, then S' divides X, Y in $\mathcal{G}$.) Then $S \subset S_1, S_2$. If there is a vertex $v \in S - S'$, then $v \in S_1 - S'$ and $v \in S_2 - S'$. But by part 2, S' separates $S_1 - S'$ and $S_2 - S'$ in G_i, a contradiction. Thus, $S \subset S'$. By the construction of $\mathcal{G}$, $S \rightsquigarrow S'$ in $\mathcal{G}$ and thus S is a predecessor of S'. Then every connected component of $\mathcal{G}_i - Preds(S')$ is a connected component of $\mathcal{G} - Preds(S')$, which means S' divides X, Y in $\mathcal{G}$.

4. Let $\mathcal{H}$ be a connected component of $\mathcal{G}(\sigma, j')$, $1 < j' \leq b(\mathcal{G})$. Since $\mathcal{H}$ is a subgraph of some $\mathcal{G}_i$, then $\mathcal{H}$ is a connected component of $\mathcal{G}_i(\sigma_i, j')$. By induction, each box of $\mathcal{G}_i$ contains at most one predecessor of a node in $\mathcal{H}$. Consider the boxes of $\mathcal{G}$. By the construction of $\mathcal{G}$, B_1 contains at most one predecessor of a node in $\mathcal{H}$. For every box $B_{i'}$, $1 \leq i' < j'$, of $\mathcal{G}$. If $B_{i'}$ is a box of $\mathcal{G}_i$, then $B_{i'}$ contains at most one predecessor of a node in $\mathcal{H}$, and otherwise, $B_{i'}$ contains no predecessor of a node in $\mathcal{H}$.

5. We show that for every connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, j')$, $G[V(\mathcal{H})]$ is a connected chordal graph with clique-separator graph $\mathcal{H}$, $1 \leq j' \leq b(\mathcal{G})$. If $j' = 1$,

then $\mathcal{G}(\sigma, j') = \mathcal{G}$ and $G[V(\mathcal{G})] = G$. Suppose $l > 1$ and let $\mathcal{H}$ be a connected component of $\mathcal{G}(\sigma, j')$. By the observation, $\mathcal{H}$ is a connected component of $\mathcal{G}_i(\sigma_i, j')$. By induction, $G_i[V(\mathcal{H})]$ is a connected chordal graph with clique-separator graph $\mathcal{H}$. Since $V(\mathcal{H}) \subseteq V(\mathcal{G}_i) = V(G_i)$, then by the choice of S, $G[V(\mathcal{H})] = G_i[V(\mathcal{H})]$. Hence, $G[V(\mathcal{H})]$ is a connected chordal graph with clique-separator graph $\mathcal{H}$. □

In general, it is not true that $\mathcal{G}(\sigma, j)$ is the clique-separator graph of the chordal graph $G[V(\mathcal{G}(\sigma, j))]$. For example, Figure 3.6 shows a chordal graph G and its clique-separator graph $\mathcal{G}$ where $\mathcal{G}(\sigma, 2)$ is not the clique-separator graph of $G[V(\mathcal{G}(\sigma, 2))] = G$.

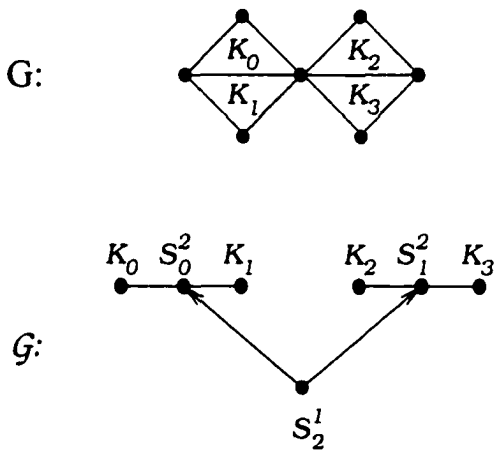


Figure 3.6: $\mathcal{G}(\sigma, 2)$ is not the clique-separator graph of $G[V(\mathcal{G}(\sigma, 2))]$.

Corollary 3.3 *Let G be a chordal graph with clique-separator graph $\mathcal{G}$ and topological sort σ. Let S be a separator node of $\mathcal{G}$.*

1. *If S divides $X, Y \in N(\mathcal{G})$, then S separates $X - S$ and $Y - S$.*

   If S divides $N_1, N_2 \subset N(\mathcal{G})$, then S separates $V(N_1) - S$ and $V(N_2) - S$.

   If S divides $\mathcal{H}_1, \mathcal{H}_2 \subset \mathcal{G}$, then S separates $V(\mathcal{H}_1) - S$ and $V(\mathcal{H}_2) - S$.

2. *components$_{\mathcal{G}}(S) \geq 2$.*

3. *If S divides $X, Y \in N(\mathcal{G})$ and $S \subseteq X, Y$, then $S = X \cap Y$ and S is a minimal uv-separator for any $u \in X - S$ and $v \in Y - S$.*

4. *Let box B_i contain S. If $B_i - S$ has subtrees T, T' and S has a successor in connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, i+1)$, then S divides T, T' and S divides $T, \mathcal{H}$. If B_i also contains separator node $S' \neq S$ that has a successor in connected component $\mathcal{H}'$ of $\mathcal{G}(\sigma, i+1)$ and S'' is any separator node on the S-S' subpath in B_i, then S'' divides $\mathcal{H}, \mathcal{H}'$.*

Proof

1. By the Main Theorem-2.

2. Although this follows from the Main Theorem's proof, we prove it as a corollary. By Theorem 1.1, for some edge $\{K, K'\}$ in a clique tree of G, $S = K \cap K'$ and S separates $K - S$ and $K' - S$. By the Main Theorem-2, K and K' are in different connected components of $\mathcal{G} - Preds(S)$, which means $components_{\mathcal{G}}(S) \geq 2$. (This implies $degree(S) + outdegree(S) \geq 2$.)

3. We have $S \subseteq X \cap Y$. Since S divides X and Y, then by part 1, S separates $X - S$ and $Y - S$, which implies $S = X \cap Y$. Since every vertex in S is adjacent to every vertex in $X - S$ and $Y - S$, then S is a minimal uv-separator for any $u \in X - S$ and $v \in Y - S$.

4. By the Main Theorem-4, S divides T, T' in $\mathcal{G}(\sigma, i)$ and S divides $T, \mathcal{H}$ in $\mathcal{G}(\sigma, i)$. If $i > 1$, then the Main Theorem-3 implies that S divides T, T' in $\mathcal{G}$ and S divides $T, \mathcal{H}$ in $\mathcal{G}$. (Thus, if S has distinct neighbors X, Y or neighbor X and successor Y, then S divides X, Y.) The same argument shows that S'' divides $\mathcal{H}, \mathcal{H}'$. □

3.3 The clique-separator graph of an interval graph

Recall that interval graphs are a subclass of chordal graphs and that G is interval if and only if G has a clique path, which is a path on the clique nodes of $\mathcal{G}$ with the clique intersection property. We will decide whether G is interval by deciding whether G has a clique path. Therefore, in this section and the next two sections, we consider paths of nodes, which we represent as sequences of nodes.

Let G be a chordal graph with clique-separator graph $\mathcal{G}$ and let P, Q be paths of nodes of $\mathcal{G}$. Let $N(P)$ denote the set of nodes of P and for $X, Y \in N(P)$, let $P(X, Y)$ denote the X-Y subpath of P. We say P is a *subsequence* of Q if (sequence) P or its reverse is a subsequence of (sequence) Q, and P is *c.i.p.* if P has the clique intersection property. Thus, if P is a subsequence of Q and Q is c.i.p., then P is c.i.p.. We say P is *constraining* if P is the unique c.i.p. path on $N(P)$, equivalently, if P is a subsequence of every c.i.p. path containing the nodes in $N(P)$.

3.3.1 Relevance of $\mathcal{G}$

We will consider c.i.p. paths on all separator nodes and certain clique nodes of $\mathcal{G}$, as follows. A node is *relevant* if it is a separator node or an internal clique node, equivalently, if it is not a leaf clique node. A *relevant path* of $\mathcal{H} \subseteq \mathcal{G}$ is a c.i.p. path on the relevant nodes of $\mathcal{H}$.

Lemma 3.4 *Let G be a chordal graph with clique-separator graph $\mathcal{G}$. G has a clique path if and only if $\mathcal{G}$ has a relevant path.*

Proof

($\Rightarrow$) Let $P = (K_1, K_2, \ldots, K_j)$ be a clique path of G. Let $P' = (K_1, S_1, K_2, \ldots, K_{j-1}, S_{j-1}, K_j)$, where $S_i = K_i \cap K_{i+1}, 1 \leq i < j$. Then P' is a c.i.p. path and by Theorem 1.1.1, P' is a path containing every node of $\mathcal{G}$, possibly with duplicate separator

nodes. Deleting every duplicate separator node from P' yields a c.i.p. path P'' containing every node of $\mathcal{G}$ exactly once. Deleting every leaf clique node from P'' yields a c.i.p. path on the relevant nodes of $\mathcal{G}$.

($\Leftarrow$) Let P be a relevant path of $\mathcal{G}$. For every separator node S, let $\mathcal{K}_S$ be the set of leaf clique nodes adjacent to S. Let P' be the path obtained from P by applying the following operation to every separator node S with $\mathcal{K}_S \neq \emptyset$: replace S with any path on $\mathcal{K}_S$. Then P' contains every clique node of $\mathcal{G}$ and by Lemma 3.1.4, a clique node $K \in \mathcal{K}_S$ is the only maximal clique of G containing any vertex in $K - S$, which implies P' is a c.i.p. path. Deleting every remaining separator node (with $\mathcal{K}_S = \emptyset$) from P' yields a c.i.p. path P'' on the clique nodes of $\mathcal{G}$, i.e., a clique path of G. Figure 3.7 shows a clique-separator graph $\mathcal{G}$ with one box, where $P = (S_1, K_3, S_2, K_4, S_3)$, $P' = (K_1, K_2, K_3, S_2, K_4, K_5)$, and $P'' = (K_1, K_2, K_3, K_4, K_5)$. □

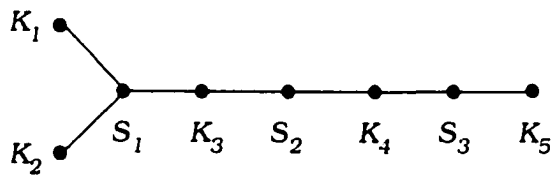


Figure 3.7: A box.

3.3.2 Structure and size of $\mathcal{G}$

Lemma 3.5 *Let G be an interval graph with clique-separator graph $\mathcal{G}$.*

1. *If X is a clique node or separator node, then the predecessors of X can be partitioned into at most two chains.*

2. *Every separator node of $\mathcal{G}$ has at most two immediate predecessors. Every clique node of $\mathcal{G}$ has at most two neighbors.*

3. *$\mathcal{G}$ has $O(n)$ nodes, edges and arcs.*

4. *For every clique tree T and maximal clique K of G, $\mathcal{S}_T(K)$ can be partitioned into at most two chains.*

Proof

1. By the proof of Lemma 3.4, there is a c.i.p. path on the nodes of $\mathcal{G}$. Let $\mathcal{S}$ be the set of predecessors of X. Suppose $\{S_1, S_2, S_3\} \subset \mathcal{S}$ is an antichain. Since $S_1, S_2 \subset X$, then $P = (S_1, X, S_2)$ is a constraining path. Since $S_3 \subset X$, then no c.i.p. path contains S_3 and $N(P)$, a contradiction. Thus, $(\mathcal{S}, \subseteq)$ is a partially ordered set whose largest antichain has size at most 2. By Dilworth's Theorem, which states that the size of the largest antichain is equal to the size of the smallest chain decomposition, $\mathcal{S}$ can be partitioned into at most two chains.

2. By part 1.

3. Since G has at most n maximal cliques and at most $n-1$ minimal vertex separators, then $\mathcal{G}$ has at most $2n-1$ nodes. The Main Theorem-1 implies that $\mathcal{G}$ has at most $2n-2$ edges and part 2 implies that $\mathcal{G}$ has at most $2n-2$ arcs.

4. Since every minimal vertex separator of $\mathcal{S}_T(K)$ is a predecessor of K in $\mathcal{G}$, then by part 1, $\mathcal{S}_T(K)$ can be partitioned into at most two chains. □

Lemma 3.6 *Let G be an interval graph with clique-separator graph $\mathcal{G}$. Let B be a box of $\mathcal{G}$.*

1. *Every path between two separator nodes of B is a constraining path.*

2. *Every separator node of B has at most two neighbors that are internal nodes of B.*

Proof By the Main Theorem-1, the set of separator nodes of a box is an antichain. We claim that if a clique node K of B contains a separator node S of B, then K is

a neighbor of S. Otherwise, let $S' \neq S$ be K's neighbor on the K–S path in B. By the clique intersection property, $S \subset S'$, a contradiction.

1. Let $(S_1, K_1, S_2, K_2, S_3)$ be a path contained in box B. Then $\{S_1, S_2, S_3\}$ is an antichain. Since $S_1, S_2 \subset K_1$, then (S_1, K_1, S_2) is a constraining path. By the claim, $S_1 \not\subset K_2$. It follows that $(S_1, K_1, S_2, K_2, S_3)$ is a constraining path. A similar induction argument shows that every path between two separator nodes of B is a constraining path.

2. Suppose a separator node S in box B has three internal neighbors (Figure 3.8). Then B contains paths $(S, K_1, S_1), (S, K_2, S_2), (S, K_3, S_3)$ and $\{S, S_1, S_2, S_3\}$ is an antichain. By part 1, (S_1, K_1, S, K_2, S_2) is a constraining path and by the claim, $P = (S_1, K_1, K_3, K_2, S_2)$ is a constraining path. Since $S_3 \subset K_3$, then no c.i.p. path contains S_3 and $N(P)$, a contradiction. □

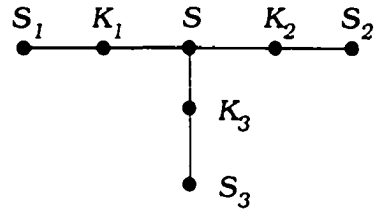


Figure 3.8: A separator node S with three internal neighbors.

In light of Lemma 3.4, we will focus on the relevant nodes of $\mathcal{G}$. The *body* of box B, denoted *body*(B), is the subgraph of B induced by the set of its relevant nodes. Equivalently, deleting every leaf clique node of B yields *body*(B). A *caterpillar* is a tree T such that deleting every leaf of T yields a path, equivalently, T has at least three nodes and every internal node of T has at most two neighbors that are internal nodes of T. By Lemmas 3.5.2 and 3.6, we have the following.

Corollary 3.7 *Let G be an interval graph with clique-separator graph $\mathcal{G}$. Every box B of $\mathcal{G}$ with three or more nodes is a caterpillar and body(B) is a constraining path whose endpoints are separator nodes.*

Let G be an interval graph with clique-separator graph $\mathcal{G}$. The *outer nodes* of box B are the endpoints of *body*(B) and the *inner nodes* of B are the remaining separator nodes of B. The next lemma is used by the update algorithm and its proof illustrates the proof technique that will be used to prove the Ordering Lemma on page 61.

Lemma 3.8 *Let G be an interval graph with clique-separator graph $\mathcal{G}$. Let B be a long box of $\mathcal{G}$ with outer nodes S_1, S_2.*

1. *If S is a predecessor of a node of B, then S is a predecessor of S_1 or S_2.*

2. *If S, S' are incomparable predecessors of nodes of B, then S is a predecessor of S_1 and S' is a predecessor of S_2, or vice versa.*

Proof Let P be a relevant path of $\mathcal{G}$. By Corollary 3.7, *body*(B) is a subsequence of P.

1. If S is on $P(S_1, S_2)$, then some separator node of *body*(B) is contained in S, which implies $\mathcal{G}$ has a semidirected cycle, contradicting the Main Theorem-1. Thus, S is on P and not on $P(S_1, S_2)$. Since S is contained in at least one node of $P(S_1, S_2)$, then $S \subset S_1$ or $S \subset S_2$.

2. By the proof of part 1, each of S, S' is on P and not on $P(S_1, S_2)$. If (S, S_1, S_2, S') or (S', S_1, S_2, S) is a subsequence of P, then $S \subset S_1, S' \subset S_2$ or $S' \subset S_1, S \subset S_2$. If (S, S', S_1, S_2) is a subsequence of P, then $S \subset S'$, a contradiction, and the other cases likewise lead to contradictions. □

3.4 The train tree

3.4.1 PQ-trees

This subsection reviews PQ-trees, which were invented by Booth and Lueker [BL76] for their linear time algorithms to recognize interval and planar graphs. Their algorithms incrementally construct the PQ-tree using a complicated set of templates. In contrast, we use only the definition and none of the templates of PQ-trees.

A PQ-tree represents a set of permutations of a set U. A *PQ-tree* is a rooted tree T whose leaves are the elements of U and each internal node of T is either a *P-node* or a *Q-node*. Two PQ-trees are *equivalent* ($\equiv$) if one can be transformed into the other by any combination of the following operations: (1) permute the children of a P-node, (2) reverse the order of the children of a Q-node. Let $sequence(T)$ be the sequence of leaves of T. Then T represents the set of permutations $\{sequence(T') \mid T' \equiv T\}$. Figure 3.9 shows a PQ-tree that represents 8 permutations of $\{1, 2, 3, 4\}$. P-nodes are drawn as circles and Q-nodes as rectangles.

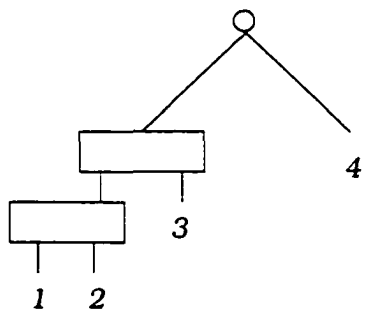


Figure 3.9: A PQ-tree.

3.4.2 Train trees

Let G be a chordal graph with clique-separator graph $\mathcal{G}$ and let T be a PQ-tree whose leaves are relevant nodes of $\mathcal{G}$. We usually use lower case variables for the nodes of T and for clarity, we use $V(l)$ to denote the set of vertices contained in leaf l of T.

Given a node x of T, let *sequence*(x) be the sequence of leaves of *subtree*$_T(x)$ and let *leaves*(x) be the set of leaves of *subtree*$_T(x)$, where *subtree*$_T(x)$ is the subtree of T rooted at x. We view *sequence*(x) as a path on *leaves*(x), so that x represents a set of paths on *leaves*(x). If a separator node S is contained in every endpoint of *sequence*(x), then S *meets* x, and otherwise, if S is contained in exactly one endpoint of *sequence*(x), then S *semi-meets* x. Define *sequence*(T), *leaves*(T) similarly. If S meets *root*(T), then S *meets* T, and if S semi-meets *root*(T), then S *semi-meets* T.

A *train* is a sequence $l_1, l_2, \ldots, l_k$ of nodes of $\mathcal{G}$ such that for each $1 \leq j < k$, either $V(l_j) \subset V(l_{j+1})$ or $V(l_{j+1}) \subset V(l_j)$, which implies l_j or l_{j+1} is a separator node, respectively. For example, the body of a box is a train.

A *train tree of* $\mathcal{H} \subseteq \mathcal{G}$ is a PQ-tree T with the following six properties:

1. *leaves*(T) is the set of relevant nodes of $\mathcal{H}$.

2. Every internal node x represents all c.i.p. paths on *leaves*(x).

3. Every P-node has at least two children.

   Every Q-node has at least three children.

4. Every P-node x has a *meeting child* S_x, which is a separator node that meets x and satisfies the following.

   a. S_x divides *leaves*$(x) - \{S_x\}$ and *leaves*$(T) -$ *leaves*(x) in $\mathcal{H}$.

   b. S_x divides *leaves*(y) and *leaves*(y') in $\mathcal{H}$ for any children $y, y' \neq S_x$ of x.

An *S-node* is a P-node or a separator node. Each S-node x is closely associated with a separator node S_x: if x is a P-node, then S_x has been defined, and if x is a separator node S, then S_x is S.

5. Every Q-node x with children $(y_1, y_2, \ldots, y_k)$ satisfies the following.

   a. If any y_i is a Q-node, then $1 < i < k$ and y_{i-1}, y_{i+1} are S-nodes such that the *meeting siblings* $S_{y_{i-1}}, S_{y_{i+1}}$ of y_i are incomparable and meet y_i.

   b. If any y_i is an S-node, then S_{y_i} divides $leaves(y_1) \cup \cdots \cup leaves(y_{i-1})$ and $leaves(y_{i+1}) \cup \cdots \cup leaves(y_k)$ in $\mathcal{H}$.

   c. y_1, y_k are S-nodes and no separator node in $leaves(x)$ is a predecessor of S_{y_1} or S_{y_k} in $\mathcal{H}$.

Let x be a Q-node. Given a contiguous sequence α of children of x where each child is a P-node or a leaf, replacing each P-node y with S_y yields an *offspring block* α' of x; each leaf in α' is an *offspring* of x. Equivalently, an offspring of x is a leaf that is a child of x or the meeting child of a P-node child of x. (For example, $(S_0^1, K_0, S_1^1, K_1, S_2^1)$ is an offspring block of $root(T)$ in Figure 3.10.)

6. Every offspring block of a Q-node is a train. The body of every box contained in $\mathcal{H}$ is an offspring block of some Q-node.

Figure 3.10 shows an interval graph G, its clique-separator graph $\mathcal{G}$, and a train tree T of $\mathcal{G}$. The meeting child of a P-node is drawn as a triangle and if $X \rightsquigarrow Y$, then X is drawn below Y in the clique-separator graph and above Y in the train tree. A train tree is *trivial* if it has exactly one leaf.

Let T be a train tree of $\mathcal{H} \subseteq \mathcal{G}$ with internal node x and let S be any separator node of $\mathcal{G}$. The train tree definition immediately implies the following.

- T represents all relevant paths of $\mathcal{H}$.
- S meets x if and only if S is contained in every leaf in $leaves(x)$.
- If x is a P-node, then S_x is the only minimal node of $leaves(x)$. Thus, if $S \rightsquigarrow S_x$, then $S \not\rightarrow S'$ for any $S' \in leaves(x) - \{S_x\}$.

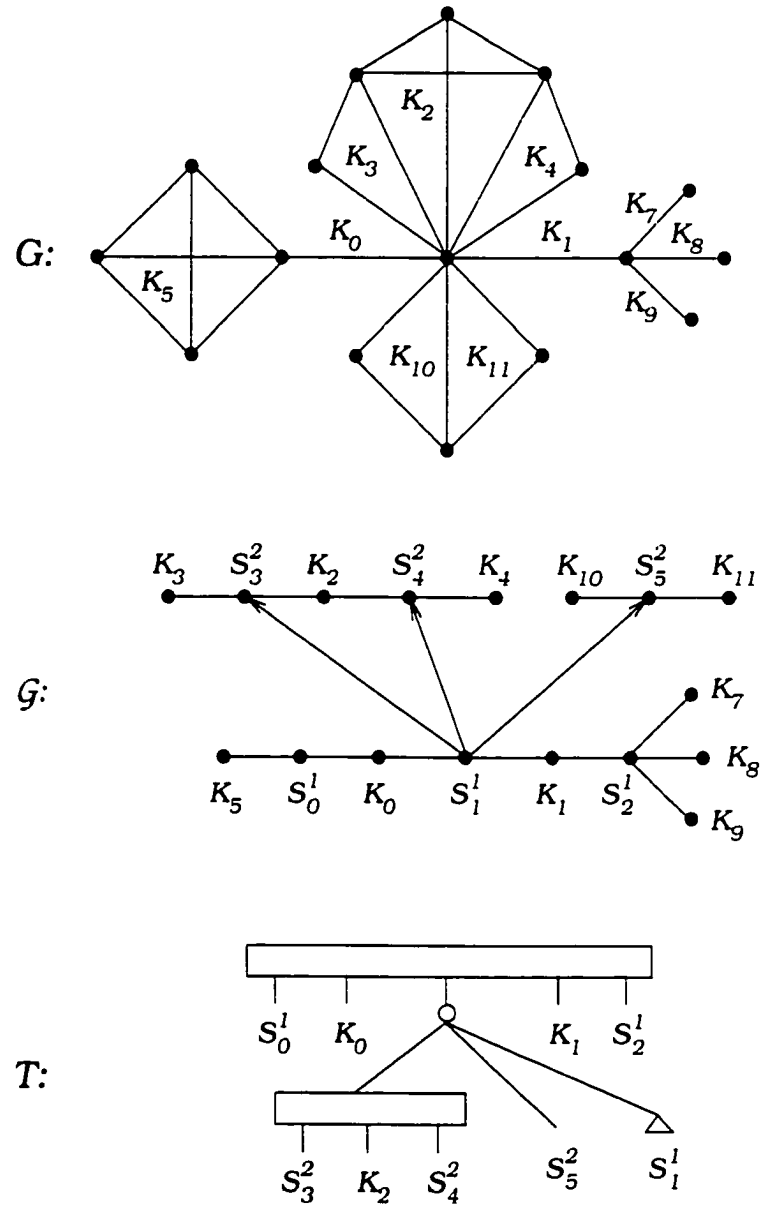


Figure 3.10: An interval graph, its clique-separator graph, and its train tree.

- If x is a Q-node with meeting siblings S_{y_1}, S_{y_2}, then S_{y_1}, S_{y_2} are the only minimal nodes of $leaves(x) \cup \{S_{y_1}, S_{y_2}\}$. Thus, if $S \rightsquigarrow S_{y_1}$ or $S \rightsquigarrow S_{y_2}$, then $S \not\rightarrow S'$ for any $S' \in leaves(x)$.

- If x is a Q-node with leftmost and rightmost children z_1, z_2, then S_{z_1}, S_{z_2} are minimal nodes of $leaves(x)$. Furthermore, every minimal node of $leaves(x)$ is a separator node offspring of x.

3.4.3 Properties of train trees

In the rest of this section, we present lemmas that will imply the correctness of the train tree algorithm and the update algorithm, as well as a lemma showing that $\mathcal{G}$ has a train tree only if G is connected. The first lemma shows that an internal node x encapsulates $leaves(x)$ in a particular way.

Lemma 3.9 *Let G be an interval graph with clique-separator graph $\mathcal{G}$ and train tree T of $\mathcal{G}$.*

1. *If x is a P-node, then there is no arc in $\mathcal{G}$ from a leaf in $leaves(x)$ to a leaf in $leaves(T) - leaves(x)$ and vice versa, except for arcs from and to S_x.*

2. *If x is a Q-node with parent y, then there is no arc in $\mathcal{G}$ from a leaf in $leaves(x)$ to a leaf in $leaves(T) - leaves(x)$ and vice versa, except for arcs from the meeting child of y (if y is a P-node) or from a meeting sibling of x (if y is a Q-node) to leaves in $leaves(x)$.*

Proof

1. By train tree property 4a.

2. By induction on the height of T. Let r be the root of T. Suppose r is a P-node (Figure 3.11a). Let $y, y' \neq S_r$ be any children of r. By induction, the claim holds for

$subtree_T(y)$. By train tree property 4b, there is no arc from a leaf in $leaves(y)$ to a leaf in $leaves(y')$ and there is no arc from $leaves(y)$ to S_r. Thus, the claim holds for T. Suppose r is a Q-node (Figure 3.11b). Let y be any child of r. By induction, the claim holds for $subtree_T(y)$. If y is a P-node, then by part 1, there is no arc from $leaves(y) - \{S_y\}$ to any leaf in $leaves(T) - leaves(y)$ and vice versa. If y is a Q-node with meeting siblings S_{y_1}, S_{y_2}, then by train tree property 5a and 5b, there is no arc from $leaves(y)$ to any leaf in $leaves(T) - leaves(y)$ and no arc from any leaf in $leaves(T) - (leaves(y) \cup \{S_{y_1}, S_{y_2}\})$ to any leaf in $leaves(y)$. Thus, the claim holds for T. □

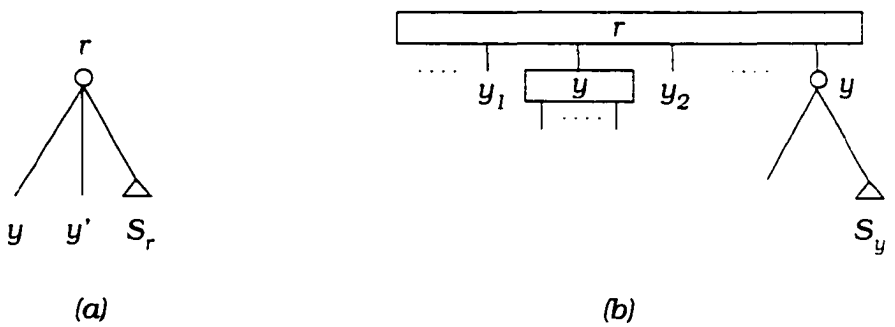


Figure 3.11: Proof of Lemma 3.9.

In the next lemma, part 1 specifies where a separator node's successors appear in the train tree, and parts 2 and 3 specify where its immediate successors appear in the train tree.

Lemma 3.10 *Let G be an interval graph with clique-separator graph $\mathcal{G}$ and train tree T of $\mathcal{G}$. Let S be a separator node of $\mathcal{G}$.*

1. *If S is a predecessor of S', then the following holds.*

   If S is an offspring of Q-node x, then $S' \in leaves(x)$. If S is not an offspring of any Q-node, then S is the meeting child of a P-node x and $S' \in leaves(x)$.

2. *If S is an immediate predecessor of S', then at least one of the following holds.*

   (a) S is the meeting child of a P-node x and $S' \in leaves(x)$.

   (b) S is a meeting sibling of a Q-node y and $S' \in leaves(y)$.

   (c) S and S' are offspring of the same Q-node.

3. *If S is the meeting child of a P-node x, then S is an immediate predecessor of the minimal nodes of $leaves(y)$ for every child $y \neq S$ of x.*

   If S is a meeting sibling of a Q-node y, then S is an immediate predecessor of the minimal nodes of $leaves(y)$.

   If S is an offspring of a Q-node z and $\mathcal{S}$ is the set of offspring of z properly containing S, then S is an immediate predecessor of the minimal nodes of $\mathcal{S}$.

Proof

1. If S is an offspring of some Q-node x, then by Lemma 3.9.2, $S' \in leaves(x)$. Otherwise, S is a child of a P-node x such that either x is the root of T or x has a P-node parent y. In the first case, $S' \in leaves(x)$. In the second case (Figure 3.12), train tree property 4a and 4b applied to y implies $S' \in leaves(x)$. If $S \neq S_x$, then train tree property 4a and 4b applied to x implies S has no successors, a contradiction. Thus, $S = S_x$.

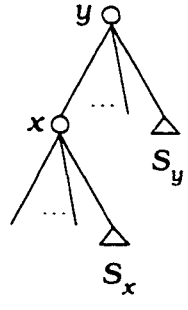


Figure 3.12: Proof of Lemma 3.10.

2. If S is not an offspring of any Q-node, then by part 1, S' satisfies (a). Otherwise, S is an offspring of some Q-node x and by part 1, $S' \in leaves(x)$. Let y be a child of

x. If y is a P-node and $S \neq S_y$, then by Lemma 3.9.1, S has no immediate successor in $leaves(y) - \{S_y\}$. If y is a Q-node and S is not a meeting sibling of y, then by Lemma 3.9.2, S has no immediate successor in $leaves(y)$. Thus, if S' doesn't satisfy (a) or (b), then S' satisfies (c).

3. Suppose S is the meeting child of a P-node x with child $y \neq S$. Then S is a predecessor of every leaf in $leaves(y) - \{S\}$. By Lemma 3.9.1, no leaf in $leaves(T) - leaves(y)$ is a predecessor of a leaf in $leaves(y) - \{S\}$, which implies S is an immediate predecessor of the minimal nodes of $leaves(y)$.

Suppose S is a meeting sibling of a Q-node y. Then S is a predecessor of every leaf in $leaves(y)$. Let $S' \neq S$ be the other meeting sibling of y. By Lemma 3.9.2, no leaf in $leaves(T) - (leaves(y) \cup \{S, S'\})$ is a predecessor of a leaf in $leaves(y)$. Since S and S' are incomparable, then S and S' are immediate predecessors of the minimal nodes of $leaves(y)$.

Suppose S is an offspring of a Q-node z and $\mathcal{S}$ is the set of offspring of z properly containing S. Then S is a predecessor of every leaf in $\mathcal{S}$. Suppose $S \rightarrow S'$ and S' is a predecessor of an offspring of z. By part 1, $S' \in leaves(z)$. By Lemma 3.9, S' is an offspring of z. It follows that S is an immediate predecessor of the minimal nodes of $\mathcal{S}$. □

Next is a technical lemma that is used only to prove the subsequent corollary.

Lemma 3.11 *Let G be an interval graph with clique-separator graph $\mathcal{G}$ and train tree T of $\mathcal{G}$. Let $l_1, l_2 \in leaves(T)$ and let x be the lowest common ancestor of l_1, l_2 with children y_1, y_2 such that $l_1 \in leaves(y_1), l_2 \in leaves(y_2)$.*

1. *If x is a P-node, then $S_x = V(l_1) \cap V(l_2)$.*

2. *If x is a Q-node and y_1, y_2 are consecutive children of x, then there is a separator node offspring S of x such that $S = V(l_1) \cap V(l_2)$.*

If x is a Q-node and y_1, y_2 are not consecutive children of x, then either (a) there is a separator node offspring S of x such that $V(l_1) \cap V(l_2) \subseteq S$ and ($S \subset V(l_1)$ or $S \subset V(l_2)$), or (b) l_1, l_2 are offspring of x.

Proof

1. If S_x is l_1, then $S_x = V(l_1) \subset V(l_2)$ and so $S_x = V(l_1) \cap V(l_2)$. Likewise, if S_x is l_2, then $S_x = V(l_1) \cap V(l_2)$. Suppose S_x is neither of l_1, l_2. Since S_x meets x, then $S_x \subseteq V(l_1) \cap V(l_2)$. By train tree property 4b, S_x divides l_1 and l_2 and so S_x separates $V(l_1) - S_x$ and $V(l_2) - S_x$, which implies $V(l_1) \cap V(l_2) \subseteq S_x$. Thus, $S_x = V(l_1) \cap V(l_2)$.

2. Define leaf X_1 as follows. If y_1 is leaf l_1, then X_1 is l_1. If y_1 is a P-node, then X_1 is S_{y_1}. If y_1 is a Q-node, then X_1 is the meeting sibling of y_1 that is closer to y_2: X_1 is S_{y_2} if y_1, y_2 are consecutive. Define X_2 similarly.

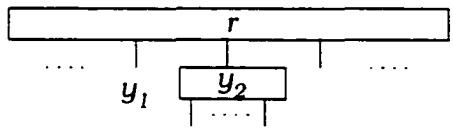


Figure 3.13: Proof of Lemma 3.11.

Suppose y_1, y_2 are consecutive children of x. By train tree property 5a, y_1, y_2 are not both Q-nodes. Case 1: One of y_1, y_2 is a Q-node, say y_2. (Figure 3.13.) Then y_1 is an S-node and $S_{y_1} = X_1$ is a meeting sibling of y_2. If y_1 is leaf l_1, then $S_{y_1} = V(l_1) = V(l_1) \cap V(l_2)$. Suppose y_1 is a P-node. If $S_{y_1} = l_1$, then $S_{y_1} = V(l_1) = V(l_1) \cap V(l_2)$. Otherwise, since S_{y_1} meets y_1 and y_2, then $S_{y_1} \subseteq V(l_1) \cap V(l_2)$. Since S_{y_1} divides l_1 and l_2, then S_{y_1} separates $V(l_1) - S_{y_1}$ and $V(l_2) - S_{y_1}$, which implies $V(l_1) \cap V(l_2) \subseteq S_{y_1}$. Thus, $S_{y_1} = V(l_1) \cap V(l_2)$. Case 2: Neither of y_1, y_2 is a Q-node. Then $X_1 \neq X_2$ and X_1, X_2 are consecutive leaves in an offspring block of x, which is a train by train tree property 6. Then $X_1 \subset X_2$ or $X_2 \subset X_1$, say $X_1 \subset X_2$. Then X_1 is a separator node and y_1 is an S-node with $S_{y_1} = X_1$. If y_1 is leaf l_1, then $S_{y_1} = V(l_1) = V(l_1) \cap V(l_2)$. If y_1 is a P-node, then as before, $S_{y_1} = V(l_1) \cap V(l_2)$.

Suppose y_1, y_2 are not consecutive children of x. Case 1: One of y_1, y_2 is a Q-node, say y_2. Then y_2 has meeting sibling X_2, which is a separator node such that $X_2 \subset V(l_2)$. By train tree property 5b, X_2 divides l_1 and l_2 and so X_2 separates $V(l_1) - X_2$ and $V(l_2) - X_2$. Thus, $V(l_1) \cap V(l_2) \subseteq X_2$. Case 2: Neither of y_1, y_2 is a Q-node. If l_1, l_2 are offspring of x, then we are done, so suppose at least one of l_1, l_2 is not an offspring of x, say l_1. Then y_1 is not a leaf, implying y_1 is a P-node with meeting child $X_1 \neq l_1$. By train tree property 4a, X_1 divides l_1, l_2 and so X_1 separates $V(l_1) - X_1$ and $V(l_2) - X_1$. Thus, $V(l_1) \cap V(l_2) \subseteq X_2$. □

Corollary 3.12 *Let G be an interval graph with clique-separator graph $\mathcal{G}$ and train tree T of $\mathcal{G}$.*

1. *If $l_1, l_2 \in leaves(T)$ are consecutive in sequence(T), then there is a separator node $S \in leaves(T)$ such that $S = V(l_1) \cap V(l_2)$.*

2. *Let C be any clique of G. If separator nodes S_1, S_2 are minimal nodes of $leaves(T)$ such that $C \subseteq S_1$ and $C \subseteq S_2$, then S_1, S_2 are offspring of the same Q-node of T.*

3. *G is connected.*

Proof

1. By Lemma 3.11.

2. Let x be the lowest common ancestor of S_1, S_2. Suppose x is a P-node. By Lemma 3.11.1, $S_x = S_1 \cap S_2$ and so $C \subseteq S_x$. Now $S_x \neq S_1$ or $S_x \neq S_2$, say $S_x \neq S_1$. Then $S_x \subset S_1$ and thus S_1 is not minimal, a contradiction. Hence, x is a Q-node. Let y_1, y_2 be the children of x such that $S_1 \in leaves(y_1), S_2 \in leaves(y_2)$. If y_1, y_2 are consecutive children of x, then by Lemma 3.11.2, for some separator node $S \in leaves(x)$, $S = S_1 \cap S_2$ and so $C \subseteq S$. Again, since $S \neq S_1$ or $S \neq S_2$, then S_1 or S_2 is not minimal, a contradiction. Hence, y_1, y_2 are not consecutive children of

x. By Lemma 3.11.2, either (a) for some separator node $S \in leaves(x)$, $S_1 \cap S_2 \subseteq S$ and ($S \subset S_1$ or $S \subset S_2$), or (b) S_1, S_2 are offspring of x. In (a), $C \subseteq S$ and if $S \subset S_1$ (resp. $S \subset S_2$), then S_1 (resp. S_2) is not minimal, a contradiction. Hence, we must have (b).

3. If G has two connected components, then $\mathcal{G}$ has two connected components $\mathcal{G}_1, \mathcal{G}_2$ and $sequence(T)$ has consecutive leaves l_1, l_2 that are respectively contained in $\mathcal{G}_1, \mathcal{G}_2$. By part 1, $S = V(l_1) \cap V(l_2)$ for some separator node S of $\mathcal{G}$. But since $V(l_1), V(l_2)$ are sets of vertices in different connected components of G, then $V(l_1) \cap V(l_2) = \emptyset$, a contradiction. □

3.4.4 Preview of the train tree algorithm

The preceding lemmas have assumed G is an interval graph. Now let G be a chordal graph with clique-separator graph $\mathcal{G}$ and topological sort σ. A set $\mathcal{T}$ is a *spanning set of train trees of* $\mathcal{G}(\sigma, i)$ if each connected component of $\mathcal{G}(\sigma, i)$ has a unique train tree $T \in \mathcal{T}$. If $\mathcal{G}$ has a spanning set of train trees, then every connected component of $\mathcal{G}$ has a relevant path and thus every connected component of G is interval, which means G is interval. The train tree algorithm will subsequently show that if G is interval, then $\mathcal{G}$ has a spanning set of train trees.

The train tree algorithm incrementally builds a spanning set of train trees of $\mathcal{G}$ by examining the boxes of $\mathcal{G}$ in reverse topological order. Let $\mathcal{T}$ be a spanning set of train trees of $\mathcal{G}(\sigma, i+1)$ and let S be a separator node of B_i. If $S \rightarrow S'$ and S' is contained in connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, i+1)$ with train tree $T \in \mathcal{T}$, then T is an *outtree* of S and B_i. The train tree algorithm merges B_i and the outtrees of B_i into a train tree of the connected component of $\mathcal{G}(\sigma, i)$ containing B_i; adding this train tree to $\mathcal{T}$ yields a spanning set of train trees of $\mathcal{G}(\sigma, i)$.

Given connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, i+1)$ with train tree $T \in \mathcal{T}$, the Main

Theorem-5 implies $G[V(\mathcal{H})]$ is an interval graph with clique-separator graph $\mathcal{H}$. Thus, the preceding lemmas hold with $G[V(\mathcal{H})], \mathcal{H}, T$ replacing $G, \mathcal{G}, T$, respectively. The correctness of the train tree algorithm relies on Corollary 3.12 and the Ordering Lemma, which is presented next. (The correctness of the update algorithm relies on Lemma 3.9, Lemma 3.10, and Corollary 3.12.)

3.4.5 The Ordering Lemma

In the rest of the thesis, *disjoint* means *node-disjoint*.

Lemma 3.13 *Let G be a chordal graph with clique-separator graph $\mathcal{G}$ and topological sort σ. Let $\mathcal{T}$ be a spanning set of train trees of $\mathcal{G}(\sigma, i+1)$ and let S be a separator node of B_i. Let $\hat{\mathcal{H}}$ be the connected component of $\mathcal{G}(\sigma, i)$ containing B_i. Suppose $\hat{\mathcal{H}}$ has a relevant path P.*

1. *If B_i has outtrees T, T', then there are disjoint subpaths of P respectively containing leaves(T) and leaves(T').*

2. *If B_i has inner node S with outtree T and with neighbors K_1, K_2, then S meets T and every node in leaves(T) is on $P(K_1, K_2)$.*

3. *If B_i has outer node S with outtree T, then S semi-meets or meets some $T' \equiv T$ and the following holds.*

   If S semi-meets T', then $N(body(B_i))$ and leaves(T) are contained in disjoint subpaths of P.

   If S meets T', then $N(body(B_i) - S)$ and leaves(T) are contained in disjoint subpaths of P.

4. *If B_i has outer node S and B_i is long (resp. short), then S semi-meets at most one outtree (resp. two outtrees).*

Proof We make some observations regarding any separator node S of B_i with outtree T. Let T be the train tree of connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, i+1)$. Since $\hat{\mathcal{H}}$ contains $\mathcal{H}$, then P contains $leaves(T)$. Since T represents every c.i.p. path on $leaves(T)$, then there exists $T' \equiv T$ such that $sequence(T')$ is a subsequence of P. Therefore, we assume henceforth that $sequence(T)$ is a subsequence of P. By Corollary 3.7, $body(B_i)$ is also a subsequence of P.

Observation 1. S does not appear between two nodes of $leaves(T)$ on P. (Otherwise, S appears between two consecutive nodes of $sequence(T)$ and by Corollary 3.12.1, a separator node in $leaves(T)$ is contained in S, contradicting the choice of σ.)

Observation 2. If $S \not\subset V(l), l \in leaves(T)$, then l does not appear between two nodes of $body(B_i)$ on P. (Otherwise, a separator node $S' \neq S$ in B_i is contained in $V(l)$, contradicting the Main Theorem-4.)

1. Suppose the claim is false. Then some $l \in leaves(T)$ appears between two consecutive nodes of $sequence(T')$. By Corollary 3.12.1, a separator node $S' \in leaves(T')$ is contained in $V(l)$, which implies S' is a predecessor of l in $\mathcal{G}$. Since S' is a node of $\mathcal{G}(\sigma, i+1)$, then $\mathcal{G}(\sigma, i+1)$ contains a directed path from a node in $leaves(T')$ to a node in $leaves(T)$, a contradiction.

2. Since S is an inner node, then there is a subpath (S_1, K_1, S, K_2, S_2) of $body(B_i)$ that is a subsequence of P. Suppose S does not meet T. Then $sequence(T)$ contains consecutive nodes l, l' such that $S \subset V(l)$ and $S \not\subset V(l')$. If l is not on $P(S_1, S_2)$, then $S \subset S_1$ or $S \subset S_2$, contradicting the Main Theorem-1. Thus, l is on $P(S_1, S_2)$. If l is not on $P(K_1, K_2)$, then S_1 or S_2 is contained in $V(l)$, contradicting the Main Theorem-4. Thus, l is on $P(K_1, K_2)$, which means (S_1, K_1, l, K_2, S_2) is a subsequence of P. By Observation 2, l' is not on $P(S_1, S_2)$. Suppose $(l', S_1, K_1, l, K_2, S_2)$ is a subsequence of P: the other case is symmetric. By Corollary 3.12.1, a separator node

$S' \in leaves(T)$ is contained in S_1, contradicting the choice of σ. Hence, S meets T. We have already shown that $S \subset V(l)$ implies l is on $P(K_1, K_2)$, so we are done.

3. Let l_1, l_2 be the endpoints of $sequence(T)$. By Observation 1, S is not on $P(l_1, l_2)$ and thus S is on a subpath of P disjoint from $P(l_1, l_2)$. Since S is contained in at least one node of $sequence(T)$, then S is contained in $V(l_1)$ or $V(l_2)$ or both. Thus, S semi-meets or meets T.

Suppose S semi-meets T. If B_i is long, let $S' \neq S$ be the other outer node of B_i, and if B_i is short, let $S' = S$. Since $body(B_i)$ is a subsequence of P, then every node of $body(B_i)$ is on $P(S, S')$. Since $S \not\subset V(l)$ for at least one node $l \in leaves(T)$, then by Observation 2, l is not on $P(S, S')$. By Observation 1, every node of $leaves(T)$ is on a subpath of P that is disjoint from $P(S, S')$. Hence, $leaves(T)$ and $N(body(B_i))$ are contained in disjoint subpaths of P.

Suppose S meets T. We are done if B_i is short, so assume B_i is long. Let $S' \neq S$ be the other outer node of B_i and let K be the neighbor of S in $body(B_i)$. If $l \in leaves(T)$ is on $P(K, S')$, then some separator node of B_i besides S is contained in $V(l)$, contradicting the Main Theorem-4. Thus, l is not on $P(K, S')$. By Observation 1, every node of $leaves(T)$ is on a subpath of P that is disjoint from $P(K, S')$. Hence, $leaves(T)$ and $N(body(B_i) - S)$ are contained in disjoint subpaths of P.

4. Suppose B_i is long and S semi-meets outtrees T_1, T_2. By parts 1 and 3, $leaves(T_1), leaves(T_2), N(body(B_i))$ are respectively contained in disjoint subpaths P_1, P_2, P_3 of P. If P_1, P_3, P_2 appear in this order on P, then S is contained in the other outer node of B_i, contradicting the Main Theorem-1. Suppose P_1, P_2, P_3 appear in this order on P; the other case is symmetric. Then S is contained in every node of P_2, which means S meets T_2, a contradiction. Hence, S semi-meets at most one outtree.

Suppose B_i is short and S semi-meets outtrees T_1, T_2, T_3. By part 1, $leaves(T_1)$, $leaves(T_2), leaves(T_3)$ are respectively contained in disjoint subpaths P_1, P_2, P_3 of P.

Without loss of generality, suppose P_1, P_2, P_3 appear in this order on P. Then S is contained in every node of P_2, which means S meets T_2, a contradiction. Hence, S semi-meets at most two outtrees. □

3.5 Deciding whether a graph is interval

Let G be a chordal graph with clique-separator graph $\mathcal{G}$ and topological sort σ. This section describes the train tree algorithm, which computes a relevant path of $\mathcal{G}$ or rejects because G is not interval. Given a relevant path of $\mathcal{G}$, a clique path of G is readily computed, which shows G is interval.

3.5.1 Overview

The train tree algorithm either computes a spanning set of train trees of $\mathcal{G}$, from which a relevant path of $\mathcal{G}$ is easily computed, or rejects because G is not interval. The algorithm maintains a spanning set of train trees $\mathcal{T}$ as it processes the boxes of $\mathcal{G}$ in reverse topological order. Let $\mathcal{T}$ be a spanning set of train trees of $\mathcal{G}(\sigma, i+1)$ and let S be a separator node of B_i. Let $\mathcal{T}_S$ be the set of outtrees of S. By the Main Theorem-4, if S, S' are distinct separator nodes of B_i, then $\mathcal{T}_S \cap \mathcal{T}_{S'} = \emptyset$. Let $\mathcal{T}_S^m$ and $\mathcal{T}_S^{sm}$ be the sets of outtrees of S that S meets and semi-meets, respectively. By Corollary 3.3.4, $components_{\mathcal{G}}(S) = degree(S) + |\mathcal{T}_S^m| + |\mathcal{T}_S^{sm}|$.

Informally, the algorithm processes B_i as follows. Let S_1, S_2 be the outer node(s) of B_i, where $S_1 = S_2$ if and only if B_i is short. In Step 1, the algorithm creates train tree T^{new}, which is a Q-node r whose children are $body(B_i)$. In Step 2, the algorithm examines the $\mathcal{T}^{sm}$ sets. By the Ordering Lemma-2, $\mathcal{T}_S^{sm} = \emptyset$ for every inner node S of B_i, or G is not interval. By the Ordering Lemma-4, there are at most two trees T_1, T_2 such that $T_1 \in \mathcal{T}_{S_1}^{sm}$ and $T_2 \in \mathcal{T}_{S_2}^{sm}$, or G is not interval. The algorithm merges T_1, T_2 into T^{new} such that $sequence(T^{new}) = sequence(T_1) + body(B_i) + sequence(T_2)$.

where + is concatenation. Using the Ordering Lemma-3, we will show that this is the unique way to perform the merge. In Step 3, the algorithm examines the $\mathcal{T}_S^m$ sets. For every separator node S of B_i with $\mathcal{T}_S^m \neq \emptyset$, the algorithm replaces S with a P-node whose meeting child is S and whose other children are the trees in $\mathcal{T}_S^m$. After Step 3, every outtree of B_i has been merged into T^{new}, and adding T^{new} to $\mathcal{T}$ yields a spanning set of train trees of $\mathcal{G}(\sigma, i)$.

The algorithm uses three subroutines: $Scan(S)$ computes $\mathcal{T}_S^m$, $\mathcal{T}_S^{sm}$, $SemiMeets(S, T)$ merges outtree T of outer node S into T^{new}, and $Meets(S, T)$ merges outtree T of any node S into T^{new}. The algorithm maintains the property that in every train tree T, every offspring S of a Q-node x has pointers $left(S), right(S)$ to the leftmost and rightmost offspring of x that contain S. Initially, $left(S) = right(S) = S$ for every S. These pointers, which are used by the update algorithm, are not affected by applying the PQ-tree operations to T.

Figure 3.14 shows the train tree algorithm applied to the graph in Figure 3.3.

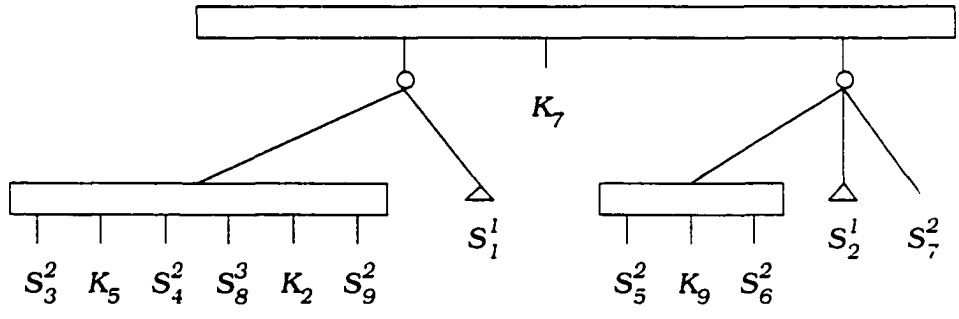
3.5.2 The train tree algorithm

Input: A chordal graph G and its clique-separator graph $\mathcal{G}$.

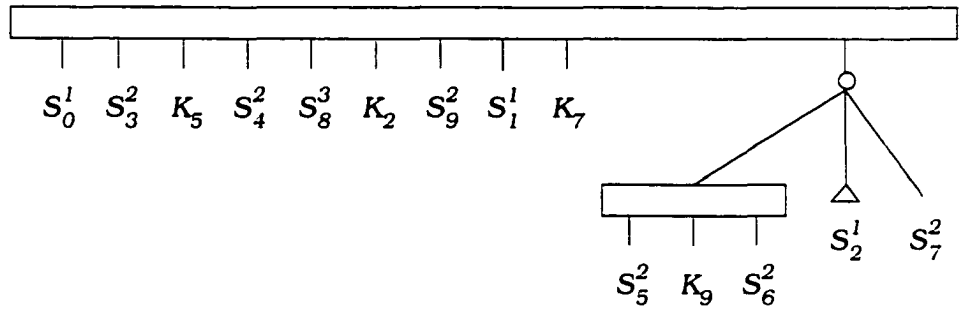
Output: A spanning set of train trees of $\mathcal{G}$ or a rejection.

If any box of $\mathcal{G}$ with three or more nodes is not a caterpillar, or any separator node of $\mathcal{G}$ has indegree greater than 2, then reject because G is not interval by the lemmas in Section 3. Otherwise, compute the contracted clique-separator graph $\mathcal{G}^c$ and a topological sort σ of $\mathcal{G}^c$. Index the boxes so that the boxes without outtrees (which have outdegree 0 in $\mathcal{G}^c$) are $B_j, B_{j+1}, \ldots, B_{b(\mathcal{G})}, j \leq b(\mathcal{G})$. For each $B_i, j \leq i \leq b(\mathcal{G})$, create the following train tree T: if B_i is long, T is a Q-node whose sequence of children is $body(B_i)$, and if B_i is short, T is the trivial tree containing the single node in $body(B_i)$.

(a) After processing 4 boxes.



(b) After processing 5 boxes.



(c) After processing all 6 boxes.

Figure 3.14: The train tree algorithm applied to Figure 3.3.

For each $i = 1, 2, \ldots, j-1$ in decreasing order, process B_i with the following three steps.

Step 1. [Create T^{new}.] Create train tree T^{new}, which is a Q-node r whose sequence of children is $body(B_i)$. For each separator node S of B_i, call $Scan(S)$ to compute $\mathcal{T}_S^m, \mathcal{T}_S^{sm}$ and assign $components_{\mathcal{G}}(S) \leftarrow degree(S) + |\mathcal{T}_S^m| + |\mathcal{T}_S^{sm}|$.

Step 2. [Examine the $\mathcal{T}_S^{sm}$ sets.]

Case A: B_i is long. For each inner node S of B_i, if $|\mathcal{T}_S^{sm}| > 0$, then reject. For each outer node S of B_i, if $|\mathcal{T}_S^{sm}| > 1$, then reject, and otherwise, if $\mathcal{T}_S^{sm} = \{T\}$, call $SemiMeets(S, T)$.

Case B: B_i is short with outer node S. If $|\mathcal{T}_S^{sm}| = 0$, then delete r so that T^{new} becomes the trivial tree containing S, and otherwise, for each $T \in \mathcal{T}_S^{sm}$, call $SemiMeets(S, T)$.

Step 3. [Examine the $\mathcal{T}_S^m$ sets.]

For each separator node S of B_i and each $T \in \mathcal{T}_S^m$, call $Meets(S, T)$. Add T^{new} to $\mathcal{T}$.

$Scan(S)$:

$Scan(S)$ computes $\mathcal{T}_S^m, \mathcal{T}_S^{sm}$ by examining S's immediate successors. For each immediate successor S', $Scan(S)$ marks one or two ancestors of S' in the train tree that contains it, and after examining all immediate successors, $Scan(S)$ unmarks certain nodes. Lemma 3.14 on page 71 shows that now each outtree T of S has a unique marked node x that satisfies the following. If x is a P-node, then a child of x is the unique immediate successor of S in $leaves(T)$ and (S meets x if and only if $S \rightarrow S_x$). If x is a Q-node with leftmost and rightmost children y_1, y_2, then every immediate successor of S in $leaves(T)$ is an offspring of x and (S meets x if and only if $S \rightarrow S_{y_1}$ and $S \rightarrow S_{y_2}$).

1. [Mark.] For each immediate successor S' of S, if S' is a trivial train tree, then add S' to $\mathcal{T}_S^m$, and otherwise, mark the parent y of S' in the train tree that contains it and if y is a P-node with Q-node parent z, mark z.

2. [Unmark.] For each marked Q-node w in any train tree, if w has exactly one offspring that is an immediate successor of S, then unmark w, and otherwise, unmark every marked child (if any) of w.

3. [Create $\mathcal{T}_S^m, \mathcal{T}_S^{sm}$.] For each marked node x in any train tree, if S meets x and x is the root of the outtree T that contains it, add T to $\mathcal{T}_S^m$, and otherwise, do the following three steps.

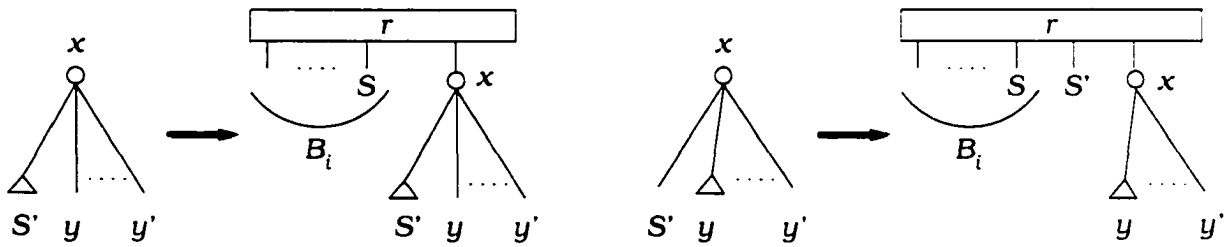
   (a) If x is a P-node, find the unique immediate successor S' of S in $leaves(x)$, which is a child of x. If x is a Q-node, find an immediate successor S' of S that is the leftmost or rightmost offspring of x; if S' doesn't exist, reject. (b) Make S' the leftmost leaf of T by applying the PQ-tree operations permute and reverse to the nodes on the path from S' to $root(T)$. If this is not possible, which means S' has an ancestor y is neither the leftmost nor the rightmost child of y's Q-node parent, then reject. (c) Add T to $\mathcal{T}_S^{sm}$ and record whether S meets x. If $|\mathcal{T}_S^{sm}| > 2$, reject.

$SemiMeets(S, T)$:

Let r be the parent of S. Let x be the unique marked node of T and let S' be the leftmost leaf of T, which were computed by $Scan(S)$. $SemiMeets(S, T)$ merges T into T^{new} as follows. First, x becomes a child of root r (possibly with one fewer child) or its children become children of r, depending on whether S meets x and whether x is a P-node or Q-node. Second, every proper ancestor of x in T becomes a child of r (with one fewer child) or its children become children of r, depending on whether x is a P-node or Q-node. Third, the pointers $left(S), right(S)$ are computed.

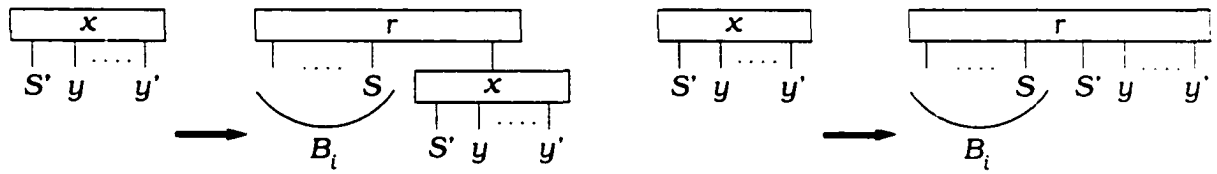
1. Case P: x is a P-node. If S meets x, make x the right sibling of S (Figure 3.15a). Otherwise, make S' the right sibling of S and make x the right sibling of S' (Figure 3.15b) and if x's only child is S_x, replace x with S_x.

   Case Q: x is a Q-node. If S meets x, make x the right sibling of S (Figure 3.16a). Otherwise, make S' the right sibling of S and make r the parent of x's children while preserving their order, and delete x (Figure 3.16b).



(a) S meets y *(b) S doesn't meet y*

Figure 3.15: Case P.



(a) S meets x *(b) S doesn't meet x*

Figure 3.16: Case Q.

2. If x was the root of T, then go to step 3. Otherwise, $x_0 = x$ had ancestors $x_1, x_2, \ldots, x_k$ in T, where each x_j is the left child of x_{j+1} and $k \geq 1$. For each $j = 1, 2, \ldots, k$ in increasing order, (a) if x_j is a P-node, make x_j a right sibling of x_{j-1} and if x_j's only child is S_{x_j}, replace x_j with S_{x_j}. (b) if x_j is a Q-node, make r the parent of x_j's children while preserving their order, and delete x_j.

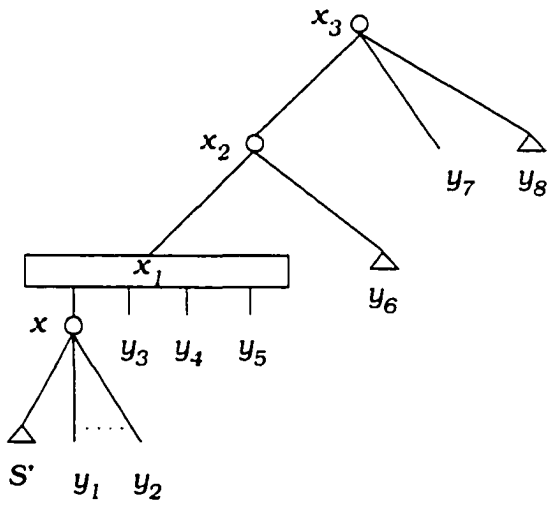
3. Compute the pointers $left(S), right(S)$ as follows. In Case P, if S met x, then assign $right(S) \leftarrow x$, and otherwise, assign $right(S) \leftarrow S'$. In Case Q, if S met

x, then assign $right(S) \leftarrow S$, and otherwise, assign $right(S) \leftarrow S''$, where S'' is the rightmost offspring of x that is an immediate successor of S.

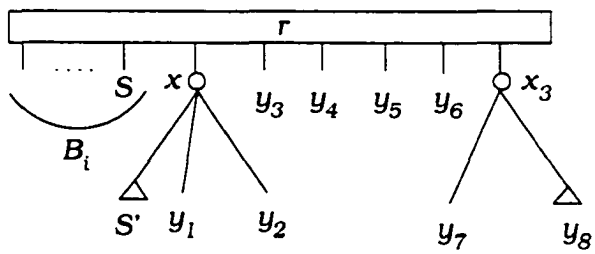
Figure 3.17 illustrates $SemiMeets(S, T)$ when x is a P-node and S meets x.

$Meets(S, T)$:

If S is not the meeting child of a P-node, which indicates this is the first call of $Meets(S, \cdot)$, then replace S with a P-node with meeting child S and child $root(T)$. Otherwise, S is the meeting child of some P-node x. Make $root(T)$ a child of x.



(a) Initial train tree T.



(b) Final train tree.

Figure 3.17: $SemiMeets(S, T)$.

3.5.3 Correctness and complexity

Lemma 3.14 *After step 2 of $Scan(S)$, each nontrivial outtree T of S has a unique marked node x, which is the lowest common proper ancestor of S's immediate successors in $leaves(T)$, and the following holds.*

- *If x is a P-node, then a child of x is the unique immediate successor of S in $leaves(T)$ and (S meets x if and only if $S \rightarrow S_x$).*
- *If x is a Q-node with leftmost and rightmost children y_1, y_2, then every immediate successor of S in $leaves(T)$ is an offspring of x and (S meets x if and only if $S \rightarrow S_{y_1}$ and $S \rightarrow S_{y_2}$).*

After step 3 of $Scan(S)$, $\mathcal{T}_S^m, \mathcal{T}_S^{sm}$ have been computed.

Proof Let T be a nontrivial outtree of S. Case 1: S has exactly one immediate successor $S' \in leaves(T)$. Then the parent y of S' in T is marked in step 1. If y is a P-node with Q-node parent z, then z is unmarked in step 2. Thus, y is the only marked node of T after step 2. Case 2: S has at least two immediate successors $S', S'' \in leaves(T)$. Since S', S'' are minimal nodes of $leaves(T)$ such that $S \subseteq S'$ and $S \subseteq S''$, then Corollary 3.12.2 implies S', S'' are offspring of the same Q-node z, which is marked in step 1. Then every marked child of z (which must be a P-node) is unmarked in step 2. Thus, z is the only marked node of T after step 2. Moreover, every immediate successor of S in $leaves(T)$ is an offspring of z.

We have shown that after step 2, each nontrivial outtree T of S has a unique marked node x, which is the lowest common proper ancestor of S's immediate successors in $leaves(T)$. We next show how to decide whether S meets x. Recall that S meets x if and only if S is a predecessor of every node in $leaves(x)$.

Suppose x is a P-node. Since S_x is the minimal node of $leaves(x)$, then S meets x if and only if $S \rightarrow S_x$. Suppose x is a Q-node with leftmost and rightmost children

y_1, y_2, which are S-nodes by train tree property 5c. Then S meets x if and only if $S \rightsquigarrow S_{y_1}$ and $S \rightsquigarrow S_{y_2}$. Since S_{y_1}, S_{y_2} are minimal nodes of x, then $S \rightsquigarrow S_{y_1}$ and $S \rightsquigarrow S_{y_2}$ if and only if $S \rightarrow S_{y_1}$ and $S \rightarrow S_{y_2}$. Thus, S meets x if and only if $S \rightarrow S_{y_1}$ and $S \rightarrow S_{y_2}$.

By definition, S meets T if and only if S meets $root(T)$. Since the preceding paragraph applies to $root(T)$, if x is not $root(T)$, then S doesn't meet T. Thus, $Scan(S)$ correctly adds T to $\mathcal{T}_S^m$ or $\mathcal{T}_S^{sm}$. □

Theorem 3.15 *Let G be a chordal graph with clique-separator graph $\mathcal{G}$. G is interval if and only if the train tree algorithm computes a spanning set of train trees of $\mathcal{G}$.*

Proof Initially, the algorithm creates a train tree for each $B_i, i = j, j+1, \ldots, b(\mathcal{G})$. Since each $body(B_i)$ is a constraining path by Corollary 3.7, this initial $\mathcal{T}$ is a spanning set of train trees of $\mathcal{G}(\sigma, j)$. Let $\mathcal{T}$ be a spanning set of train trees of $\mathcal{G}(\sigma, i+1), 1 \leq i \leq j-1$.

Suppose the algorithm rejects when processing B_i. If $Scan(S)$ rejects in step 4(a) or 4(b), then x has a Q-node ancestor y (possibly itself) such that S doesn't meet or semi-meet y. It follows that S doesn't semi-meet or meet any $T' \equiv T$, which means G is not interval by the Ordering Lemma-3. If $Scan(S)$ rejects in step 4(c), then G is not interval by the Ordering Lemma-4. If the algorithm rejects because S is an inner node and $|\mathcal{T}_S^{sm}| > 0$, then G is not interval by the Ordering Lemma-2. If it rejects because S is an outer node and $|\mathcal{T}_S^{sm}| > 1$, then G is not interval by the Ordering Lemma-4. Thus, if the algorithm rejects, then G is not interval.

Suppose the algorithm doesn't reject when processing B_i. In the rest of the proof, we show that after B_i is processed, T^{new} is a train tree of the connected component of $\mathcal{G}(\sigma, i)$ containing B_i and thus adding T^{new} to $\mathcal{T}$ yields a spanning set of train trees of $\mathcal{G}(\sigma, i)$. Hence, if the train tree algorithm never rejects, it computes a spanning set of train trees $\mathcal{T}$ of $\mathcal{G}$. Then every connected component of $\mathcal{G}$ has a relevant path and

thus every connected component of G is interval, which means G is interval.

Consider the processing of B_i. Step 1 creates T^{new} with root r. If B_i is long, then r has three or more children. If B_i is short, then r has one child S, violating train tree property 3. This is a temporary violation because if Step 2 makes no $SemiMeets$ calls, then r is deleted and T^{new} becomes trivial tree S, and otherwise, every $SemiMeets$ call adds at least two children to r, as we will show. (Whether B_i is long or short, if Step 2 makes no $SemiMeets$ calls, then Step 3 makes at least one call of $Meets$ because B_i has at least one outtree.)

The bulk of the proof considers a particular $SemiMeets(S, T)$ call, one of at most two calls by Step 2. Before the call, T is a train tree of a connected component $\mathcal{H}$ of $\mathcal{G}(\sigma, i+1)$. We will show that immediately after the call, T^{new} is a train tree of the subgraph $\mathcal{H}^{new}$ of $\mathcal{G}(\sigma, i)$ containing exactly B_i and $\mathcal{H}$ and the arcs from nodes of B_i to nodes of $\mathcal{H}$.

$SemiMeets(S, T)$ changes T as follows: every node on the path from x to $root(T)$ is made a child of root r (possibly with one fewer child) or its children become children of r. Thus, if z is a node of T^{new} such that z is neither r nor a child of r, then $subtree_{T^{new}}(z) = subtree_T(z)$. Since T had the six train tree properties, it will usually suffice to consider r and the children of r in order to show that T^{new} has the properties. We use $oldleaves(T)$ and $oldleaves(w), w \in T$, to denote $leaves(T)$ and $leaves(w)$ before the $SemiMeets$ call.

Property 1: $leaves(T^{new})$ is the set of relevant nodes of $\mathcal{H}^{new}$.

Since $oldleaves(T)$ is the set of relevant nodes of $\mathcal{H}$, then after the call, $leaves(T^{new}) = leaves(r) = body(B_i) \cup oldleaves(T)$ is the set of relevant nodes of $\mathcal{H}^{new}$. Thus, T^{new} has property 1.

Property 2: Every internal node y represents all c.i.p. paths on $leaves(y)$.

It suffices to show that root r and every child of r satisfies property 2. We will

show that throughout the *SemiMeets* call, root r represents every c.i.p. path P on $leaves(r)$ with the *Disjoint Property*: P has two disjoint subpaths that contain $\mathcal{N}(body(B_i))$ and $leaves(r) \cap oldleaves(T)$. Initially, $leaves(r) = \mathcal{N}(body(B_i))$ and $leaves(r) \cap oldleaves(T) = \emptyset$.

After step 1 of *SemiMeets*, $leaves(r) = \mathcal{N}(body(B_i)) \cup oldleaves(x)$. Since x represented every c.i.p. path on $oldleaves(x)$, then in any c.i.p. path on $leaves(r)$ with the Disjoint Property, the following holds: in Case P, if S meets x, then S and any node in $oldleaves(x)$ are consecutive, and otherwise, S and S' are consecutive, and S' and any node in $oldleaves(x) - \{S'\}$ are consecutive, and in Case Q, if S meets x, then S and any endpoint in $sequence(x)$ are consecutive, and otherwise, S and S' are consecutive. This implies that in both cases, if S meets x, then x must become the right sibling of S, and otherwise, S' must become the right sibling of S. Thus, r represents every c.i.p. path on $leaves(r)$ with the Disjoint Property after step 1.

If step 2 is omitted, then $leaves(r) = \mathcal{N}(body(B_i)) \cup oldleaves(T)$. Otherwise, x had ancestors $x_1, x_2, \ldots, x_k$ in T. By Lemma 3.14, S has no successors in $oldleaves(x_1) - oldleaves(x)$, which means S is not contained in any node in $oldleaves(x_1) - oldleaves(x)$. Since x_1 represented all c.i.p. paths on $oldleaves(x_1)$, then after substep (a) or (b) is applied to x_1, $leaves(r) \cap oldleaves(T) = oldleaves(x_1)$ and r represents every c.i.p. path on $leaves(r)$ with the Disjoint Property. Similarly, after substep (a) or (b) is applied to $x_2, \ldots, x_k$, $leaves(r) \cap oldleaves(T) = oldleaves(T)$ and r represents every c.i.p. path on $leaves(r)$ with the Disjoint Property. Thus, r represents every c.i.p. path on $leaves(r) = \mathcal{N}(body(B_i)) \cup oldleaves(T)$ with the Disjoint Property after step 2. By the Ordering Lemma-3, every c.i.p. path on $\mathcal{N}(body(B_i)) \cup oldleaves(T)$ has the Disjoint Property, which implies r represents every c.i.p. path on $leaves(r)$ after step 2.

We next show every child of r satisfies property 2. Consider any $x_j, 0 \leq j \leq k$, before step 2. If x_j is a P-node, then x_j becomes a child of root r with one fewer

child, which implies x_j satisfies property 2 in T^{new}. If x_j is a Q-node, then every child y of x_j becomes a child of root r and $subtree_{T^{new}}(y) = subtree_T(y)$, which implies y satisfies property 2 in T^{new}. Thus, T^{new} has property 2.

Property 3: Every P-node has at least two children. Every Q-node has at least three children.

It suffices to show that root r and every child of root r satisfies property 3. Consider steps 1 and 2 of *SemiMeets*. If S meets x, then x is not the root of T because S semi-meets T, which implies step 1 of *SemiMeets* adds one child to r and step 2 adds at least one more child to r. If S doesn't meet x, then step 1 adds at least two children to r. Therefore, *SemiMeets* adds at least two children to r and thus r is a Q-node of T^{new} with at least three children. Also, if any $x_j, 0 \leq j \leq k$, becomes a P-node whose only child is S_{x_j}, then x_j is replaced with S_{x_j}. Thus, T^{new} has property 3.

The remaining properties are more complicated. To prove properties 4 and 5, we use the following observation. By the Main Theorem-3, if a separator node divides two nodes in $\mathcal{H}$ ($\subseteq \mathcal{G}(\sigma, i+1)$), then it divides the same nodes in $\mathcal{H}^{new}$ ($\subseteq \mathcal{G}(\sigma, i)$). By the Main Theorem-4, S divides $body(B_i) - S$ and $\mathcal{H}$ in $\mathcal{H}^{new}$.

Property 4: Every P-node y has a *meeting child* S_y, which is a separator node such that S_y meets y and satisfies the following.

a. S_y divides $leaves(y) - \{S_y\}$ and $leaves(T^{new}) - leaves(y)$ in $\mathcal{H}^{new}$.

b. S_x divides $leaves(z)$ and $leaves(z')$ in $\mathcal{H}^{new}$ for any children $z, z' \neq S_y$ of y.

Since property 4 is a property of P-nodes, we do not need to consider the Q-nodes of T^{new}. We first consider each $x_j, 0 \leq j \leq k$, that is a P-node of T and then consider each $x_j, 0 \leq j \leq k$, that is a Q-node of T.

Suppose x is a P-node of T (Case P) before step 1. Then S has a unique immediate successor $S' \in oldleaves(T)$ and x becomes a P-node of T^{new}, possibly

with one fewer child. By the observation, x satisfies property 4b, whether or not x has one fewer child. It is more difficult to show x satisfies property 4a because $leaves(T^{new}) = N(body(B_i)) \cup oldleaves(T)$. If S meets x, then $S \rightarrow S' = S_x$ and thus no connected component of $\mathcal{H}^{new} - Preds(S_x)$ contains a node of $body(B_i)$ and a node of $oldleaves(T)$. By the observation, x satisfies property 4a. If S doesn't meet x, then $S \rightarrow S' \neq S_x$ and $S \not\rightarrow S''$ for any $S'' \in oldleaves(x) - \{S'\}$. Then some connected component of $\mathcal{H}^{new} - Preds(S_x)$ contains $body(B_i)$ and S'. By the observation, x satisfies property 4a. Thus, x satisfies property 4 in T^{new}. Since S didn't meet any of $x_1, x_2, \ldots, x_k$, a similar argument shows that for every $1 \leq j \leq k$, if x_j is a P-node, then x_j satisfies property 4 in T^{new}.

Suppose x is a Q-node of T (Case Q) before step 1. If S meets x, then x becomes a Q-node of T^{new}, so suppose S doesn't meet x. Then every P-node child y of x in T becomes a P-node child of r in T^{new}. Now S may have several immediate successors in $oldleaves(T)$, each of which is an offspring of x in T. If $S \rightarrow S_y$, then no connected component of $\mathcal{H}^{new} - Preds(S_y)$ contains a node of $body(B_i)$ and a node of $oldleaves(T)$. If $S \not\rightarrow S_y$, then some connected component of $\mathcal{H}^{new} - Preds(S_y)$ contains $body(B_i)$ and S'. In either case, the observation implies x satisfies property 4. A similar argument shows that for every $1 \leq j \leq k$, if x_j is a Q-node, then every P-node child of x_j in T becomes a P-node of T^{new} that satisfies property 4.

We have shown that every P-node child of root r satisfies property 4. Consider a P-node z of T^{new} that is not a child of r. Then $subtree_{T^{new}}(z) = subtree_T(z)$. If $S \rightsquigarrow S_z$, then no connected component of $\mathcal{H}^{new} - Preds(S_z)$ contains a node of $body(B_i)$ and a node of $oldleaves(T)$. If $S \not\rightsquigarrow S_z$, then $S' \not\rightsquigarrow S_z$ and some connected component of $\mathcal{H}^{new} - Preds(S_z)$ contains $body(B_i)$ and S'. In either case, the observation implies x satisfies property 4. Thus, T^{new} has property 4.

Property 5: Every Q-node y with children $(z_1, z_2, \ldots, z_k)$ satisfies the following.

a. If any z_i is a Q-node, then $1 < i < k$ and z_{i-1}, z_{i+1} are S-nodes such that the *meeting siblings* $S_{z_{i-1}}, S_{z_{i+1}}$ of z_i are incomparable and meet z_i.

b. If any z_i is an S-node, then S_{z_i} divides $leaves(z_1) \cup \cdots \cup leaves(z_{i-1})$ and $leaves(z_{i+1}) \cup \cdots \cup leaves(z_k)$ in $\mathcal{H}^{new}$.

c. z_1, z_k are S-nodes and no separator node in $leaves(y)$ is a predecessor of S_{z_1} or S_{z_k} in $\mathcal{H}^{new}$.

If y is a Q-node of T^{new} and $y \neq r$, then $subtree_{T^{new}}(y) = subtree_T(y)$ and by the observation, y satisfies property 5 in T^{new}. Thus, it suffices to show root r satisfies property 5 in T^{new}.

Property 5a: We must consider every Q-node child of root r. Suppose x is a Q-node of T^{new}. Then x was a Q-node of T and S met x, which implies x had a parent x_1. Since x was the leftmost child of x_1, then x_1 was a P-node of T because a Q-node cannot be the leftmost child of another Q-node by property 5a. Then S_{x_1} met x in T. Since S and S_{x_1} are incomparable, then S and S_{x_1} are the meeting siblings of x in T^{new}. Suppose $y \neq x$ is a Q-node child of r in T^{new}. Then y was a Q-node child of some x_j, $1 \leq j \leq k$, that was a Q-node of T. Since y had meeting siblings in T, then y has the same meeting siblings in T^{new}. Thus, r satisfies property 5a.

Property 5b: We must consider every S-node child of root r. By Corollary 3.3.4, if $\hat{S}$ is any separator node of B_i, then $\hat{S}$ satisfies property 5b.

Suppose x is a P-node of T (Case P) before step 1. Then x satisfied property 4 in T. If S meets x, then x becomes a P-node of T^{new} and by the observation, S_x satisfies property 5b in T^{new}. Otherwise, S' becomes the right sibling of S and x becomes a P-node with one fewer child. Since $S \rightarrow S'$, then S' satisfies property 5b and by the observation, S_x also satisfies property 5b.

Suppose x was a Q-node of T (Case Q) before step 1. Then x satisfied property 5b

in T. If x is a Q-node of T^{new}, we need not consider it. Otherwise, the children of x in T become children of r in T^{new}. Since $S \rightarrow S'$, then S' satisfies property 5b. Consider the other S-node children of x in T. If x was the root of T, then each S-node child of x in T^{new} satisfies property 5b and we are done. Otherwise, x_1 was a P-node of T because a Q-node cannot be the leftmost child of another Q-node. Then S_{x_1} was a predecessor of each S-node child of x in T and therefore each child satisfies property 5b in T^{new}. A similar argument shows that for every $1 \leq j \leq k$, if x_j was a P-node of T, then by the observation, S_{x_j} satisfies property 5b in T^{new}, and if x_j was a Q-node of T, then every S-node child of x_j becomes an S-node child of root r that satisfies property 5b in T^{new}. Thus, r satisfies property 5b.

Property 5c: We must show that the leftmost and rightmost children w, z of r are S-nodes and that no separator node in $leaves(r)$ is a predecessor of S_w or S_z in $\mathcal{H}^{new}$. Exactly one of w, z is a node of B_i, say w. Then w is a separator node of B_i and thus no separator node in $leaves(r)$ is a predecessor of S_w.

Consider z. Before the *SemiMeets* call, the root of T is $x_k, k \geq 0$. Case 1: x_k is a P-node. Then $z = x_k$ and since S_z met T, no separator node in $oldleaves(T)$ is a predecessor of S_z. Since S semi-met T, then S is not a predecessor of S_z. Thus, no separator node in $leaves(r)$ is a predecessor of S_z. Case 2: x_k is a Q-node. Then z was the rightmost child of x_k in T. Since T satisfied property 5c, then z is an S-node and no separator node in $oldleaves(T)$ is a predecessor of S_z. Since S semi-met T, then S is not a predecessor of S_z. Thus, no separator node in $leaves(r)$ is a predecessor of S_z, which means r satisfies property 5c. Hence, T^{new} has property 5.

Property 6: Every offspring block of a Q-node is a train. The body of every box contained in $\mathcal{H}^{new}$ is an offspring block of some Q-node.

In the *SemiMeets* call, $body(B_i)$ becomes an offspring block of root r and every Q-node in T is either unchanged or merged into root r. Therefore, $body(B_i)$ is in some

offspring block throughout the remainder of the train tree algorithm. Therefore, it suffices to show that every offspring block of r is a train.

Clearly, $body(B_i)$ is a train. Consider x. In Case P, x was a P-node of T, so if S met x, then $S \subset S_x$, and otherwise, $S \subset S' \supset S_x$. In Case Q, x was a Q-node of T, so if S met x, then x is a Q-node in T^{new} and not in any offspring block, and otherwise, $S \subset S'$ and since S' was in an offspring bloc in T, then S' is in the same offspring block in T^{new}.

Consider x_1 before step 2. Case 1: x_1 is a P-node. Then let z be the left sibling of x_1 in T^{new}. What is z? If x was a P-node of T (Case P), then $z = x$ is a P-node in T^{new} and $S_z \supset S_{x_1}$. Suppose x was a Q-node of T (Case Q). If S met x, then $z = x$ is a Q-node in T^{new} and not in any offspring block, and otherwise, z was x's rightmost child in T, which is an S-node because x satisfied property 5c in T. Since S_{x_1} met x in T, then $S_z \supset S_{x_1}$. Case 2: x_1 is a Q-node. Since x was the leftmost child of x_1, then x was a P-node of T (Case P). Since S_x is in an offspring block in T, then S_x is in the same offspring block in T^{new}. A similar argument for each of $x_2, \ldots, x_k$ shows that every offspring block of r is a train. Thus, T^{new} has property 6.

We have shown that after the $SemiMeets(S, T)$ call, T^{new} is a train tree of the subgraph $\mathcal{H}^{new}$ of $\mathcal{G}(\sigma, i)$ containing exactly B_i and $\mathcal{H}$ and the arcs from nodes of B_i to nodes of $\mathcal{H}$. Now Step 2 of the algorithm may call $SemiMeets(S', T')$ for outtree $T' \neq T$ of outer node S' of B_i, where $S' = S$ if and only if B_i is short. Before the call, T' is a train tree of a connected component $\mathcal{H}'$ of $\mathcal{G}(\sigma, i+1)$. By Corollary 3.3.4, the foregoing argument applies to the $SemiMeets(S', T')$ call. It follows that after Step 2 of the train tree algorithm, T^{new} is a train tree of the subgraph of $\mathcal{G}(\sigma, i)$ containing exactly $\mathcal{H}, B_i, \mathcal{H}'$ and the arcs from nodes of B_i to nodes of $\mathcal{H}$ or $\mathcal{H}'$.

Step 3 of the algorithm calls $Meets(S, T)$ for each separator node S of B_i and each $T \in \mathcal{T}_S^m$. We will show that T^{new} is now a train tree of the connected component $\hat{\mathcal{H}}$ of $\mathcal{G}(\sigma, i)$ containing B_i. The only train tree properties of T^{new} that are nontrivial

are properties 1, 2, 4, and 5b. After Step 3, $leaves(r)$ contains the leaves of every outtree of B_i, which means $leaves(r)$ is the set of relevant nodes of $\hat{\mathcal{H}}$. Thus, T^{new} has property 1.

If B_i is short with separator node S and Step 2 made no $SemiMeets$ calls, then the root of T^{new} is a P-node with meeting child S and by the Ordering Lemma-1, r represents all c.i.p. paths on $leaves(r)$. Otherwise, B_i is long or Step 2 made at least one $SemiMeets$ call. Then the root of T^{new} is a Q-node and by the Ordering Lemma-1, 2, 3, r represents all c.i.p. paths on $leaves(r)$. Thus, T^{new} has property 2. Lastly, by Corollary 3.3.4, T^{new} has properties 4 and 5b. We conclude that T^{new} is a train tree of $\hat{\mathcal{H}}$ after Step 3. Hence, adding T^{new} to $\mathcal{T}$ yields a spanning set of train trees of $\mathcal{G}(\sigma, i)$. □

Theorem 3.16 *The train tree algorithm runs in $O(n)$ time, where n is the number of vertices in G.*

Proof Since $\mathcal{G}$ has size $O(n)$ by Lemma 3.5.3, computing $\mathcal{G}^c$ and a topological sort of $\mathcal{G}^c$ requires $O(n)$ time. We must bound the cost of all calls to $Scan$, $SemiMeets$, and $Meets$. Each $Meets$ call takes $O(1)$ time, so all $Meets$ calls take $O(n)$ time. Since $Scan(S)$ requires $O(1)$ time to examine each outgoing arc from S and there are $O(n)$ arcs in $\mathcal{G}$, the total cost of steps 1-2 in all $Scan$ calls is $O(n)$ time. In step 3, $Scan(S)$ follows parent pointers from x to $root(T)$ and $Scan(S)$ doesn't reject only if $T \in \mathcal{T}_S^{sm}$ and $|\mathcal{T}_S^{sm}| \leq 2$, in which case $SemiMeets(S, T)$ is subsequently called. Thus, the cost of following parent pointers in $Scan(S)$ is bounded by the cost of following parent pointers in $SemiMeets(S, T)$, which we analyze next.

The train tree algorithm may create a Q-node for each box B_i processed and it may create a P-node for each separator node of B_i. Therefore, the total number of (train tree) internal nodes created is bounded by the number of boxes in $\mathcal{G}$ plus the number of separator nodes in $\mathcal{G}$, which at most $2n$. Now in any call of $SemiMeets(S, T)$, either

the algorithm rejects, or every internal node on the path from x to $root(T)$ becomes a child of root r of T^{new}. Therefore, during the course of processing $B_1, B_2, \ldots, B_{j-1}$, each parent pointer is followed at most once. Thus, the total cost of following parent pointers in all $Semi.Meets$ calls is $O(n)$.

Finally, we must bound the cost of making the children of each Q-node $x_j, 0 \leq j \leq k$, into children of root r. We represent a Q-node's children with a linked list, so that in $O(1)$ time, we can concatenate two lists and compute the parent pointer for the first and last elements of the list. Since there is at most one concatenation for each parent pointer followed, the total cost of concatenating in all $Semi.Meets$ calls is $O(n)$. In a later call of $Scan$, if $Scan$ follows parent pointers and an ancestor y of x is not the first or last child of its Q-node parent, then y doesn't have a parent pointer, but in this case $Scan(S)$ rejects because S doesn't semi-meet T. Hence, the train tree algorithm runs in $O(n)$ time. □

3.5.4 Computing a clique path

Let $\mathcal{T}$ be a spanning set of train trees of $\mathcal{G}$. Since each $T \in \mathcal{T}$ is a train tree of a connected component $\mathcal{H}$ of $\mathcal{G}$, then $sequence(T)$ is a relevant path of $\mathcal{H}$. Concatenating the paths in $\{sequence(T) \mid T \in \mathcal{T}\}$ in any order yields a relevant path P of $\mathcal{G}$.

For each separator node S in P, replace S in P with any path on $\mathcal{K}_S$, where $\mathcal{K}_S$ is the (possibly empty) set of leaf clique nodes adjacent to S in $\mathcal{G}$. By Lemma 3.1.4, each $K \in \mathcal{K}_S$ is the unique node of $\mathcal{G}$ containing any vertex in $K - S$. Thus, the resulting path is a c.i.p. path on the clique nodes of $\mathcal{G}$, which is a clique path of G. This establishes the following theorem.

Theorem 3.17 *Let G be a chordal graph. The given algorithm computes a clique path of G if and only if G is interval.*

3.6 Updates

Let G be an interval graph with clique-separator graph $\mathcal{G}$ and a spanning set of train trees $\mathcal{T}$ of $\mathcal{G}$. In this section, we show how to update $\mathcal{G}$ after an edge insertion or deletion in G. In each case, updating requires only a small, local change in $\mathcal{G}$. We then use the train tree algorithm to compute a spanning set of train trees $\mathcal{T}'$ of the updated $\mathcal{G}'$.

3.6.1 Insert

Throughout this subsection, $\{u, v\} \notin E(G)$. Then $\{u, v\}$ is not contained in any node of $\mathcal{G}$. Assume for the moment that G is connected, so that $\mathcal{G}$ is connected. The next theorem, which is copied from Chapter 2, does not require that G be connected, but the subsequent theorem does require that G be connected because it depends on Theorem 1.1, which assumes a connected graph.

Theorem 3.18 *Let G be a chordal graph and let $\{u, v\} \notin E(G)$. Then $G + \{u, v\}$ is chordal if and only if G has a clique tree T with $u \in K, v \in K'$ for some $\{K, K'\} \in E(T)$.*

Theorem 3.19 *Let G be a connected chordal graph with clique-separator graph $\mathcal{G}$ and let $\{u, v\} \notin E(G)$. Then $G + \{u, v\}$ is chordal if and only if $\mathcal{G}$ has a separator node S and clique nodes K, K' such that S divides K, K' in $\mathcal{G}$ and $S = K \cap K'$ and $u \in K, v \in K'$. Moreover, S is unique if it exists.*

Proof We claim that for any maximal cliques K, K', G has a clique tree T such that $\{K, K'\} \in E(T)$ if and only if $\mathcal{G}$ has a separator node S such that S divides K, K' in $\mathcal{G}$ and $S = K \cap K'$. ($\Rightarrow$) By Theorem 3.18, G has a clique tree T with $u \in K, v \in K'$ for some $\{K, K'\} \in E(T)$. By Theorem 1.1, $S = K \cap K'$ is a minimal vertex separator of G and S separates $K - S$ and $K' - S$ in G. By the Main Theorem-2, S divides

K, K' in $\mathcal{G}$. ($\Leftarrow$) Let T be a clique tree of G and let $K = K_0, K_1, \ldots, K_j = K'$ be the K-K' path in T, $j \geq 1$. Since K_0, K_j are in different connected components of $\mathcal{G} - Preds(S)$ by the definition of divides, then for some $0 \leq i \leq j-1$, K_i, K_{i+1} are in different connected components of $\mathcal{G} - Preds(S)$. By the clique intersection property, $S \subseteq K_i \cap K_{i+1}$. Since S divides K_i, K_{i+1}, then S separates $K_i - S$ and $K_{i+1} - S$ and so $S = K_i \cap K_{i+1} = K_0 \cap K_j$. Hence, $T - \{K_i, K_{i+1}\} + \{K_0, K_j\}$ is a clique tree of G.

Theorem 3.18 and the claim imply the theorem, except for the uniqueness of S. Suppose separator nodes S, S' satisfy the theorem. By the claim, there are clique trees T, T' of G such that for some $\{K_1, K_2\} \in E(T)$, $S = K_1 \cap K_2$ and $u \in K_1, v \in K_2$, and for some $\{K_1', K_2'\} \in E(T')$, $S' = K_1' \cap K_2'$ and $u \in K_1', v \in K_2'$. Since $\{K_1, K_2\}$ is on the K_1'-K_2' path in T, then by the clique intersection property, $K_1 \cap K_2 \supseteq K_1' \cap K_2'$. Similarly, $K_1 \cap K_2 \subseteq K_1' \cap K_2'$. Thus, $S = K_1 \cap K_2 = K_1' \cap K_2' = S'$. □

For notational convenience, henceforth we use $v_1 = u$ and $v_2 = v$. Figure 3.18 illustrates the premise of the following lemma, which specifies the clique nodes and separator nodes of $\mathcal{G}'$.

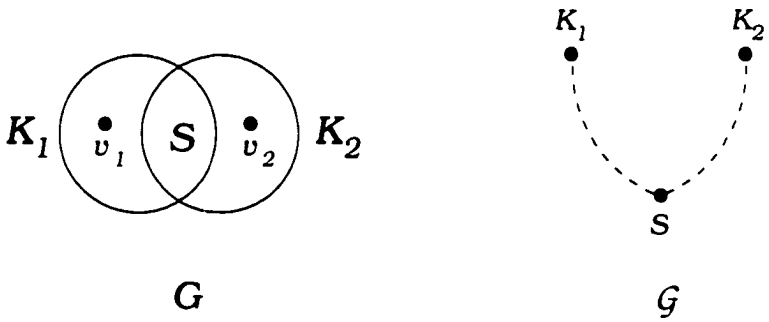


Figure 3.18: $S = K_1 \cap K_2$ and S divides K_1, K_2.

Lemma 3.20 *Let G be a connected chordal graph with clique-separator graph $\mathcal{G}$ and let $\{v_1, v_2\} \notin E(G)$. Suppose $G' = G + \{v_1, v_2\}$ is chordal with clique-separator graph $\mathcal{G}'$. Let S be the unique separator node of $\mathcal{G}$ such that S divides clique nodes K_1, K_2 in $\mathcal{G}$ and $S = K_1 \cap K_2$ and $v_1 \in K_1, v_2 \in K_2$.*

1. *Every clique node* $K' \neq K_1, K_2$ *of* $\mathcal{G}$ *is a clique node of* $\mathcal{G}'$.

2. $K = S \cup \{v_1, v_2\}$ *is a clique node of* $\mathcal{G}'$.

3. K_i *is a clique node of* $\mathcal{G}'$ *if and only if* $|K_i - S| > 1, i = 1, 2$. *Moreover, if* $|K_i - S| = 1$, *then* K_i *is a neighbor of* S *in* $\mathcal{G}$.

4. *Every separator node* $S' \neq S$ *of* $\mathcal{G}$ *is a separator node of* $\mathcal{G}'$.

5. $S_i = S \cup \{v_i\}$ *is a separator node of* $\mathcal{G}'$ *if and only if* $|K_i - S| > 1, i = 1, 2$.

6. S *is a separator node of* $\mathcal{G}'$ *if and only if* $components_{\mathcal{G}}(S) > 2$.

Proof

1. By Section 2.3.4.

2. By Section 2.3.4.

3. The first part follows by Section 2.3.4 and the second part follows because $S \subset K_i$ and $|K_i - S| = 1$ implies K_i and S are neighbors in $\mathcal{G}$.

4. By Theorem 1.1, for some maximal cliques K_1', K_2' of G, $S' = K_1' \cap K_2'$ and S' separates $K_1' - S'$ and $K_2' - S'$ in G. Let V_1, V_2 be the vertex sets of the connected components of $G - S'$ that contain $K_1' - S'$, $K_2' - S'$ and let G_1, G_2 be the subgraphs of G induced by $V_1 \cup S'$, $V_2 \cup S'$, respectively. (Figure 3.19.) We claim S' separates $K_1' - S'$ and $K_2' - S'$ in G'. Suppose otherwise. Since $G' = G + \{v_1, v_2\}$, then $v_1 \in V_1, v_2 \in V_2$ or vice versa. Since S' separates V_1, V_2 in G, then S' is a v_1v_2-separator of G. Since every vertex of S is adjacent to v_1 and v_2, then $S \subset S'$. Since G_1 (resp. G_2) contains a path from a vertex of $S' - S$ to v_1 (resp. v_2), then v_1, v_2 are connected in $G - S$, a contradiction. Thus, S' separates $K_1' - S'$ and $K_2' - S'$ in G'. Since every vertex of S' is adjacent to every vertex of $K_1' - S'$ and $K_2' - S'$, then S' is an minimal vertex separator of G', which means $S' \in \mathcal{G}'$.

5. ($\Leftarrow$) Suppose $|K_1 - S| > 1$. Then there exists a vertex $w \in K_1 - S$. Since S is a minimal wv_2-separator of G, then $S_1 = \cup\{v_1\}$ is a minimal wv_2-separator of G'.

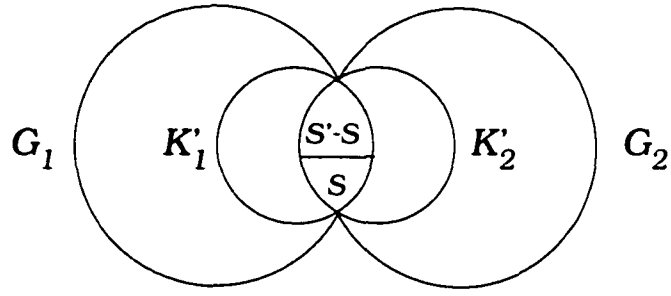


Figure 3.19: Proof of Lemma 3.20.4.

which means $S_1 \in \mathcal{G}'$. ($\Rightarrow$) Suppose $|K_1 - S| = 1$. By part 3, $K_1 = S \cup \{v_1\} = S_1$. If S_1 is a minimal vertex separator of G', then by Theorem 1.1, there exists a maximal clique $K' \neq K$ of G' such that $K_1 = S_1 \subset K'$, a contradiction because K' is also a maximal clique of G'. Thus, S_1 is not a minimal vertex separator of G', which means $S_1 \notin \mathcal{G}'$.

We showed $S_1 \in \mathcal{G}'$ if and only if $|K_1 - S| > 1$. The symmetric argument shows that $S_2 \in \mathcal{G}'$ if and only if $|K_2 - S| > 1$. (This result is implicit in the *Insert* operation in Section 2.3.4.)

6. ($\Leftarrow$) Suppose $components_{\mathcal{G}}(S) > 2$. Then $\mathcal{G}$ contains a clique node $K_3 \neq K_1, K_2$ such that $S \subset K_3$ and K_1, K_2, K_3 are in three different connected components of $\mathcal{G} - Preds(S)$. Then S separates $(K_1 \cup K_2) - S$ and $K_3 - S$ in G. Since $K = S \cup \{u, v\} \subseteq K_1 \cup K_2$, then S separates $K - S$ and $K_3 - S$ in G'. By Corollary 3.3.3, $S = K_1 \cap K_3 = K_2 \cap K_3$, which implies $S = K \cap K_3$. Since S separates $K - S$ and $K_3 - S$, then S is an minimal xy-separator of G' for every $x \in K - S$ and $y \in K_3 - S$. Thus, S is a minimal vertex separator of G', which means $S \in \mathcal{G}'$.

($\Rightarrow$) Suppose $components_{\mathcal{G}}(S) = 2$. Let V_1, V_2 be the vertex sets of the connected components of $G - S$ that contain $K_1 - S, K_2 - S$ and let G_1, G_2 be the subgraphs of G induced by $V_1 \cup S, V_2 \cup S$, respectively. (Figure 3.19 shows a similar case.) Then every maximal clique of G that contains S is contained in G_1 or G_2. Since G' contains edge $\{v_1, v_2\}$ and $v_1 \in V_1, v_2 \in V_2$, then G' does not have no two maximal cliques

$\bar{K}_1, \bar{K}_2$ such that $S = \bar{K}_1 \cap \bar{K}_2$ and S separates $\bar{K}_1 - S$ and $\bar{K}_2 - S$. By Theorem 1.1, S is not a minimal vertex separator of G', which means $S \notin \mathcal{G}'$. □

The $Insert(v_1, v_2)$ operation computes $\mathcal{G}'$ as follows. If v_1, v_2 are in different connected components of G, then the operation merges the connected components and merges their clique-separator graphs. Otherwise, the operation finds the nodes S, K_1, K_2 specified by Theorem 3.19 as follows. Given a clique tree of G, the *Insert* operation in Section 2.3.4, finds a clique tree T such that $u \in K_1, v \in K_2$ for some $\{K_1, K_2\} \in E(T)$ or it verifies that no such T exists. By Theorem 1.1, $S = K_1 \cap K_2$ is a minimal vertex separator of G and S separates $K_1 - S$ and $K_2 - S$ in G, which implies S divides K_1, K_2 in $\mathcal{G}$. Thus, we use the operations in Section 2.3.4, to maintain a clique tree of G. (Recall that if G is interval, then an arbitrary clique tree of G may not be a path.)

After S, K_1, K_2 are found, the operation makes the following node changes in $\mathcal{G}$, as required by Lemma 3.20.

- Create K.
- Delete K_i if $|K_i - S| = 1$, $i = 1, 2$.
- Create S_i if $|K_i - S| > 1$ and S_i is not already a node, $i = 1, 2$.
- Delete S if $components_{\mathcal{G}}(S) = 2$.

After $\mathcal{G}'$ is computed, if any box with three or more nodes is not a caterpillar, or any separator node has indegree greater than 2, then the operation rejects because G' is not interval by the lemmas in Section 3. If $\mathcal{G}'$ satisfies the lemmas and yet G' is not interval, then the train tree algorithm, which computes a spanning set of train trees of $\mathcal{G}'$, rejects the insertion.

For an example of an edge inserted between two different connected components, let G be the graph in Figure 3.2 with the cut edge K_7 deleted, so that its clique-

separator graph $\mathcal{G}$ is the graph in Figure 3.3 with nodes S_1^1, K_7, S_2^1 deleted. Inserting K_7 into G yields the original graph and clique-separator graph.

Figure 3.20 shows a chordal graph G, its clique-separator graph $\mathcal{G}$, and the clique-separator graph that results when the edges e, f, g, h are individually inserted into G. The separator node and clique nodes specified by Theorem 3.19 are the following: (e) S_3, K_1, K_6 or S_3, K_2, K_6, (f) S_2, K_2, K_3, (g) S_4, K_4, K_5, (h) S_4, K_2, K_5. Each insertion yields an interval graph except (h).

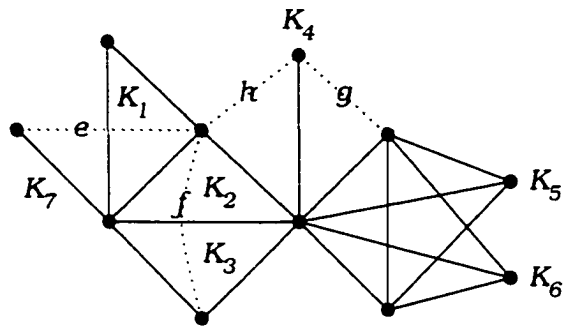
Insert(v_1, v_2):

1. If v_1, v_2 are in the same connected component of G, then go to step 2. Otherwise, v_1, v_2 are in connected components G_1, G_2 of G with clique-separator graphs $\mathcal{G}_1, \mathcal{G}_2$, respectively. Create clique node $K = \{v_1, v_2\}$ and let $S_1 = \{v_1\}, S_2 = \{v_2\}$. For each $i = 1, 2$, if $S_i \in \mathcal{G}_i$, create edge $\{S_i, K\}$, and otherwise, do the following.

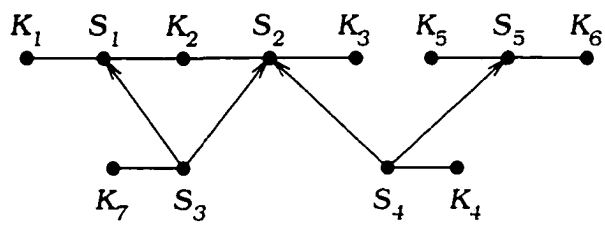
   Create S_i and edge $\{S_i, K\}$ and compute the other edges and arcs incident to S_i as follows. Compute the set $\mathcal{S}$ of minimal nodes of $\{X \mid X$ is a node of $\mathcal{G}_i$ such that $v_i \in X\}$ by examining the nodes of $\mathcal{G}_i$ in topological order, where each edge is viewed as an arc from separator node to clique node. If $\mathcal{S}$ contains a clique node K', create edge $\{S_i, K'\}$, and otherwise, create an arc from S_i to every separator node in $\mathcal{S}$. (Lemma 3.21 will show that if $\mathcal{S}$ contains a clique node K', then $\mathcal{S} = \{K'\}$.)

2. Use the clique tree of G to find the separator node S and clique nodes K_1, K_2 such that S divides clique nodes K_1, K_2 in $\mathcal{G}$ and $S = K_1 \cap K_2$ and $v_1 \in K_1, v_2 \in K_2$. If no such nodes exist, then reject.

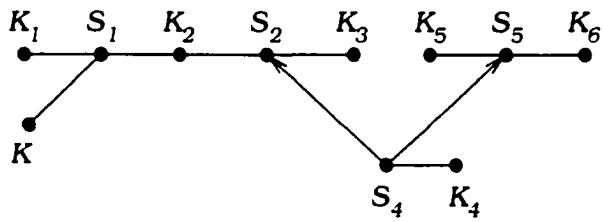
3. Create clique node $K = S \cup \{v_1, v_2\}$. For each $i = 1, 2$, do the appropriate case:



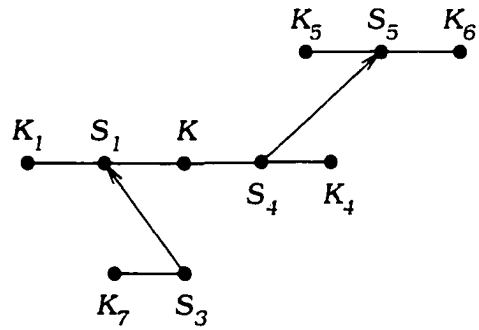
G



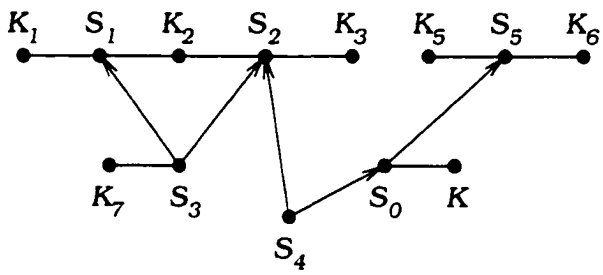
$\mathcal{G}$



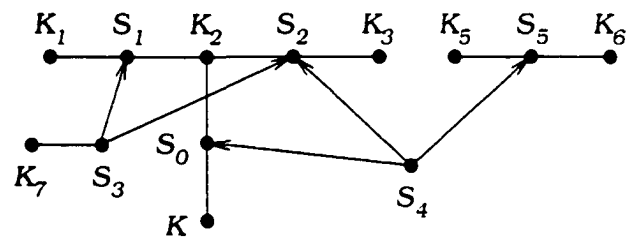
(a)



(b)



(c)



(d)

Figure 3.20: Insertion examples.

Case 1: $|K_i - S| = 1$. If K_i has a neighbor $S' \neq S$ in $\mathcal{G}$, then create edge $\{S', K\}$. Delete K_i.

Case 2: $|K_i - S| > 1$. Let $S_i = S \cup \{v_i\}$. Decide whether $S_i \in \mathcal{G}$, which is true if and only if (S, S_i) is an arc of $\mathcal{G}$. If $S_i \in \mathcal{G}$, create edge $\{S_i, K\}$. If $S_i \notin \mathcal{G}$, create S_i and edge $\{S_i, K\}$ and execute substeps (a) and (b) to compute the other edges and arcs incident to S_i.

(a) [Arcs to S_i.] Create arc (S, S_i). Find the chain C of predecessors of K_i such that S is not on C, which can be done by finding all predecessors of K_i and then eliminating all predecessors and successors of S (Figure 3.21). Find the lowest node S^l on C such that $v_i \in S^l$. If C does not exist or S^l does not exist or $S^l \not\subset S_i$, then go to (b). Otherwise, use binary search to find the highest node S^h on C such that $S^h \subset S_i$; create arc (S^h, S_i) and if $\{S^h, K_i\}$ is an edge of $\mathcal{G}$, delete $\{S^h, K_i\}$.

(b) [Arcs and edges from S_i.] Compute the set $\mathcal{S}$ of minimal nodes of $\{X \mid X$ is a node of $\mathcal{G}$ such that $S_i \subset X\}$ by examining the successors of S in topological order, where each edge is viewed as an arc, as before. If $\mathcal{S}$ contains a clique node, create edge $\{S_i, K_i\}$ and delete $\{S, K_i\}$ if it is an edge of $\mathcal{G}$. Otherwise, for every separator node S' in $\mathcal{S}$, create arc (S_i, S') and delete (S, S') if it is an arc of $\mathcal{G}$. (Lemma 3.21 will show that if $\mathcal{S}$ contains a clique node, then $\mathcal{S} = \{K_i\}$.)

4. Suppose $components_{\mathcal{G}}(S) > 2$. If $|K_1 - S| = 1$ and $|K_2 - S| = 1$, create edge $\{S, K\}$. Suppose $components_{\mathcal{G}}(S) = 2$. For every immediate predecessor S^p of S, do the following and then delete S.

   If $S_1 \in \mathcal{G}'$, create arc (S^p, S_1), and if $S_2 \in \mathcal{G}'$, create arc (S^p, S_2). If $S_1 \notin \mathcal{G}'$ and $S_2 \notin \mathcal{G}'$, create edge $\{S^p, K\}$ unless K has a neighbor $S' \neq S$ in $\mathcal{G}'$ such that $S^p \subset S'$.

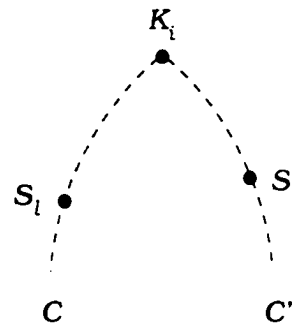


Figure 3.21: The *Insert* algorithm.

3.6.2 Insert correctness

Lemma 3.21 *The Insert operation computes $\mathcal{G}'$ in $O(n \log n)$ time.*

Proof Suppose v_1, v_2 are in distinct connected components G_1, G_2 of G with clique-separator graphs $\mathcal{G}_1, \mathcal{G}_2$, respectively. Then $K = \{v_1, v_2\}$ is a maximal clique and a cut edge of G', and $S_1 = \{v_1\}, S_2 = \{v_2\}$ are minimal vertex separators of G'. Suppose $S_i \notin \mathcal{G}$ and let $\mathcal{S}$ be the set of minimal nodes of $\mathcal{G}_i$ containing v_i. If v_i is contained in exactly one maximal clique K' of G_i, then $\mathcal{S} = \{K'\}$. Otherwise, the set of maximal cliques of G_i containing v_i induces a subtree T_i of a clique tree of G_i. Since v_i is contained in every minimal vertex separator corresponding to an edge of T_i, then $\mathcal{S}$ contains only separator nodes. Thus, step 1 computes the arcs and edges incident to S_i.

Suppose v_1, v_2 are in the same connected component of G. The following holds for each $i = 1, 2$. Suppose $|K_i - S| = 1$. By Lemma 3.20.3, $\{K_i, S\}$ is an edge of $\mathcal{G}$ and $K_i \notin \mathcal{G}'$. If $\{S', K_i\}$ is a edge of $\mathcal{G}$ for some $S' \neq S$, then $\{S', K\}$ is a edge of $\mathcal{G}'$ because $K_i = S \cup \{v_i\} \subset S \cup \{v_1, v_2\} = K$. Suppose $|K_i - S| > 1$. By Lemma 3.20.3 and 3.20.2, $K_i \in \mathcal{G}'$ and $S_i \in \mathcal{G}'$. Since $S_i = S \cup \{v_i\} \subset S \cup \{v_1, v_2\} = K$, then $\{S_i, K\}$ is a edge of $\mathcal{G}'$. If $S_i \notin \mathcal{G}$, substeps (a) and (b) compute the arcs and edges incident to S_i in $\mathcal{G}'$, as follows.

(a) Suppose there exists a separator node $\bar{S} \in \mathcal{G}$ such that $\bar{S} \not\subset S$ and $\bar{S} \subset S_i$. Since $S_i = S \cup \{v_i\}$, then $v_i \in \bar{S}$. Since $S_i \subset K_i$, then $\bar{S}$ is a predecessor of K_i. By Lemma 3.5.1, the predecessors of K_i can be partitioned into at most two chains C, C', where S is on C' (Figure 3.21). Then $\bar{S}$ must be on C. Let S^l be the lowest node on C such that $v_i \in S^l$. If $S^l \not\subset S_i$, then no node higher than S^l on C is a subset of S_i and thus $\bar{S}$ does not exist. Otherwise, let S^h be the highest node on C such that $S^h \subset S_i$. Then (S^h, S_i) is an arc of $\mathcal{G}'$. Thus, (a) finds S^h if it exists.

(b) Let $\mathcal{S}$ be the set of minimal nodes of $\mathcal{G}$ containing S_i, each of which is a successor of S. Since S has successor K_i and $S_i \subset K_i$, if there is exactly one maximal clique of G containing S_i, then $\mathcal{S} = \{K_i\}$. Otherwise, the set of maximal cliques of G containing S_i induces a subtree T of a clique tree of G. Since S_i is contained in every minimal vertex separator corresponding to an edge of T, then $\mathcal{S}$ contains only separator nodes. Thus, (b) computes the arcs and edges from S_i.

Suppose $components_{\mathcal{G}}(S) > 2$. Then $S \in \mathcal{G}'$. If $|K_i - S| > 1$, then after step 3, S_i is a node, (S, S_i) is an arc, and $\{S_i, K\}$ is an edge. If $|K_1 - S| = 1$ and $|K_2 - S| = 1$, however, then after step 3, S_1, S_2 are not nodes and K has no incident edges. Since $S \subset S \cup \{v_1, v_2\} = K$, then $\{S, K\}$ is an edge of $\mathcal{G}'$.

Suppose $components_{\mathcal{G}}(S) = 2$. Then $S \notin \mathcal{G}'$. Consider each immediate predecessor S^p of S. If $S_i \in \mathcal{G}'$, then (S^p, S_i) is an arc of $\mathcal{G}'$ because $S_i = S \cup \{v_i\} \subset S \cup \{v_1, v_2\} = K$. If $S_1 \notin \mathcal{G}'$ and $S_2 \notin \mathcal{G}'$, then $\{S^p, K\}$ is an edge of $\mathcal{G}'$ unless step 3 created a neighbor $S' \neq S$ of K such that $S^p \subset S'$.

Since $\mathcal{G}$ has size $O(n)$, then each step requires $O(n)$ time except the binary search in step 3. Testing whether one separator node is contained in another requires $O(n)$ time, so the binary search requires $O(n \log n)$ time. Hence, the *Insert* operation computes $\mathcal{G}'$ in $O(n \log n)$ time. □

3.6.3 Delete

Throughout this subsection, $\{u, v\} \in E(G)$. Then $\{u, v\}$ is contained in at least one node of $\mathcal{G}$. The next theorem is copied from Chapter 2.

Theorem 3.22 *Let G be a chordal graph and let $\{u, v\} \in E(G)$. Then $G - \{u, v\}$ is chordal if and only if G has exactly one maximal clique containing $\{u, v\}$.*

By Lemma 3.1.2, K is the unique clique node containing $\{u, v\}$ if and only if K contains $\{u, v\}$ and no neighbor of K contains $\{u, v\}$. Thus, we can readily test whether K is the unique clique node containing $\{u, v\}$. Figure 3.22 illustrates the premise of the following lemma, which specifies the clique nodes and separator nodes of $\mathcal{G}'$ except for $K^u = K - \{v\}$ and $K^v = K - \{u\}$, which may or may not be clique nodes of $\mathcal{G}'$.

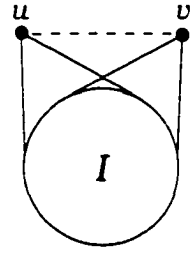


Figure 3.22: Maximal clique K.

Lemma 3.23 *Let G be a chordal graph with clique-separator graph $\mathcal{G}$ and let $\{u, v\} \in E(G)$. Suppose $G' = G - \{u, v\}$ is chordal with clique-separator graph $\mathcal{G}'$. Let K be the unique clique node of $\mathcal{G}$ containing $\{u, v\}$. Let $I = K^u \cap K^v = K - \{u, v\}$.*

1. *Every clique node $K' \neq K$ of G is a clique node of G'.*
2. *If $I \neq \emptyset$, then I is a separator node of $\mathcal{G}'$.*
3. *If separator node S of $\mathcal{G}$ is not a neighbor of K, then S is a separator node of $\mathcal{G}'$.*

4. *If separator node S of $\mathcal{G}$ is a neighbor of K, the following holds.*

   (a) If S contains neither of u, v, then S is a separator node of $\mathcal{G}'$.

   (b) If S contains exactly one of u, v and $|K - S| > 1$, then S is a separator node of $\mathcal{G}'$.

   (c) If S contains exactly one of u, v and $|K - S| = 1$, then S is a separator node of $\mathcal{G}'$ if and only if components$_{\mathcal{G}}(S) > 2$.

Proof

1. By Section 2.3.2.

2. Observe that in a chordal graph with edge e, a shortest cycle containing e is a triangle. (The converse is not true, as shown by the graph formed by identifying each edge of a square with an edge of a triangle.) We claim $\{u, v\}$ is a cut edge of $G - I$. Suppose otherwise. Then $\{u, v\}$ is contained in a cycle of $G - I$. Since $G - I$ is chordal, then the shortest cycle C containing $\{u, v\}$ is a triangle. Then C is $\{u, v, w\}$ for some $w \in V(G) - I$. Since $w \notin K$, then C is contained in some maximal clique $K' \neq K$ of G. Then K, K' are maximal cliques of G containing u, v, a contradiction. Thus, $\{u, v\}$ is a cut edge of $G - I$, which means I is a uv-separator of $G - \{u, v\}$. Since every vertex of I is adjacent to u and v, then I is a minimal uv-separator of $G - \{u, v\}$, i.e., I is a minimal vertex separator of G'. (This result is implicit in the *Delete* operation in Section 2.3.2.)

3. By Corollary 3.3.2 and 3.3.3, $\mathcal{G}$ contains clique nodes K_1, K_2 such that S divides K_1, K_2 and $S = K_1 \cap K_2$ and S is a minimal xy-separator in G for any $x \in K_1 - S$ and $y \in K_2 - S$ (Figure 3.23a). If neither of K_1, K_2 is K, then neither of K_1, K_2 contains edge $\{u, v\}$, which implies S is a minimal vertex separator of G'. Suppose otherwise, say $K_1 = K$. Since $S \subset K$, then $S \rightsquigarrow S'$ for some neighbor S' of K. By Corollary 3.3.2 and 3.3.3, $\mathcal{G}$ contains a clique node $K' \neq K$ such that S' divides K, K' and $S' = K \cap K'$ (Figure 3.23b). Then K, K' are in the same connected component

of $\mathcal{G} - Preds(S)$ (because S' is a neighbor of K and S' is either a neighbor of K' or a predecessor of a neighbor of K'). Since S divides K, K_2, then S divides K', K_2. Since $S \subset K', K_2$, then by Corollary 3.3.3, $S = K' \cap K_2$ and S is a minimal xy-separator in G for any $x \in K' - S$ and $y \in K_2 - S$. Since neither of K', K_2 contains edge $\{u, v\}$, then S is a minimal vertex separator of G'.

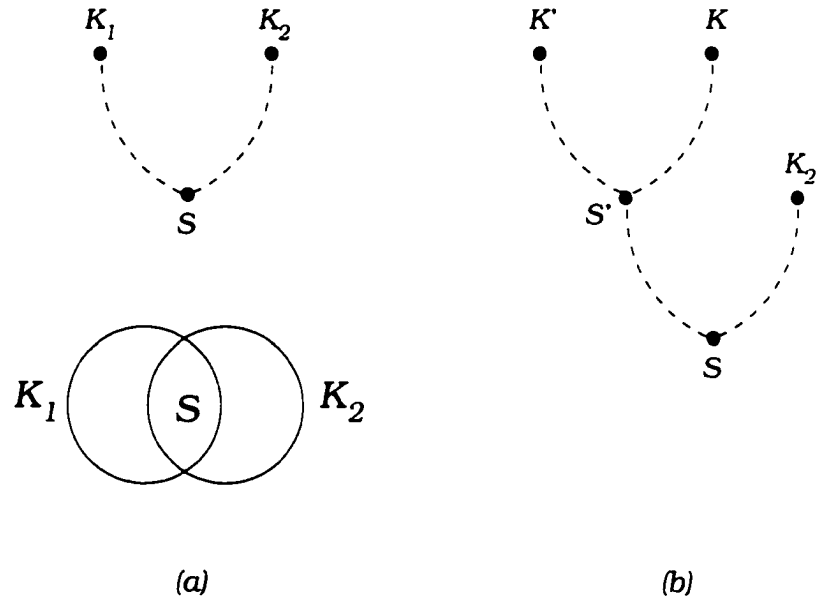


Figure 3.23: Proof of part 2.

4. We have two main cases, which depend on the value of $components_{\mathcal{G}}(S)$. Suppose $components_{\mathcal{G}}(S) > 2$. By Corollary 3.3.3, $\mathcal{G}$ contains clique nodes $K', K'' \neq K$ such that $S = K' \cap K''$ and S is a minimal xy-separator for any $x \in K' - S$ and $y \in K'' - S$ in G. Since neither of K', K'' contains edge $\{u, v\}$, then S is a minimal vertex separator of G', regardless of whether S contains neither or one of u, v.

Suppose $components_{\mathcal{G}}(S) = 2$. By Corollary 3.3.4, S has one or more immediate successors or exactly one neighbor $K' \neq K$, but not both. Then $\mathcal{G} - Preds(S)$ has exactly two connected components with a clique node containing S: one component contains K and the other component contains clique nodes $K_1, \ldots, K_j$, where $j > 1$ if S has one or more immediate successors, and $j = 1$ if S has a neighbor $K' \neq K$.

Furthermore, K and $K_1, \ldots, K_j$ are all of the clique nodes of $\mathcal{G}$ containing S. Since S divides K and $K_1, \ldots, K_j$, then S separates $K - S$ and $(K_1 \cup \cdots \cup K_j) - S$. Let V_1, V_2 be the vertex sets of the two connected components of $G - S$ with a clique node containing S, where V_1 contains the vertices in $K - S$ and V_2 contains the vertices in $(K_1 \cup \cdots \cup K_j) - S$. Let G_1, G_2 be the subgraphs of G induced by $V_1 \cup S, V_2 \cup S$, respectively.

Assume S contains neither of u, v (Figure 3.24a). Let $\bar{K}$ be the maximal clique of G' containing K^u; we may have $\bar{K} = K^u$. Since G_1 contains the vertices in $K^u - S$, then G_1 contains $\bar{K}$. Then for any $1 \leq i \leq j$, $S = \bar{K} \cap K_i$ and S separates $\bar{K} - S$ and $K_i - S$ in G', which means S is a minimal xy-separator of G' for any $x \in \bar{K} - S$ and $y \in K_i - S$. Thus, S is a minimal vertex separator of G'.

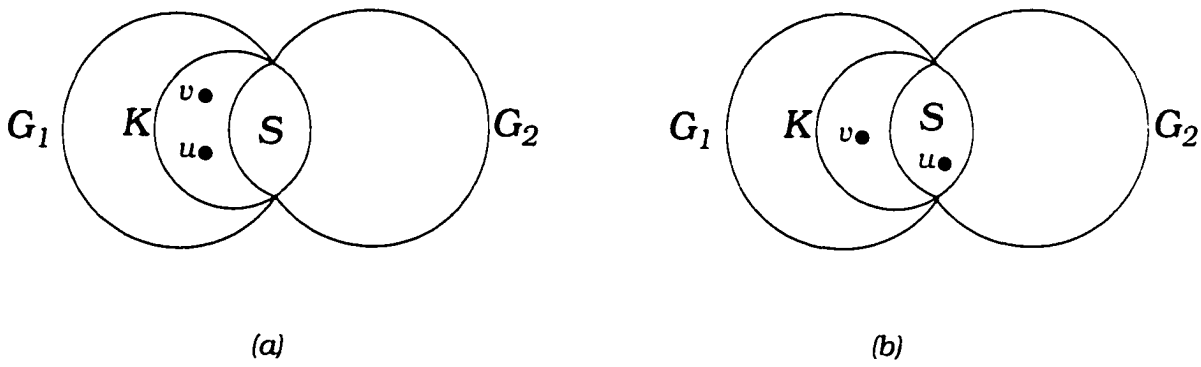


Figure 3.24: Proof of part 3.

Assume S contains exactly one of u, v, say u (Figure 3.24b). Suppose $|K - S| > 1$. Then $K - S$ contains a vertex $w \neq v$ and $\{w\} \cup S$ is contained in some maximal clique $\bar{K}$ of G'. Since G_1 contains w, then G_1 contains $\bar{K}$. Then as before, $S = \bar{K} \cap K_i$ and S separates $\bar{K} - S$ and $K_i - S$ in G' for any $1 \leq i \leq j$. Thus, S is a minimal vertex separator of G'. Suppose $|K - S| = 1$. Since K is the only maximal clique of G_1 that contains S, then v is the only vertex of G_1 that is adjacent to every vertex of S. It follows that in G', there are no two maximal cliques $\bar{K}_1, \bar{K}_2$ such that $S = \bar{K}_1 \cap \bar{K}_2$ and S separates $\bar{K}_1 - S$ and $\bar{K}_2 - S$. By Theorem 1.1, S is not a minimal vertex

separator of G'. □

The $Delete(u, v)$ operation computes $\mathcal{G}'$ as follows. Let K be the unique clique node containing u, v. The operation has two main cases. In the first, K is a leaf with neighbor S, and in the second, K is an internal node with neighbors S_1, S_2. Each case has several subcases that depend on whether u, v are contained in a neighbor of K. (Recall that u, v are not contained in the same neighbor of K, or else K is not unique.) The operation uses two subroutines: $GetMax(S, u)$ is a function that returns the set of maximal predecessors of S that do not contain vertex $u \in S$, and $Remove(S, K)$ deletes separator node S, which has neighbor K, and updates the clique-separator graph accordingly.

After $\mathcal{G}'$ is computed, if any box with three or more nodes is not a caterpillar, or any separator node has indegree greater than 2, then the operation rejects because G' is not interval by the lemmas in Section 3. If $\mathcal{G}'$ satisfies the lemmas and yet G' is not interval, then the train tree algorithm, which computes a spanning set of train trees of $\mathcal{G}'$, rejects the deletion.

$Delete(u, v)$:

(Assume K is the unique clique node containing u, v.)

Case one: K is a leaf with neighbor S

Figure 3.25 illustrates Cases 1–4, and Figure 3.26 shows examples of Cases 1–3.

- $u \notin S, v \notin S$.

  Case 1. $|K - S| = 2$, equivalently, $S = I$ Figure 3.25a). Create nodes K^u, K^v and edges $\{K^u, S\}, \{S, K^v\}$.

  Case 2. $|K - S| > 2$, equivalently, $S \subset I$ Figure 3.25b). Create nodes I, K^u, K^v, edges $\{K^u, I\}, \{I, K^v\}$, and arc (S, I).

- $u \in S, v \notin S$.

Case 3. $|K - S| > 1$, equivalently, $I \not\subset S$ and $S \not\subset I$ Figure 3.25c). Create nodes I, K^u, K^v and edges $\{S, K^u\}, \{K^u, I\}, \{I, K^v\}$. For each $S' \in Get.Max(S, u)$, create arc (S', I).

Case 4. $|K - S| = 1$, equivalently, $S = K^u = I \cup \{u\}$ Figure 3.25d). Then $I \in \mathcal{G}$ if and only if (I, S) is an arc of $\mathcal{G}$.

$I \in \mathcal{G}$: Create node K^v and edge $\{I, K^v\}$. If $components_{\mathcal{G}}(S) = 2$, call $Remove(S, K)$.

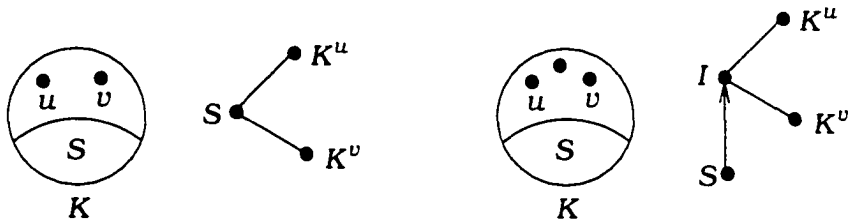
$I \notin \mathcal{G}$: Create nodes I, K^v and edge $\{I, K^v\}$. Compute the arcs incident to I as follows.

1. [Arcs to I.] For each $S' \in Get.Max(S, u)$, create arc (S', I) and if (S', S) is an arc, delete (S', S).

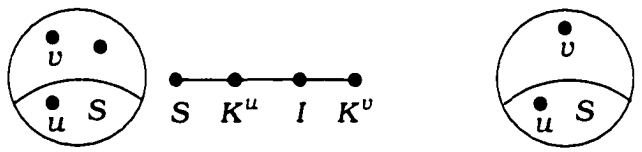
2. [Arcs from I.] If S is not an offspring of any Q-node, then let $\mathcal{S} = \{S\}$. Otherwise, S is an offspring of a Q-node x. Find the leftmost and rightmost offspring of x containing I by applying binary search to the offspring of x: the intervening offspring contain I. Compute the minimal elements of the set of intervening offspring as follows. (a) Every intervening offspring has an integer value, initially 0. Mark any offspring whose value is subsequently changed. (Recall every offspring of a Q-node has $left, right$ pointers.) (b) For every intervening offspring l, add $+1$ to the value of $left(l)$ and -1 to the value of the offspring on the immediate left of l, and add $+1$ to the value of the offspring on the immediate right of l and -1 to the value of $right(l)$. (c) For each intervening offspring l, compute its left to right prefix sum and assign the sum to the value of l.

Let $\mathcal{S}$ be the set of intervening offspring that are unmarked and then unmark all marked offspring. For each separator node $S' \in \mathcal{S}$, create arc (I, S').

Delete K and any edges or arcs incident to K.



(a) Case 1 *(b) Case 2*



(c) Case 3 *(d) Case 4*

Figure 3.25: K is a leaf.

Case two: K is an internal node with neighbors S_1, S_2.

Figure 3.27 illustrates Cases 1–5.

- $u, v \notin S_1 \cup S_2$ (Figure 3.27a). Create nodes I, K^u, K^v, edges $\{I, K^u\}, \{I, K^v\}$, and arcs $(S_1, I), (S_2, I)$.

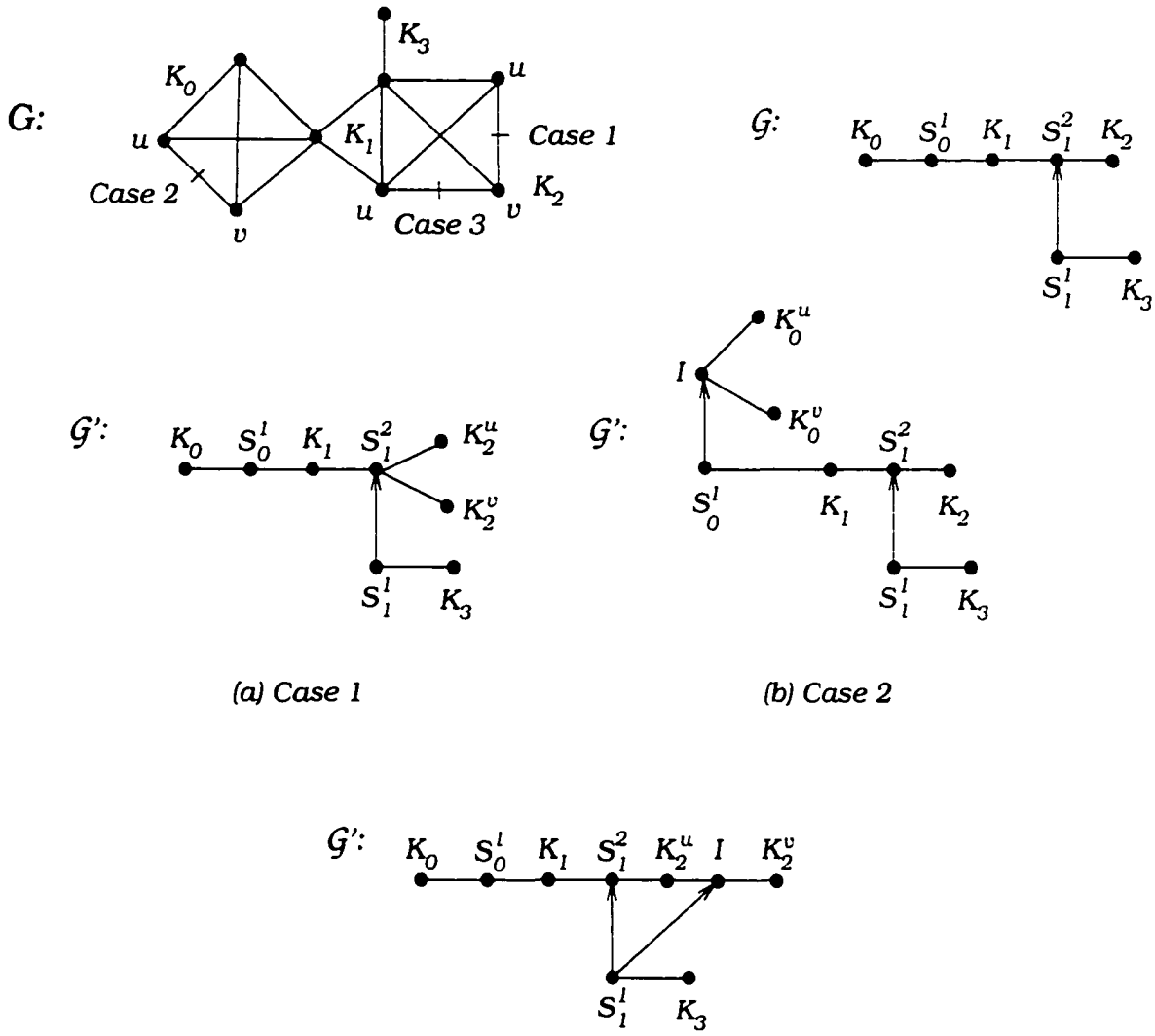
- $u \in S_1 \cap S_2, v \notin S_1 \cup S_2$ (Figure 3.27b). Reject the deletion.

- $u \in S_1 - S_2, v \notin S_1 \cup S_2$ (Figure 3.27c). Create clique nodes K^u, K^v.

  Case 1. $|K-(S_1 \cup S_2)| > 1$ or $|S_1 - S_2| > 1$. Create node I, edges $\{S_1, K^u\}, \{K^u, I\}, \{I, K^v\}$ and arc (S_2, I).

  Case 2. $|K-(S_1 \cup S_2)| = 1$ and $|S_1 - S_2| = 1$. Create edges $\{S_1, K^u\}, \{K^u, S_2\}, \{S_2, K^v\}$.

- $u \in S_1 - S_2, v \in S_2 - S_1$.



(a) Case 1 *(b) Case 2*

(c) Case 3

Figure 3.26: Deletion examples.

Case 3. $|K - (S_1 \cup S_2)| > 0$ (Figure 3.27d). Create nodes I, K^u, K^v and edges $\{S_1, K^u\}, \{K^u, I\}, \{I, K^v\}, \{K^v, S_2\}$. Compute $\mathcal{S}_1 = Get.Max(S_1, u)$ and $\mathcal{S}_2 = Get.Max(S_2, v)$. For each $S' \in \mathcal{S}_1$, create arc (S', I). For each $S' \in \mathcal{S}_2 - \mathcal{S}_1$, create arc (S', I).

Case 4. $|K - (S_1 \cup S_2)| = 0$ and $|K| > 2$ (Figure 3.27e). If $|S_2 - S_1| = 1$, test whether $I \in Get.Max(S_1, u)$. If $|S_1 - S_2| = 1$, test whether $I \in Get.Max(S_2, v)$. If no test succeeds, then create node I and if $I \subset S_1$ (resp. $I \subset S_2$), then create arc (I, S_1) (resp. (I, S_2)).

If $|S_2 - S_1| > 1$, then create node K^u and edges $\{S_1, K^u\}, \{K^u, I\}$, and otherwise, if $components_{\mathcal{G}}(S_1) = 2$, then $Remove(S_1, K)$. Symmetrically, if $|S_1 - S_2| > 1$, then create node K^v and edges $\{S_2, K^v\}, \{K^v, I\}$, and otherwise, if $components_{\mathcal{G}}(S_2) = 2$, then $Remove(S_2, K)$.

Case 5. $|K - (S_1 \cup S_2)| = 0$ and $|K| = 2$ (Figure 3.27f). Lemma 3.24 shows $\{u, v\}$ is a cut edge of G and K is a cut node of $\mathcal{G}$. Delete K from $\mathcal{G}$ to obtain $\mathcal{G}_1, \mathcal{G}_2$. If $components_{\mathcal{G}}(S_i) = 2$, then call $Remove(S_i, K), i = 1, 2$.

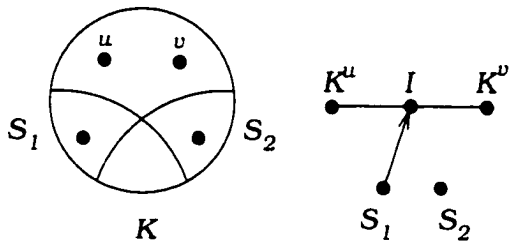
Delete K and any edges or arcs incident to K.

$Get.Max(S, u)$:

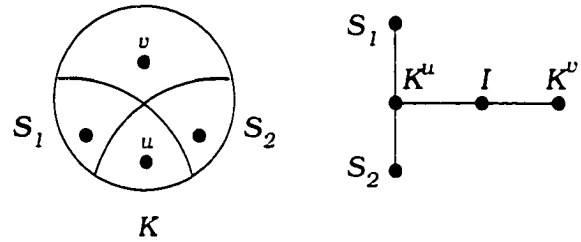
(Assume $u \in S$.) Compute the set $\mathcal{S}$ of maximal nodes of $\{S' \mid S'$ is a separator node of $\mathcal{G}$ such that $S' \subset S - \{u\}\}$ by examining the predecessors of S in reverse topological order. Return $\mathcal{S}$.

$Remove(S, K)$:

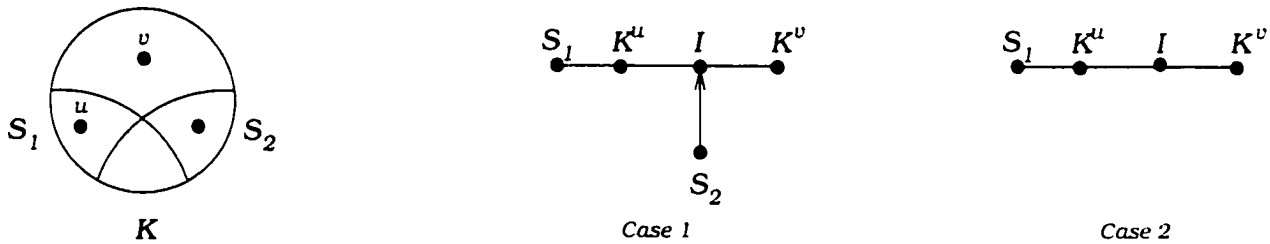
(Assume $\{S, K\}$ is an edge of $\mathcal{G}$.) For each immediate predecessor S^p of S, do the following. (a) Mark all successors of S^p. (b) If S has some immediate successor(s), then for each arc (S, S^s), create arc (S^p, S^s) unless S^s has a marked immediate predecessor $\bar{S} \neq S$ (Figure 3.28). If S has a neighbor $K' \neq K$ in $\mathcal{G}$, then create edge $\{S^p, K'\}$ unless K' has a marked neighbor $S' \neq S$. (c) Unmark all successors of S^p.



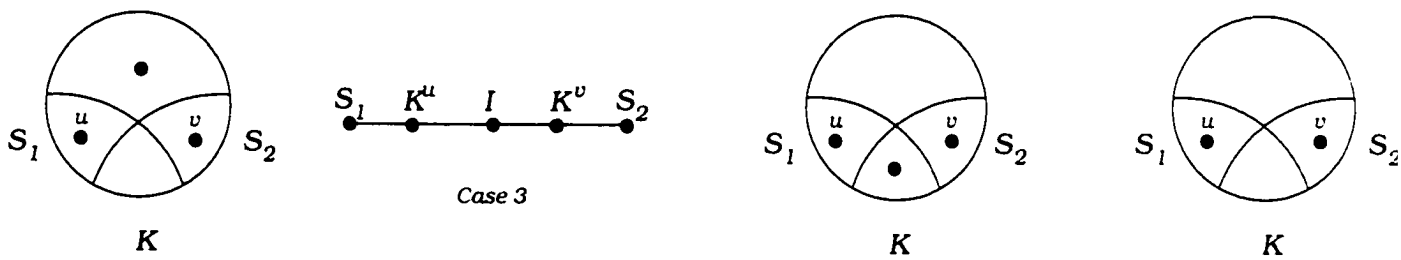
(a) u, v not in S_1 ∪ S_2



(b) u in S_1 ∩ S_2, v not in S_1 ∪ S_2



(c) u in S_1 - S_2, v not in S_1 ∪ S_2



(d) u in S_1 - S_2, v in S_2 - S_1

Figure 3.27: K is not a leaf.

Delete S and any edges or arcs incident to S.

3.6.4 Delete correctness

Lemma 3.24 *The Delete operation computes $\mathcal{G}'$ in $O(n \log n)$ time.*

Proof Although I may not be a node of $\mathcal{G}$, I is a node of $\mathcal{G}'$ by Lemma 3.23.2. We will show how to decide whether $I \in \mathcal{G}$ and whether $K^u, K^v \in \mathcal{G}'$.

Observation 1. If K^u (resp. K^v) contains a vertex not contained in any neighbor of K, then $K^u \in \mathcal{G}'$ (resp. $K^v \in \mathcal{G}'$). If I contains a vertex not contained in any neighbor of K, then $I \notin \mathcal{G}$ and I has no successors in $\mathcal{G}'$. (By Lemma 3.1.4, K is the only node of $\mathcal{G}$ containing a vertex not contained in any neighbor of K.)

Observation 2. If $K^u \in \mathcal{G}'$ (resp. $K^v \in \mathcal{G}'$), then $\{K^u, I\}$ (resp. $\{K^v, I\}$) is an edge of $\mathcal{G}'$.

Case one: K is a leaf with neighbor S

By Lemma 3.23.3, every separator node of $\mathcal{G}$ is a separator node of $\mathcal{G}'$, except possibly for S. By Corollary 3.3.2 and 3.3.3, there is a clique node K' of $\mathcal{G}$ such that $S = K' \cap K$ and S divides K', K in $\mathcal{G}$. Recall that since $\{S, K\}$ is an edge of $\mathcal{G}$, there is no separator node S' of $\mathcal{G}$ such that $S \subset S' \subset K$.

• $u \notin S, v \notin S$. Then $S \subseteq I$. By Observation 1, $K^u, K^v \in \mathcal{G}'$ and by Observation 2, $\{K^u, I\}, \{I, K^v\}$ are edges of $\mathcal{G}'$.

Case 1: $|K - S| = 2$, equivalently, $S = I$ (Figure 3.25a). Then S has the same incident arcs in $\mathcal{G}$ and $\mathcal{G}'$ and we are done.

Case 2: $|K - S| > 2$, equivalently, $S \subset I$ (Figure 3.25b). By Observation 1, $I \notin \mathcal{G}$ and there are no arcs from I in $\mathcal{G}'$. By Lemma 3.23.4, $S \in \mathcal{G}'$ and thus (S, I) is an

arc of $\mathcal{G}'$. There are no other arcs to I because $I - S \subset K - S$ and for any separator node $S' \neq S$ of $\mathcal{G}$, $S' \subset I$ implies $S' \subset S$.

- $u \in S, v \notin S$.

Case 3: $|K - S| > 1$, equivalently, $I \not\subset S$ and $S \not\subset I$ (Figure 3.25c). By Observations 1 and 2, $K^u, K^v \in \mathcal{G}'$ and $\{K^u, I\}, \{I, K^v\}$ are edges of $\mathcal{G}'$. By Lemma 3.23.4, $S \in \mathcal{G}'$ and thus $\{S, K^u\}$ is an edge of $\mathcal{G}'$. By Observation 1, $I \notin \mathcal{G}$ and there are no arcs from I in $\mathcal{G}'$, but there may be arcs to I. Since $I - S \subset K - S$, then for any separator node S' of $\mathcal{G}$, $S' \subset I$ implies $S' \subset S - \{u\}$. Therefore, $GetMax(S, u)$ computes the arcs to I in $\mathcal{G}'$.

Case 4: $|K - S| = 1$, equivalently, $S = I \cup \{u\} = K^u$ (Figure 3.25d). This is the most complicated case. Since $K^u = S \subset K'$, then $K^u \notin \mathcal{G}'$. By Observations 1 and 2, $K^v \in \mathcal{G}'$ and $\{I, K^v\}$ is an edge of $\mathcal{G}'$. We decide whether $I \in \mathcal{G}$ and $S \in \mathcal{G}'$ as follows. Since $I = S - \{u\}$, then $I \in \mathcal{G}$ if and only if (I, S) is an arc of $\mathcal{G}$. By Lemma 3.23.4, $S \in \mathcal{G}'$ if and only if $components_{\mathcal{G}}(S) > 2$.

The subcases will depend on whether $I \in \mathcal{G}$ and $S \in \mathcal{G}'$. Figure 3.29 gives examples of the various subcases. One difficulty is that if $(S^p, S), (S, S^s)$ are arcs of $\mathcal{G}$ and $S \notin \mathcal{G}'$, then (S^p, S^s) is an arc of $\mathcal{G}'$ unless there is a node $\bar{S} \neq S$ such that $(\bar{S}, S^s)$ is an arc and $S^p \rightsquigarrow \bar{S}$ in $\mathcal{G}$ (Figure 3.28). This will require a graph search in $Remove(S, K)$ to find all successors of S^p.

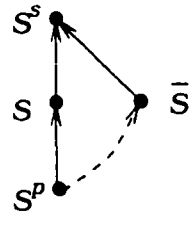


Figure 3.28: Deleting S.

First, suppose $I \in \mathcal{G}$. Then (I, S) is an arc of $\mathcal{G}$. If $S \in \mathcal{G}'$ (Figure 3.29a), then (I, S) is an arc of $\mathcal{G}'$ and we are done. If $S \notin \mathcal{G}'$ (Figure 3.29b), then $components_{\mathcal{G}}(S) =$

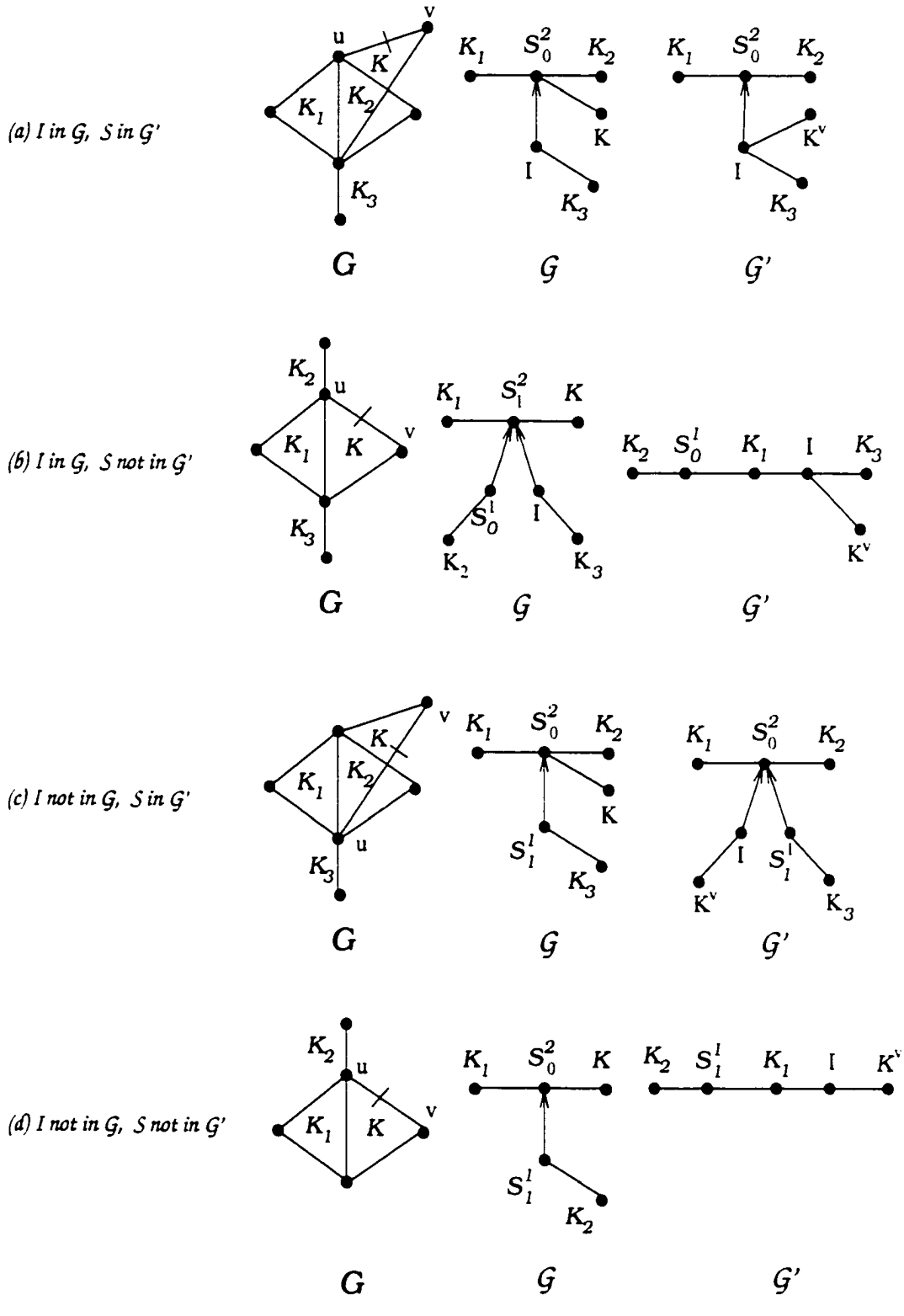


Figure 3.29: More examples.

2 and by Corollary 3.3.4, S has one or more immediate successors or exactly one neighbor $K' \neq K$, but not both. Then $Remove(S, K)$ deletes S and computes the new arcs of $\mathcal{G}'$.

Second, suppose $I \notin \mathcal{G}$. Assume $S \in \mathcal{G}'$ (Figure 3.29c). Then $\mathcal{G}'$ is the graph formed by adding I to $\mathcal{G}$ and updating its edges and arcs as required by the clique-separator graph definition. Consider the immediate predecessors of I in $\mathcal{G}'$. Since $I = S - \{u\}$, then step 1 of Case 4 computes the arcs to I in $\mathcal{G}'$. Now consider the immediate successors and neighbors of I in $\mathcal{G}'$, which are the minimal elements of $\mathcal{G}$ containing I, equivalently, the minimal elements of $leaves(T)$ containing I.

Claim Assume $S \in \mathcal{G}'$. If S is an offspring of Q-node x of T, then every minimal element of $leaves(T)$ containing I is an offspring of x. Otherwise, S is the unique minimal element of $leaves(T)$ containing I.

Proof Since $I = S - \{u\}$, then S is a minimal element of $leaves(T)$ containing I. Consequently, if S is an offspring of Q-node x, then by Corollary 3.12.2, any other minimal element of $leaves(T)$ containing I is also an offspring of x. Otherwise, S is a child of a P-node x such that x is the root of T or x has a P-node parent y. In the first case, S is the only minimal element of $leaves(T)$ containing I. In the second case, if $S' \neq S$ is a minimal element of $leaves(T)$ containing I, then $S' \in leaves(T) - leaves(x)$ and therefore $I \subseteq S_y$, a contradiction. Hence, S is the unique minimal element of $leaves(T)$ containing I. □

Suppose I has a neighbor K' in $\mathcal{G}'$. By the claim, K' and S must be offspring of a Q-node x. By train tree property 5a, both siblings of K' are S-nodes: the sibling of K' nearer to S is some S-node y. By the clique intersection property, $I \subset S_y$, and by train tree property 6, $S_y \subset K'$, which implies I is not a neighbor of K' in $\mathcal{G}'$, a contradiction. Hence, I has no neighbors in $\mathcal{G}'$.

We now show that steps 2 and 3 of Case 4 compute the arcs from I in $\mathcal{G}'$. If S is not an offspring of any Q-node, then by the claim, S is the unique minimal element of $leaves(T)$ containing I. Otherwise, S is an offspring of Q-node x. Then step 2 computes the leftmost and rightmost offspring of x containing I and by the claim, this range contains every minimal element of $leaves(T)$ containing I. Let S' be an offspring of x. If S' is not a minimal element of $leaves(T)$ containing I, then some offspring S'' of x contains I and $S'' \subset S'$, which implies $left(S'')$ or $right(S'')$ points "over" S'. Conversely, if S'' contains I and $left(S'')$ or $right(S'')$ points "over" S', then $S'' \subset S'$ and S' is not a minimal element of $leaves(T)$ containing I. Thus, the set $\mathcal{S}$ computed by step 2 is exactly the set of immediate successors of I in $\mathcal{G}'$ and this set includes S. Then step 2 creates the arcs from I in $\mathcal{G}'$. Notice that there may be three or more arcs to some separator node in $\mathcal{G}'$, in which case the operation rejects.

Assume $S \notin \mathcal{G}'$ (Figure 3.29d). Let $\mathcal{G}''$ be the graph computed when $S \in \mathcal{G}'$. Then deleting S from $\mathcal{G}''$ and updating $\mathcal{G}''$ yields $\mathcal{G}'$. It follows that $Remove(S, K)$ correctly computes $\mathcal{G}'$.

Since $GetMax$ and $Remove$ each require $O(n)$ time, the running time is dominated by the $O(n \log n)$ time to do the binary search when $I \notin \mathcal{G}$.

Case two: K is an internal node with neighbors S_1, S_2.

Since S_1, S_2 are incomparable, then $|S_2 - S_1| > 0$ and $|S_1 - S_2| > 0$. By Corollary 3.3.2 and 3.3.3, there are clique nodes K_1, K_2 of $\mathcal{G}$ such that $S_1 = K_1 \cap K$ and S_1 divides K_1, K in $\mathcal{G}$ and $S_2 = K_2 \cap K$ and S_2 divides K_2, K in $\mathcal{G}$. We again have several cases and in every case except the last, $I \neq \emptyset$ and thus $I \in \mathcal{G}'$ by Lemma 3.23.2. Recall that since $\{S_1, K\}, \{S_2, K\}$ are edges of $\mathcal{G}$, there is no separator node S' of $\mathcal{G}$ such that $S_1 \subset S' \subset K$ or $S_2 \subset S' \subset K$. Also recall that by Lemma 3.1.3, if S' is a separator node of $\mathcal{G}$ such that $S' \cap K$, then $S' \subseteq S_1$ or $S' \subseteq S_2$. Figure 3.27 illustrates the various cases.

• $u, v \notin S_1 \cup S_2$ (Figure 3.27a). By Observations 1 and 2, $K^u, K^v \in \mathcal{G}'$ and $\{I, K^u\}, \{I, K^v\}$ are edges of $\mathcal{G}'$. By Lemma 3.23.4, $S_1, S_2 \in \mathcal{G}'$. Since $S_1, S_2 \subseteq I$, then $(S_1, I), (S_2, I)$ are arcs of $\mathcal{G}'$. By Lemma 3.1.3, $I \notin \mathcal{G}$ and there are no other arcs to I and no arcs from I.

• $u \in S_1 \cap S_2, v \notin S_1 \cup S_2$ (Figure 3.27b). By Observation 1, $K^v \in \mathcal{G}'$. If $|K - (S_1 \cup S_2)| > 1$, then by Observation 1, $K^u \in \mathcal{G}'$. Otherwise, $|K - (S_1 \cup S_2)| = 1$. By Lemma 3.1.2, $K^u \in \mathcal{G}'$. (If $K^u \notin \mathcal{G}'$, then $K^u = S_1 \cup S_2 \subset K'$ for some clique node $K' \neq K$ of $\mathcal{G}$, a contradiction.) By Observation 2, $\{K^u, I\}, \{I, K^v\}$ are edges of $\mathcal{G}'$. By Lemma 3.23.4, $S_1, S_2 \in \mathcal{G}'$. Since $\{S_1, S_2, I\}$ is an antichain, then $\{S_1, K^u\}, \{K^u, S_2\}$ are edges of $\mathcal{G}'$. Since K^u has three neighbors in $\mathcal{G}'$, then G' is not interval, so the algorithm rejects. (There may be arcs to I but by Lemma 3.1.3, $I \notin \mathcal{G}$ and there are no arcs from I.)

• $u \in S_1 - S_2, v \notin S_1 \cup S_2$ (Figure 3.27c). By Observation 1, $K^v \in \mathcal{G}'$.

Case 1: $|K - (S_1 \cup S_2)| > 1$ or $|S_1 - S_2| > 1$. Then $S_2 \subset I$. If $|K - (S_1 \cup S_2)| > 1$, then $K^u \in \mathcal{G}'$ by Observation 1. Otherwise, $K^u = S_1 \cup S_2$ and by Lemma 3.1.2, $K^u \in \mathcal{G}'$.

Case 2: $|K - (S_1 \cup S_2)| = 1$ and $|S_1 - S_2| = 1$. Then $S_2 = I$ and $K^u = S_1 \cup S_2$ and by Lemma 3.1.2, $K^u \in \mathcal{G}'$.

By Observation 2, $\{K^u, I\}, \{I, K^v\}$ are edges of $\mathcal{G}'$. By Lemma 3.23.4, $S_1, S_2 \in \mathcal{G}'$. Then $\{S_1, K^u\}$ is an edge of $\mathcal{G}'$. In Case 1, (S_2, I) is an arc of $\mathcal{G}'$ and by Lemma 3.1.3, $I \notin \mathcal{G}$ and there are no arcs from I in $\mathcal{G}'$. In Case 2, $S_2 = I$ and the arcs to and from I are the same in $\mathcal{G}$ and $\mathcal{G}'$.

• $u \in S_1 - S_2, v \in S_2 - S_1$.

Case 3: $|K - (S_1 \cup S_2)| > 0$ (Figure 3.27d). By Observation 1, $K^u, K^v \in \mathcal{G}'$ and $\{K^u, I\}, \{I, K^v\}$ are edges of $\mathcal{G}'$. By Lemma 3.23.4, $S_1, S_2 \in \mathcal{G}'$. Since $\{S_1, S_2, I\}$ is

an antichain, then $\{S_1, K^u\}, \{K^v, S_2\}$ are edges of $\mathcal{G}'$. By Observation 1, $I \notin \mathcal{G}$ and there are no arcs from I in $\mathcal{G}'$. Let $\mathcal{S}_1 = GetMax(S_1, u)$ and $\mathcal{S}_2 = GetMax(S_2, v)$. If $S' \in \mathcal{S}_1$ or $S' \in \mathcal{S}_2$, then (S', I) is an arc of $\mathcal{G}'$.

Case 4: $|K - (S_1 \cup S_2)| = 0$ and $|K| > 2$ (Figure 3.27e). This is the most complicated case. By Lemma 3.1.3, $I \in \mathcal{G}$ only if $I \subset S_1$ or $I \subset S_2$, which holds only if $|S_2 - S_1| = 1$ or $|S_1 - S_2| = 1$. Thus, if $|S_2 - S_1| = 1$ or $|S_1 - S_2| = 1$, we can decide whether $I \in \mathcal{G}$ by examining $GetMax(S_1, u)$ and $GetMax(S_2, v)$.

By Lemma 3.23.4, if $|S_2 - S_1| > 1$, then $S_1 \in \mathcal{G}'$, and if $|S_2 - S_1| = 1$, then $S_1 \in \mathcal{G}'$ if and only if $components_{\mathcal{G}}(S_1) > 2$. Also, $|S_2 - S_1| = 1$ if and only if $K^u = S_1$ if and only if $K^u \notin \mathcal{G}'$. In other words, if $|S_2 - S_1| > 1$, then $S_1, K^u \in \mathcal{G}'$ and $\{S_1, K^u\}, \{K^u, I\}$ are edges of $\mathcal{G}'$, and otherwise, $K^u \notin \mathcal{G}'$, and $S_1 \in \mathcal{G}'$ if and only if $components_{\mathcal{G}}(S_1) > 2$. The other case is symmetric.

Case 5: $|K-(S_1 \cup S_2)| = 0$ and $|K| = 2$ (Figure 3.27f). Then $I = \emptyset$ and $K = \{u, v\}$. Then $\{u, v\}$ is a cut edge of G (or else a shortest cycle containing $\{u, v\}$ is a triangle and K is not maximal, a contradiction) and K is a cut node of $\mathcal{G}$. Thus, G' has two connected components G_1, G_2 with clique-separator graphs $\mathcal{G}_1, \mathcal{G}_2$. By Lemma 3.23.4, $S_i \in \mathcal{G}_i$ if and only if $components_{\mathcal{G}}(S_i) > 2, i = 1, 2$. If $components_{\mathcal{G}}(S_i) = 2$, then $Remove(S_i, K)$ updates $\mathcal{G}_i, i = 1, 2$.

Since $GetMax$ and $Remove$ each require $O(n)$ time, the running time is $O(n)$. □

3.7 Computing the clique-separator graph

In this section, we give an algorithm to compute the clique-separator graph $\mathcal{G}$ of a chordal graph G in $O(n^3)$ time. The algorithm computes the clique-separator graph of G from a clique tree of G.

Begin algorithm

Verify that G is chordal by computing a clique tree T of G. Compute $S_1, S_2, \ldots, S_{n'}$, $n' \leq n$, where S_i is the minimal vertex separator corresponding to the ith edge of T, that is, $S_i = K \cap K'$ if $\{K, K'\}$ is the ith edge. Create an $n \times n$ 0–1 matrix M initialized to 0.

1. For every $1 \leq i < j \leq n'$, compare S_i and S_j. If $S_i = S_j$, delete S_j. If $S_i \subset S_j$, set $M[i, j]$ to 1. If $S_j \subset S_i$, set $M[j, i]$ to 1.

2. For every S_i, create the corresponding separator node to S_i in $\mathcal{G}$.

3. Sort the S_i's by size using counting sort.

4. For $k = 1$ to $n - 1$,

   For $l = k + 1$ to n,

   For every S_i with size k and every S_j with size l, if $M[i, j] = 1$ and $M[i', j] = 0$ for every immediate successor $S_{i'}$ of S_i in $\mathcal{G}$, then create arc (S_i, S_j) in $\mathcal{G}$.

End algorithm

Computing a clique tree of G requires $O(m + n)$ time. Since comparing any pair S_i, S_j requires $O(n)$ time, step 1 runs in $O(n^3)$ time. Since there are at most n S_i's, each with size at most n, then steps 2 and 3 run in $O(n)$ time. Any pair S_i, S_j is considered at most once in step 4 and a separator node has at most n predecessors, so step 3 runs in $O(n^3)$ time. Thus, the algorithm requires $O(n^3)$ time.

Step 1 sets $M[i, j]$ to 1 if and only if $S_i \subset S_j$. We must show that step 3 creates arc (S_i, S_j) if and only if $S_i \subset S_j$ and there is no $S_{i'}$ such that $S_i \subset S_{i'} \subset S_j$. We will show that for every separator node S, the arcs from S are correctly computed.

Let S be a separator node of size 1 and consider iteration $k = 1$ of the k loop. In the l loop, iteration $l = 2$ creates arc (S, S') for every S' of size 2 such that $S \subset S'$.

and iteration $l = l'$ creates arc (S, S') for every S' of size l' such that $S \subset S'$ and no successor S'' of S satisfies $S'' \subset S'$. Therefore, the l loop maintains the invariant: after iteration $l = l'$, arc (S, S') has been created for every S' of size at most l' such that $S \subset S'$ and no S'' satisfies $S \subset S'' \subset S'$. Thus, after iteration $k = 1$ of the k loop, every arc from every node of size 1 has been correctly computed. Similarly, after iteration $k = k'$ of the k loop, every arc from every node of size k' has been correctly computed. We conclude that the algorithm correctly computes the arcs of $\mathcal{G}$.

To compute the edges of $\mathcal{G}$, we extend the algorithm so that the maximal cliques are considered in the same way as the minimal vertex separators. Then M is a $2n \times 2n$ matrix. After the algorithm halts, we replace each arc (S, K) from a separator node S to a clique node K with an edge $\{S, K\}$.

3.8 Conclusions and future work

In this thesis, we have presented the following.

1. A fully dynamic algorithm for chordal graphs.
2. The clique-separator graph representation of a chordal graph and its properties.
3. The train tree representation of an interval graph and its properties.
4. The algorithm to compute a train tree in $O(n)$ time.
5. A fully dynamic algorithm for interval graphs with $O(n \log n)$ time per operation.
6. An algorithm to compute the clique-separator graph in $O(n^3)$ time.

We are pursuing the following goals to extend this work.

1. Find the properties of the clique-separator graph $\mathcal{G}$ when G is strongly chordal, etc.. We have already found the properties when G is proper interval or split.

2. Develop dynamic algorithms for recognizing strongly chordal graphs, proper interval graphs, split graphs, etc..

3. Improve the running time of the dynamic algorithm for interval graphs. The clique-separator graph may undergo $\Theta(n)$ arc changes when an edge is inserted or deleted. If the train tree can represent the arcs implicitly, as suggested by Lemma 3.10, then it may be possible to maintain the train tree without maintaining the clique-separator graph. We conjecture that with a vertex representation in the train tree, this approach will improve the update time to $o(n)$ time.

4. Improve the running time of the algorithm to compute the clique-separator graph $\mathcal{G}$ when G is interval. We have already developed an algorithm that, given a clique tree, computes the edges of $\mathcal{G}$ in $O(n)$ time when G is interval. We conjecture that the arcs can be computed in $O(n \log n)$ time by simulating a reverse topological order of the boxes of $\mathcal{G}$.

5. Improve the running time of the algorithm to compute the clique-separator graph $\mathcal{G}$ when G is strongly chordal, split, etc.. We have already developed an $O(n)$ time algorithm to compute $\mathcal{G}$ when G is proper interval.

The clique-separator graph provides considerable information about the structure of a chordal graph, and it may lead to dynamic algorithms for every well known subclass of chordal graphs. The train tree provides even more information about the structure of an interval graph, and it may lead to fast dynamic algorithms for a variety of problems on interval graphs. This is particularly interesting since many NP-complete problems are polynomial time solvable on interval graphs.

Bibliography

[BCD+90] C. Benzaken, Y. Crama, P. Duchet, P. L. Hammer, and F. Maffray. More characterizations of triangulated graphs. *J. Graph Theory*, 14(4):413–422, 1990.

[BG81] P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM J. Computing*, 10:751–771, 1981.

[BHT01] J. R. S. Blair, P. Heggernes, and J. A. Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 50:125–141, 2001.

[BJ96] M. Bakonyi and C. R. Johnson. Algebraic characterizations of chordality. *Linear and Multilinear Algebra*, 40:187–191, 1996.

[BL76] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Computer and System Sciences*, 13:335–379, 1976.

[Bod89] H. L. Bodlaender. Achromatic number is NP-complete for cographs and interval graphs. *Information Processing Letters*, 31:135–138, 1989.

[BP93] J. R. S. Blair and B. Peyton. An introduction to chordal graphs and clique trees. In *Graph Theory and Sparse Matrix Computation*, pages 1–29. Springer-Verlag, New York, 1993. IMA Vol. 56.

[BT96] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM J. Computing*, 25(5):956–997, 1996.

[Bun74] P. Buneman. A characterization of rigid circuit graphs. *Discrete Math.* 9:205–212, 1974.

[CH78] F. Chin and D. Houck. Algorithms for updating spanning trees. *J. Computer and System Sciences*, 16:333–344, 1978.

[CKN+95] D. G. Corneil, H. Kim, S. Nataranjan, S. Olariu, and A. P. Sprague. Simple linear time recognition of unit interval graphs. *Information Processing Letters*, 55:99–104, 1995.

[COS98] D. G. Corneil, S. Olariu, and L. Stewart. The ultimate interval graph recognition algorithm? In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 175–180, 1998.

[dFMdM95] C. M. H. de Figueiredo, J. Meidanis, and C. P. de Mello. A linear-time algorithm for proper interval graph recognition. *Information Processing Letters*, 56:179–184, 1995.

[DHH96] X. Deng, P. Hell, and J. Huang. Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. *SIAM J. Computing*, 25(2):390–403, 1996.

[Dir61] G. A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[Duc84] P. Duchet. Classical perfect graphs: An introduction with emphasis on triangulated and interval graphs. *Annals of Discrete Mathematics*, 21:67–96, 1984.

[EGI98] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, Boca Raton, FL, 1998.

[EGIN97] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification - A technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[EGIS96] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification I. Planarity testing and minimum spanning trees. *J. Computer and System Sciences*, 52:3–27, 1996.

[EIT+92] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13:33–54, 1992.

[FG65] D. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965.

[FH77] S. Földes and P. L. Hammer. Split graphs. In *Proceedings of the 8th Southeastern Conference on Combinatorics, Graph Theory and Computing*, pages 311–315, 1977.

[FK99] J. Feigenbaum and S. Kannan. Dynamic graph algorithms. In Rosen, editor, *Handbook of Discrete and Combinatorial Mathematics*, chapter 17. CRC Press, Boca Raton, FL, 1999.

[Fre85] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985.

[Fre97] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Computing*, 26(2):484–538, 1997.

[Gav72] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Computing*, 1(2):180–187, 1972.

[Gav74a] F. Gavril. An algorithm for testing chordality of graphs. *Information Processing Letters*, 3(4):110–112, 1974.

[Gav74b] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combinatorial Theory B*, 16:47–56, 1974.

[GH64] P. C. Gilmore and A. J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canadian Journal of Mathematics*, 16:539–548, 1964.

[GJSW84] R. Grone, C. R. Johnson, E. M. Sá, and H. Wolkowicz. Positive definite completions of partial Hermitian matrices. *Linear Algebra and Its Applications*, 58:109–124, 1984.

[Gol80] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[HdT98] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity and minimum spanning tree. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 79–90, 1998.

[Heg] P. Heggernes. Personal communication, May 1999.

[HK95] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 664–672, 1995.

[HK97] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proceedings of the 24th ICALP, LNCS 1256*, pages 594–604. Springer-Verlag, 1997.

[HK99] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[HL89] C. Ho and R. C. T. Lee. Counting clique trees and computing perfect elimination schemes in parallel. *Information Processing Letters*, 31:61–68, 1989.

[HM99] W. L. Hsu and T. H. Ma. Fast and simple algorithms for recognizing chordal comparability graphs and interval graphs. *SIAM J. Computing*, 28(3):1004–1020, 1999.

[HMPV00] M. Habib, R. McConnell, C. Paul, and L. Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.

[HRS94] J. Hershberger, M. Rauch, and S. Suri. Data structures for two-edge connectivity in planar graphs. *Theoretical Computer Science*, 130:139–161, 1994.

[HSS99] P. Hell, R. Shamir, and R. Sharan. A fully dynamic algorithm for recognizing and representing proper interval graphs. In *Proceedings of the 7th*

Annual European Symposium on Algorithms, LNCS 1643, pages 527–539, 1999. To appear in SIAM J. Computing.

[HT98] M. R. Henzinger and M. Thorup. Improved sampling with applications to dynamic graph algorithms. *Random Structures and Algorithms*, 11(4):369–379, 1998.

[ILR93] G. F. Italiano, J. A. La Poutré, and M. H. Rauch. Fully dynamic planarity testing in planar embedded graphs. In *European Symposium on Algorithms '93, LNCS 726*, pages 212–223. Springer-Verlag, 1993.

[Joh85] D. S. Johnson. The NP-completeness column: an ongoing guide. *J. Algorithms*, 6:434–451, 1985.

[KM89] N. Korte and R. H. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Computing*, 18(1):68–81, 1989.

[KS99] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the 31st Annual ACM Symposium on the Theory of Computing*, 1999. Submitted by invitation to Journal of Systems Science.

[LB62] C. G. Lekkerkerker and J. Ch. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51:45–64, 1962.

[LPP89] J. G. Lewis, B. W. Peyton, and A. Pothen. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM J. Computing*, 10(6):1146–1173, 1989.

[Lun90] M. Lundquist. *Zero patterns, chordal graphs, and matrix completions*. PhD thesis, Dept. of Mathematical Sciences, Clemson University, 1990.

[MM99] T. A. McKee and F. R. McMorris. *Topics in Intersection Graph Theory*. Society for Industrial and Applied Mathematics. Philadelphia. 1999. Monograph.

[Pan96] B. S. Panda. New linear time algorithms for generating perfect elimination orderings of chordal graphs. *Information Processing Letters*. 58:111–115. 1996.

[Rau94] M. Rauch. Improved data structures for fully dynamic biconnectivity. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, pages 686–695. 1994.

[RTL76] D. J. Rose. R. E. Tarjan. and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Computing*. 5(2):266–283. 1976.

[Shi84] D. R. Shier. Some aspects of perfect elimination orderings in chordal graphs. *Discrete Applied Mathematics*. 7:325–331. 1984.

[SP75] P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM J. Computing*. 4(3):375–380. 1975.

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Computer and System Sciences*. 26:362–391. 1983.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*. 22(2):215–225. 1975.

[TY84] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs. test acyclicity of hypergraphs. and selectively reduce acyclic hypergraphs. *SIAM J. Computing*. 13(3):566–579. 1984.

[Wal72] J. R. Walter. *Representations of rigid circuit graphs*. PhD thesis. Wayne State University. 1972.

[Wal78] J. R. Walter. Representations of chordal graphs as subtrees of a tree. *J. Graph Theory*, 2:265–267, 1978.