Fast Parallel Algorithms for the Clique Separator Decomposition

Elias Dahlhaus [*] Marek Karpinski [†] Mark B. Novick [‡]

Abstract

We give an efficient *NC* algorithm for finding a clique separator decomposition of an *arbitrary* graph, that is, a series of cliques whose removal disconnects the graph. This algorithm allows one to extend a large body of results which were originally formulated for chordal graphs to other classes of graphs. Our algorithm is optimal to within a polylogarithmic factor of Tarjan's $O(mn)$ time sequential algorithm. The decomposition can also be used to find *NC* algorithms for some optimization problems on special families of graphs, assuming these problems can be solved in *NC* for the prime graphs of the decomposition. These optimization problems include: finding a maximum-weight clique, a minimum coloring, a maximum-weight independent set, and a minimum fill-in elimination order. We also give the first parallel algorithms for solving these problems by using the clique separator decomposition. Our maximum-weight independent set algorithm applied to chordal graphs yields the most efficient known parallel algorithm for finding a maximum-weight independent set of a chordal graph.

Key words. parallel algorithms, clique separator decomposition, chordal graphs, perfect graphs, maximum independent set, chromatic number

AMS(MOS) subject classifications. 05C05, 05C15, 05C75, 68Q10, 68R10

[*]Department of Computer Science, University of Bonn

[†]Department of Computer Science, University of Bonn, and International Computer Science Institute, Berkeley, California. Supported in part by Leibniz Center for Research in Computer Science, by the DFG Grant KA 673/2-1, and by the SERC Grant GR-E 68297

[‡]Department of Computer Science, Cornell University, Ithaca, NY 14853-7501. Supported by NSF grant CCS-8806979

1 Introduction

Decompositions are often used in studying graph problems. Many times a graph has a property if and only if the pieces it is decomposed into also have that property. Several graph classes have been characterized by the structure of their clique separators, among them chordal graphs [Di 61, HS 58], path graphs (the intersection graphs of paths in a tree) [MW 86], and Gallai graphs (graphs where every odd cycle of length five or more contains two non-crossing chords) [Ga 77]. One of the original motivations for studying the clique separator decomposition is related to the problem of recognizing perfect graphs [Wh 84]. If X is a clique separator of G and removing X leaves connected components with vertex sets $V_1, V_2, \ldots, V_k$, then G is perfect if and only if $G(V_1 \cup X), G(V_2 \cup X), \ldots, G(V_k \cup X)$ are all perfect. The clique separator decomposition is a perfection-preserving decomposition.

Decompositions are also useful in designing graph algorithms. The two techniques commonly used with decompositions are divide-and-conquer and dynamic programming. We repeatedly decompose a graph into smaller pieces until we obtain graphs that can no longer be decomposed. These graphs are called *prime graphs* or *atoms*. We solve the problems on these atoms and combine the solutions to find the solutions on larger and larger components until we have the solution for the original graph. Recently, this paradigm has been used to design several efficient parallel and sequential algorithms for graph classes with small separators [BLW 85, HY 88].

Suppose X is a clique separator of G. Then there is a vertex partition A, B, X such that no vertex in A is adjacent to one in B. We can then decompose into two components $G' = G(A \cup X)$ and $G'' = G(B \cup X)$, and then recursively perform the decomposition on G' and G'' in turn until only prime graphs result. We can represent the decomposition of G into G' and G'' as a binary tree where the leaves are prime graphs and the internal nodes are clique separators. Following Tarjan's [Ta 85] terminology, we call such a tree a *binary decomposition tree*. Throughout this paper we will assume that each clique separator is a minimal clique separator, because this results in smaller separators and fewer atoms in the decomposition.

Whitesides [Wh 84] designed the first polynomial time sequential algorithm for finding a clique separator decomposition. Her algorithm ran in $O(n^3m)$ time. Tarjan found a faster algorithm which ran in $O(nm)$ time. Burlet and Fonlupt gave an $O(n^4)$ algorithm which checked whether a graph had a universal clique separator S, one whose removal left each component of the graph with a vertex that was adjacent to all the vertices of S. They used this algorithm to help recognize Meyniel graphs, a class of perfect graphs where every odd cycle contains at least two chords. These algorithms are the main sequential algorithms for performing the decomposition.

In the third section of the paper we show how to efficiently parallelize the sequential algorithms for finding a clique separator decomposition. The fastest published sequential algorithm is due to Tarjan [Ta 85], but the algorithm he gave is inherently sequential. This algorithm runs in $O(nm)$ time by first finding a special ordering of the vertices called a *minimal elimination order*. Dahlhaus and Karpinski [DK 89] have given a parallel algorithm finding the minimal elimination order of an arbitrary graph, but we use a different elimination order that can be obtained more efficiently. We show how a tree representation of the graph can be constructed in parallel, where the edges of the tree correspond to clique separators. Our parallel algorithm runs in $O(\log^3 n)$ time with $O(nm)$ processors on a CREW PRAM. The processor-time product of our algorithm is nearly optimal. Our algorithm is related to, but more processor-efficient than an earlier version by Dahlhaus and Karpinski [DK 88a] that runs in $O(\log^2 n)$ time using $O(n^4)$ processors.

In Section 4 we show how the decomposition can be used to solve several optimization problems for special graph classes. In particular we give *NC* algorithms for finding a maximum clique, graph coloring, maximum independent set, and a minimum fill-in order of a graph, assuming these problems can be solved in *NC* for the prime graphs in the special graph class. The number of processors used by our parallel algorithms is at most $O(n)$ times the best known sequential running times for these problems. All four of these problems are normally *NP*-complete, but can be solved in polynomial time for chordal graphs, because the only prime chordal graphs are cliques. Our algorithms first construct a tree representation of the graph and then apply tree processing techniques such as terminal branch removal [NNS 87] and parallel tree contraction [MR 85]. We can also find the chromatic polynomial of a graph assuming it is easy to do this on the prime graphs.

2 Definitions

We let $G = (V, E)$ denote a graph with vertex set V and edge set E. Let n and m denote the number of vertices and edges respectively. In this paper we will assume that G is connected. If X is a subset of V, then $G(X)$ is the subgraph induced by X. When $G(V - X)$ is disconnected we say that X is a *separator* of G. A *clique* of G is a subgraph of G in which every pair of vertices is connected by an edge. We call X a *clique separator* of G if the vertices of X form a clique and X is also a separator of G. Furthermore, X is a *minimal clique separator* of G if it is a clique separator of G and no proper subset of X is a clique separator. A *perfect graph* is one where the maximum clique size equals the chromatic number for every induced subgraph. A *chord* of a cycle is an edge connecting two non-consecutive vertices of the cycle. *Chordal graphs* are graphs where every cycle of length greater than three contains a chord.

3 Parallel Decomposition Algorithms

The fastest methods depend on properties of chordal graphs for their efficiency so we will review the needed results. First, we define some of the terms used to talk about chordal graphs. An *elimination order* π is a numbering of the vertices of G from 1 to n. We define the *fill-in* F_π induced by the order π to be the following set of edges:
$F_\pi = \{[v,w] | v \neq w, [v,w] \notin E$, and there is a path
$v = v_1, v_2, \ldots, v_k = w$ in G such that $\pi(v_i) < min\{\pi(v), \pi(w)\}$
for $i = 2, \ldots, k-1\}$.
A *perfect elimination order*, often abbreviated to PEO, is an elimination order with no induced fill-in. In a PEO, the higher-numbered neighbors or a vertex form a clique. An order π is *minimum* if no other elimination order has a smaller cardinality fill-in, and π is *minimal* if there is not an elimination order σ such that F_σ is properly included in F_π. The *fill-in graphs* for π is the graph $G_\pi = (V, E \cup F_\pi)$. Tarjan's algorithm depends on the following theorems he proved:

Theorem 3.1 *Any order π is a PEO of G_π.*

Theorem 3.2 *G has a PEO if and only if G is chordal.*

Theorem 3.3 *Let π be a minimal order. For any decomposition by clique separators, every edge $[v,w] \in F_\pi$ is such that a unique atom contains both v and w.*

Our algorithm depends on the following theorem which we prove.

Theorem 3.4 *Let π be a minimal order of G. If C is a clique separator of G_π containing no fill-in edges, then C is also a clique separator of G. Conversely, if C is a clique separator of G, then it is also a clique separator of G_π.*

Proof: The first half of this theorem follows immediately since if C is a clique separator for a graph, then it is a clique separator for any subgraph that contains C and has the same number of vertices as the original graph. The second half follows from the correctness of Tarjan's algorithm. His algorithm checks the cliques induced by a vertex of G_π and its higher-numbered neighbors to determine if they form a clique separator of G. Clearly any clique in G is also a clique of G_π. Any clique separator this algorithm finds in G will also be a clique separator of G_π since π is a PEO of G_π. ∎

The tree representation we use to parallelize the algorithm is a generalization of the notion of a clique tree of a chordal graph. A *clique tree* of a chordal graph $G = (V, E)$ is a tree T whose nodes are the maximal cliques of G and whose arcs are defined in such a way that the set of maximal cliques containing $v \in V$ form a connected subtree of T. Buneman [Bu 74] and Gavril [Ga 74] have shown that every chordal graph has a clique tree representation. For an arbitrary graph $G = (V, E)$ we define a *simplicial tree* to be a tree T whose nodes are the atoms of G and whose arcs are defined so that the atoms containing a vertex $v \in V$ form a subtree of T. Efficient parallel algorithms exist for finding a clique tree in parallel. We will show that the clique separator decomposition of G can be computed in parallel by giving an efficient algorithm for finding a simplicial tree of G. Given the simplicial tree of G, it is easy to find the clique separator decomposition because the intersection of two adjacent atoms in the simplicial tree forms a clique separator of G.

Our algorithm is composed of the following steps:

1. Compute a clique separator-preserving chordal extension $G' = (V, E \cup F)$ of G. (Remark: F and E need not to be disjoint)
2. Find a PEO of G'. Use this to find a clique tree of G', and then convert it to a simplicial tree of G. The intersection of neighboring atoms in the simplicial tree will be a clique separator of G.

The succeeding subsections give a more detailed analysis of this algorithm.

3.5.1. An Outline of the Chordal-extension Algorithm.

Here we present a Clique Separator Decomposition algorithm which combines methods introduced recently in [DK 88b] with certain extensions of P. Klein's FOCS '88 [Kl 88] method for chordal graphs. The algorithm first computes a chordal extension G' which preserves clique separators, together with a perfect elimination order on G'. The last step is to check for each clique separator of $G' = (V, E') := (V, E \cup F)$, whether it is a clique separator of G (that means a complete subgraph).

We begin with the algorithm which computes an ascending sequence $(C_1, \cdots, C_n)$, $C_i \subseteq V$, of "convex" sets of G' (endsegments of a PEO of G') (see [FJ 86]), such that

i) $C_{i+1} \backslash C_i$ has only one element;

ii) $[x, y] \in E'$ iff $[x, y] \in E$ or $x, y \in C_i$ and there is a connected component $\tilde{C}$ of $G(V \backslash C_i)$, such that $x, y \in N(\tilde{C})$;

iii) each clique separator of G is also a clique separator of G'.

Clearly this sequence $(C_i)_{i=1}^n$ defines a perfect elimination order π for $G' = (V, E \cup F)$. Therefore G' is chordal. All these C_i are convex for G' in the sense of [FJ 86] (closed by chordless paths).

By a procedure $NONE$, we compute "convex" sets (see [FJ 86]) C_1, C_2, such that $\#C_1, \#(C_2 \backslash C_1), \#(V \backslash C_2) \leq \frac{2}{3}\#V$. Moreover, $(V, F \cup E)$ shall preserve clique separators. The procedure $REFINE$ computes for each "convex" set C suitable "convex" sets C_1, C_2, such that $C \subseteq C_1 \subseteq C_2$ and $\#(C_1 \backslash C), \#(C_1 \backslash C_2), \#(V \backslash C_2) \leq \frac{2}{3}\#(V \backslash C)$. Let $[x, y] \in F_{\hat{C}}$

iff x and y are adjacent to the same connected component of $V\backslash\hat{C}$, where $\hat{C} = C_1$ or $\hat{C} = C_2$, and let $E_{\hat{C}} := E \cup F_{\hat{C}}$.

Procedures $NONE$ and $REFINE$ are based on P. Klein's ([Kl 88]) new technique for chordal graphs.

Observation.

Whenever u and $v \in C_i\backslash C$ are in the same connected component with respect to E_{C_i}, they are in the same connected component of $V\backslash C$ with respect to E, and vice versa.

3.5.2. The Fine Structure of the Chordal-Extension Algorithm.

Procedure NONE (G (V,E)).

Step 1 Let $D := \{v \in V : \text{ degree } (v) \geq \frac{2}{3}\#V\}$.

Step 2 If D is not complete, pick some $x, y \in D, [x,y] \notin E$ and let $C' := \{v|[v,x],[v,y] \in E\}$ be the common neighborhood of x and y. Make C' a clique. (Comment: each $u, v \in C$ are on the cycle (x,u,y,v,x). Therefore each new edge of C is on a chordless cycle of G. That means that this extension $E \cup F$ preserves clique separators.)

Step 3 Let $C_1 := C_2$ be some subset of C', such that $\frac{1}{3}\#V \leq \#C_i \leq \frac{2}{3}\#V$. (Clearly $\#C' < \frac{1}{3}\#V$.)

Step 4.1 If D is complete and $\#D \geq \frac{1}{3}\#V$, then C is some subset of D, s.t.

$$\frac{1}{3}\#V \leq \#C \leq \frac{2}{3}\#V.$$

Step 4.2 If $\#D \leq \frac{1}{3}\#V$, D is complete, and all connected components of $V\backslash D$ have a size $\leq \frac{2}{3}$:
Let $(\tilde{C}_1, \cdots, \tilde{C}_k)$ be an enumeration of these connected components; let for $j = 0, \cdots, k$: $C'_j := D \cup \bigcup_{i \leq j} \tilde{C}_i, \hat{F}_{C_i} := \emptyset$. $(\#(C'_{j+1}\backslash C'_j) \leq \frac{2}{3}\#V$; note that $C'_k = V)$.
Let C_1 be some C'_j, such that $\#C'_j \leq \frac{2}{3}\#V, \#(V\backslash C'_j) \leq \frac{2}{3}\#V$. Set $C_2 := C'_{j+1}$.

Step 4.3 If D is complete or empty and there is a connected component $\hat{C}$ of $V\backslash D$, such that $\#(V\backslash D) \geq \frac{2}{3}\#V$:
Compute a spanning tree on $\hat{C}$ and, using this spanning tree, compute an enumeration $(x_i)_{i=1}^m$ of $\hat{C}$, such that each initial segment $\hat{C}_j := \{x_i | i \leq j\}$ is connected; let C'_j be the neighborhood of $\hat{C}_j$ and $\hat{C}_{m+1} := V$.

Step 5.1 Pick up a $\#(C'_j \backslash C'_{j-1}) \geq \frac{1}{3}\#V$ (if it exists, here $C'_0 := \emptyset$):
Set $C_1 := C'_{j-1}$ and $C_2 := C'_j$.

Step 5.2 Otherwise, if such C'_j, C'_{j-1} do not exist, there is a C'_j, s.t.

$$\frac{1}{3}\#V \leq \#C'_j \leq \frac{2}{3}\#V;$$

pick up such a C'_j, set $C_1 := C_2 := C'_j$.

Step 6 **Set** $NONE := (C_1, C_2)$.

End of Procedure $NONE$.

Now we proceed with the procedure $REFINE$.

Procedure $REFINE(G, C)$.

If for each connected component $K_1, \cdots, K_n$ of $V \backslash C$ $\#K_i \leq \frac{2}{3}\#(V \backslash C)$.:
Let $C_i := C \cup \bigcup_{j<i} K_i$.
Otherwise: Let K_1 be the largest component; apply $REFINE(C \cup K_1, C)$.

Assume $V \backslash C$ is connected:

Procedure $REFINE'(G, C)$.

Step 1 Let $D := \{x \in G : d_{V \backslash C}(x) := \#\{y \in V \backslash C | [y, x] \in E\} \leq \frac{2}{3}\#(V \backslash C)\}$.
Let D' be the union of all connected components of D touching C.

Step 2.1 (Low degree extension):
Compute a spanning forest on D' and, using this, an enumeration $(x_i)_{i=1}^m$ of D', such that for each initial segment $u_j := \{x_i\}_{i=1}^j$, $u_j \cup C$ is connected. Let $\hat{C}_j := N(u_j) \cup C$. Here $N(u_j)$ is the *neighborhood* of u_j.

Step 2.1.1 If $\#(\hat{C}_m \backslash C) \geq \frac{1}{3}\#(V \backslash C)$, then
let $\hat{C}_{m+1} := V$ and pick up a $\#(\hat{C}_{j+1} \backslash \hat{C}_j) \geq \frac{1}{3}\#(V \backslash C)$ (if it exists). Let $C_1 := \hat{C}_j$ and $C_2 := \hat{C}_{j+1}$.

Step 2.1.2 If such $\hat{C}_j, \hat{C}_{j+1}$ do not exist,
let C_1 be any $\hat{C}_j$ s.t.

$$\frac{1}{3}\#(V \backslash C) \leq \#(\hat{C}_j \backslash C) \leq \frac{2}{3}\#\hat{C}_j \text{ and } C_2 := \hat{C}_{j+1}.$$

If $\#\hat{C}_m < \frac{1}{3}\#(V \backslash C)$: Let $C_1 := \hat{C}_m$ (possibly $C_1 = C$).
If $\exists x \in C_1$, s.t. $d_{V \backslash C_1}(x) \leq \frac{2}{3}\#(V \backslash C)$: Let $C_2 := C_1 \cup N(x)$.

Step 2.2 Otherwise (High degree extension):

Step 2.2.1 Let $(x_i)_{i=1}^k$ be an enumeration of all $x \in C_1$, such that $d_{V\backslash C_1}(x) \neq 0$ and let $F_j := \{x \,|\, [x, x_i] \in E$ for all $i = 1, \cdots, j\}$. If for some j, $\#F_j \leq \frac{2}{3}\#(V\backslash C)$, set $C_2 := C_1 \cup F_j$.
Otherwise apply $NONE$ to $G(F_k)$ with outputs C_1' and C_2'.

Step 2.2.2 If $\#(C_2'\backslash C_1') \geq \frac{1}{3}\#(V\backslash C)$, let $C_1 := C_1 \cup C_1'$ and $C_2 := C_1 \cup C_2'$.
If $\#(F_k\backslash C_2') \geq \frac{1}{3}\#(V\backslash C)$, let $C_2 := C_1 \cup C_2'$.
Otherwise $C_2 := C_1 \cup C_1'$.

End of procedure $REFINE$.

The whole algorithm works as follows:

Algorithm Parallel Fill-In.

Input $G = (V, E)$.

Let $(C_1, C_2) := NONE(G)$. Apply $REC(C_1, C_2)$.

Procedure $REC(C_1, \cdots, C_k)$.

Step 1 If $\#C_1 = 1$ and $\#(C_{i+1}\backslash C_1) = 1$ for each i, then STOP.

Step 2 Compute new "convex" sets:
Let C_1^0, C_2^0 be the two C_1, C_2 arising from the application of $NONE(G(C_1))$; let C_1^i, C_2^i be the two C_1, C_2 arising from the application of $REFINE(G(C_{i+1}), C_i)$

Step 3 Add new edges:

Step 3.1 For C_i and C_{i+1} and each connected component K of $C_{i+1}\backslash C_i$, let x_K be a vertex in C_i adjacent to K, such that $x_K \notin C = C_k^j\backslash C_j$ such that C maximal (x_K is in a minimal number of old and new known "convex" sets).

Step 3.2 Join each vertex x of C_i adjacent to K with x_K by an edge $[x, x_K] \in E_{\text{new}}$ (hereby we have guaranteed that each $x, y \in C_i\backslash C_{i-1}^2, C_i\backslash C_{i-1}^1, C_i\backslash C$ etc. respectively those which are in the same connected component of $V\backslash C_{i-1}^2, \ldots$, resp. are also in the same connected component of $C_i\backslash C_{i-1}^2, C_i\backslash C_{i-1}^1, C_i\backslash C_{i-1}, \ldots$ etc. resp.):

Step 3.3 Let K be a connected component of $C_{i+1}\backslash C_i^2$ and x_K be again an $x \in C_i^2$ contained in a minimal member of known "convex" sets C_i, C_i^j, and for each x adjacent to K an edge $[x, x_K] \in E_{\text{new}}$.

Step 3.4 Do the same procedure also with the level $C_i^2 \backslash C_i^1$. (Hereby it is guaranteed that connectedness in $\tilde{C}_2 \backslash \tilde{C}_1$ and $V \backslash \tilde{C}_1$ for each $\tilde{C}_1 \subseteq \tilde{C}_2$, s.t. $\tilde{C}_1, \tilde{C}_2 \in \{C_i, C_i^j \mid i = 1 \dots k, j = 1,2\}$, are equivalent statements (compare the observation)); $E := E \cup E_{\text{new}}$;
apply $REC(C_0^1, C_0^2, C_1, C_1^1, C_1^2 \dots)$.

End of Procedure REC.

Output (V, E).

End of the Algorithm Parallel Fill-In.

3.5.3. Analysis of the Algorithm:

1) The recursion depth of REC is $O(\log n)$, since the maximum cardinality of levels $C_{i+1} \backslash C_i$ goes down by at least $\frac{1}{3}\#(C_{i+1} \backslash C_i)$ at each step.

2) as mentioned above, the following is valid for the output (V, E) in $C_i \subset C_j$, $x, y \in C_j \backslash C_i$: x, y in the same connected component of $C_j \backslash C_i \iff x, y$ in the same connected component of $V \backslash C_i$.

3) Let $x < y$ if for some $k : y \in C_k$ but $x \notin C_k$. For the corresponding chordal extension F of (V, E_{old}), $[x, y] \in F \iff [x, y] \in E$ or x, y are adjacent to the same connected component K of $\{v \mid v < x\}$. But for each such connected component K, we have $x_K = x$. But $[y, x] = [y, x_K] \in E_{\text{new}}$.
Therefore the output (V, E) is a chordal extension of $G = (V, E_{old})$. Moreover: Output $(V, E) = (V, F)$.

4) *Old clique separators are preserved:* We have to prove this statement for each step of the application of $NONE$ or $REFINE$.

$NONE$: Additional edges of common neighbors of $x, y \in D$ are in a cycle of length four. Therefore clique separators are preserved. Let M be a connected subset of V and $C := M'$ be the set of neighbors of M. Let $x, y \in M'$ be adjacent to the same connected component of $V \backslash M'$. Then $x, y \in M' \backslash M$. Consider a path $x, y_1, \dots, y_p, y$, s.t.$y_1 \dots y_p$ is chordless and p is minimal. But then there is also no chord $[x, y_i]$ or $[y_j, y]$. Let $x, z_1, \dots z, y$ be a chordless path, s.t. $z_1, \cdots z_p \in M$. But then the concatenation of these two paths form a cycle. Hereby the application of the procedure $NONE$ preserves clique separators.

$REFINE$ (Low degree extension): Assume any "convex" set $C := C_i$ is given. Assume that vertices adjacent to the same connected component of $V \backslash C$ form a clique in an extension F of E preserving clique separators. Assume M is connected and intersects

C. Let M' be defined as above and $C' := M' \cup C$. Let $x \in M' \backslash M$ and $y \in M' \backslash M$ be connected by a shortest path p in $V \backslash C'$. Then by the same arguments as before $[x, y]$ is a chord of a chordless cycle, if $[x, y] \notin E$.

Assume now $y \notin M' \backslash M$; that means $y \in C$. But y and $M \cap C$ are adjacent to the same connected component of $V \backslash C$. But then we find a y' and a path $P_1 \subseteq M$, such that (y, y', P_1, x, P) forms a chordless cycle in a clique separator preserving extension F of E.

$REFINE$ (High degree extension): Assume now $V \backslash C$ is connected, $D \subset C$ and $C' = C \cup \{y \mid \forall x \in D, [y, x] \in E\}$. Then we may assume that D forms a clique in some clique separator preserving extension F of E. We have to prove that for $x, y \in C' \backslash C$ which are adjacent to the same connected component of $V \backslash C'$, we can join them by an edge and no clique separator is destroyed. Let $x \in C' \backslash C$, $y \in C' \backslash C$ and $p = (x, x_1, x_2, x_3, \ldots)$ be a shortest (chordless) path $\subseteq V \backslash C'$ connecting x and y. Let $d \in D$ and $[d, x_i], [d, x_j] \in E$ but not $[d, x_k] \in E$ for $i < k < j$. Then $(d, x_i, x_{i+1}, \ldots, x_j, d)$ forms a chordless cycle. Therefore $[x_i, x_j]$ can be added, such that no clique separator is destroyed. Since each x_i is not adjacent to at least one $d \in D$, it is possible to add a chord abbreviating p, such that x_i is not used and no clique separator is destroyed. Therefore an edge $[x, y]$ can be added, such that no clique separator of G is destroyed.

Now let $x \in C' \backslash C$, but $y \in C$. Then the same argument to add an edge $[x, y]$ works. By these observations no edge of $F \backslash E$ destroys some clique separator.

3.1 Construction of the Simplicial Tree

We find the simplicial tree representation of G by first obtaining a clique tree of G', and then converting it to a clique tree. We can find G' by using the PEO we found for G' to calculate the fill it induces on G. Hafsteinsson's [Ha 88] algorithm does this in $O(\log n)$ time with $O(n^2)$ processors. Next we get a clique tree representation of G' in $O(\log n)$ time using $O(|E \cup F|) = O(n^2)$ processors. Folklore says we can convert Klein's elimination tree into a clique tree within these bounds.

Then check if the intersection of two neighboring cliques in the clique tree contains a fill-in edge takes $O(\log n)$ time on a CREW-PRAM using $O(m + n)$ processors. We find the vertices in the intersection of two cliques, say there are k of them. If $\binom{k}{2} > m$, then the intersection contains a fill-in edge. Otherwise, check each edge in this clique of size k to determine if it is an edge in E. Any edge not in E is a fill-in edge. Performing this computation for all cliques requires $O(nm)$ processors since there are $O(n)$ maximal cliques in G. Merge the vertex sets of the two neighboring cliques if their intersection contains fill-in. If there is no fill-in here, then their intersection is a clique separator of G. The merging of cliques can also be done quickly. We are left after the merging with a simplicial tree instead of a clique tree.

Theorem 3.5 *Clique separator decomposition of an arbitrary graph can be computed in*

$O(\log^3 n)$ time using $O(mn)$ processors on a CREW-PRAM.

Proof: The analysis of the chordal extension step shows this step requires $O(n^2)$ processors and $O(\log^3 n)$ time. The simplicial tree step uses $O(mn)$ processors and $O(\log n)$ time. Therefore, the entire algorithm can be implemented as claimed. ∎

4 Algorithms for Optimization Problems

Here we show that the clique separator decomposition can be used to find *NC* algorithms for several graph problems. These problems are *NP*-complete for general graphs, but if we can solve these problems efficiently for atoms, then we can solve them quickly for the entire graph. In fact, our first three algorithms only use a linear number of processors.

4.1 Minimum Fill-in Orders

The problem of determining whether there is an elimination order of G that results in k or fewer fill-in edges, is *NP*-complete if k is a problem parameter [Ya 81]. However, if we can find a minimum fill-in order for each atom G_i of G, then we can also easily find a minimum fill-in order of G. Let F_i be the fill induced by the order on atom G_i. Tarjan proved that $G' = (V, E \cup \bigcup_{i=1}^{k} F_i)$ is chordal. A PEO for G' would also give minimum fill-in for G by yielding minimum fill-in for each G_i. We parallelize the algorithm by computing the PEOs in parallel. Using Klein's algorithm we can do this in $O(\log^2 n)$ time with $O(m)$ processors.

4.2 Maximum Cliques

The maximum-weight clique of G is the maximum of the maximum-weight cliques for each atom of G. If, in parallel, we can find the maximum-weight clique of the atoms of G, then we can also get the largest-weight clique in G in $O(\log n)$ time with $O(m)$ processors.

4.3 Graph Coloring

Suppose there is an efficient parallel algorithm to color the atoms of G. We can use this algorithm to give a parallel algorithm for coloring G.

1. Find a minimum coloring for each atom of G.

2. Form a new graph H with vertex set V. For convenience, we will assume each vertex has a unique number. If there is an atom of G in which vertices v and w receive the

same color, and if w is the lowest numbered vertex of its color, then put edge $[v, w]$ in H.

3. After H has been formed, find its connected components. The vertices in a component of H will be receive the same color in G.

4. Shrink G by contracting the components of H to single vertices in G. Atoms of G are transformed into cliques of the resulting quotient graph, but both graphs have essentially the same clique separators. Therefore, the quotient graph is chordal since each of its atoms is a clique. In *NC*, we can optimally color the quotient graph with Klein's coloring algorithm.

5. Finally, color G by assigning each vertex in G the color of the corresponding vertex in the quotient graph.

Theorem 4.1 *Given an efficient algorithm for coloring the atoms of a graph, we can color the entire graph in time $O(\log^2 n)$ using $O(m)$ processors.*

Proof: This algorithm can be implemented in *NC* with the number of processors proportional to the sums of the sizes of the atoms. We can find the connected components for step 3 in $O(\log n)$ time by using $O(n)$ processors on an EREW-PRAM through the use of Euler tour techniques [TV 80]. The total running time is bounded by the time required to color a chordal graph, which is currently $O(\log^2 n)$ for Klein's algorithm, but is $O(\log n)$ for Ho and Lee's algorithm [HL 88]. ∎

4.4 Chromatic Polynomial

For any graph G, the chromatic polynomial $f(G, x)$ is defined to be the number of ways we can color G where we have a choice of x different colors. The chromatic polynomial of K_n is $x(x-1)\ldots(x-n+1)$. Suppose C is a clique whose removal leaves components with vertex sets A and B. Then $f(G, x) = f(G(A \cup C), x) f(G(B \cup C), x) / f(C, x)$. We can color $G(A \cup C)$ and $G(B \cup C)$ independently of each other as long as C receives the same color in both cases. Given the simplicial tree of G and the chromatic polynomial of each atom, we can compute the chromatic polynomial of G by repeatedly using the above formula at each clique separator of G. Thus the chromatic polynomial of G is the product of the chromatic polynomials of the atoms of G divided by the product of the chromatic polynomials of the simplicial tree clique separators of G. This algorithm parallelizes easily because we can find the clique separators easily in parallel.

4.5 Maximum Independent Sets

Given a graph G with integer weights on the vertices, the problem of deciding whether G has an independent set of weight $\leq w$ is *NP*-complete when w is a problem parameter.

Several authors [Ta 85, Wh 84] have noted that the clique separator decomposition can help solve this problem in the following way. Suppose C is a clique of G whose removal leaves connected components induced by vertex sets A and B. Denote by $wt(I)$ the total weight of vertex set I, and by $N(v)$ the set of vertices adjacent to v.

1. For each $v \in C$, find a maximum-weight independent set $I(v)$ in $G(B - N(v))$. Also find a maximum-weight independent set I' in $G(B)$.

2. For each $v \in C$, redefine the weight of v to be $wt(v) + wt(I(v)) - wt(I')$. Find a maximum weight independent set I'' in $G(A \cup C)$ with respect to the new weights.

3. Define $I = I'' \cup I'$ if $I'' \cap C = \emptyset$. If $v \in I''$, then define $I = I'' \cup I(v)$. In either case, $wt(I) = wt(I'') + wt(I')$ and I is the maximum weight independent set of G.

Naor *et al.* suggest finding the maximum weight independent set of a chordal graph by terminal branches (a leaf node together with a consecutive chain of ancestors with only one child) of its clique tree, and finding the maximum weight independent set of the remaining graph. We give a more efficient but similar algorithm, that also works for the more general case of the clique separator decomposition. Let $B_1, B_2, \ldots, B_k$ be branches in the simplicial tree of G consisting of leaf nodes plus their parents. We apply Miller and Reif's [MR 85] technique of parallel tree contraction. In a rake step, we

1. Find all vertices of G that only appear in leaf cliques of the simplicial tree.

2. For each vertex v that appears in a leaf atom and some other atom. For each such atom compute the weight of the maximum independent set that does not contain v or any vertices in $N(v)$. Also compute the weight of the maximum independent set that does not contain any vertices outside of the atom. Subtract the second of these quantities from the first, and then decrease the weight of v by this amount. If $wt(v)$ is now non-positive, remove v from the graph. We decrease v's weight once for each leaf atom it is a part of.

3. Remove all the vertices that only appear in leaf atom.

The weight of the maximum independent set can be found by applying the idea of the last paragraph at each terminal branch. The weight of the maximum independent set of G is the weight of the maximum independent set of the graph after the rake step plus the weight of the largest weight vertex in each clique that was raked away.

The other half of Miller and Reif's method is the compress step. Here we compress a parent-child pair of tree nodes into one node if both the parent and child nodes had only one child. Let A denote the parent atom, and B denote its child in the simplicial tree. Let S_1 denote the clique which separates atom A from its parent, S_2 denote the clique which separates A from its child B, and S_3 denote the clique which separates B from its child. We introduce two matrices M_A and M_B that denote the weights of maximum independent sets in A and B. In particular, for every vertex $u \in S_1$ and vertex $v \in S_2$, we let $M_A[u, v]$ denote the maximum weight of an independent set in A that does not include u or any vertex in $N(u)$, but it does contain v. We extend this definition in three ways:

- by assigning to $M_A[-, v]$ the weight of the largest independent set of A that contains v, but none of S_1.
- by assigning to $M_A[u, -]$ the weight of the largest independent set of A that does not contain u, one of u's neighbors, nor any vertex in S_2.
- by assigning to $M_A[-, -]$ the weight of the largest independent set of A that does not contain any vertex in S_1 or S_2.

We define M_B in a like manner with respect to vertices in S_2 and S_3. The goal of the compress step is to compute a product matrix M_{AB} defined on vertices in S_1 and S_3.

Theorem 4.2 *The maximum weight independent set algorithm runs in $O(\log^2 n)$ time using $O(n^4)$ processors if we can find the maximum weight independent set of each of the atoms in $O(\log^2 n)$ time using $O(n^2)$ processors.*

Proof: With an efficient algorithm for finding the atoms of G we can compute all the initial M_A and M_B matrices. We need $O(m)$ processors and $O(\log n)$ time to do all the rakes at a given moment since we perform at most $O(m)$ weight decrements, and we find the new weight of any vertex in $O(\log n)$ time. We can do a compress in $O(\log n)$ time by using $O(n^3)$ processors via the usual matrix multiplication algorithm in the $(\max, +)$ ring. At most $O(n)$ compresses are performed at once, implying that $O(n^4)$ processors are sufficient for compress. Miller and Reif showed that $O(\log n)$ rakes and compresses are enough to reduce a tree to a single node. Therefore, we can find the weight of the maximum independent set in $O(\log^2 n)$ time using $O(n^4)$ processors. ∎

Even fewer processors are needed if G is chordal. The algorithm becomes:

1. For each terminal branch B_i and every vertex v that appears in both B_i and the rest of the simplicial tree, find the largest independent set of B_i that does not contain v or its neighbors.
2. For each terminal branch find the largest independent set that contains no vertices appearing outside of the terminal branch.
3. Compute the differences between the values in the last two steps, and decrease the weight of vertex v accordingly.

Theorem 4.3 *The maximum weight independent set algorithm for chordal graphs can be implemented to run in $O(\log^3 n)$ time using $O(n^3)$ processors.*

Proof: Each terminal branch induces an interval graph, the intersection graph of intervals along the real line. We use Helmbold and Mayr's [HM 86] algorithm for finding the maximum weight independent set in an interval graph to perform steps one and two. Their algorithm has the advantage that it not only finds the largest independent set in the interval graph, but it also finds the largest set for each subportion of the interval graph that contains one of its ends. Let $|B|$ denote the number of vertices in terminal branch B that do not appear elsewhere in G, and let $|S|$ be the number of vertices in the clique that separates B from the rest of the graph. Using Helmbold and Mayr's algorithm requires time $O(\log^2 |B|)$ and $O(|B|^3 + |S|)$ processors for terminal branch B. Therefore, it takes $O(\log^2 n)$ time and $O(n^3)$ processors to process all the terminal branches at once. This running time is multiplied by a factor of $O(\log n)$ since there are a logarithmic number of iterations. ∎

References

[BLW 85] Bern, M.W., Lawler, E.L. and Wong, A.L. *Why certain Subgraph Computations require only linear time*, Proc.26^{th} FOCS (1985), pp. 117-125.

[Bu 74] Buneman, P. *A Characterization on Rigid Circuit Graphs*, Discrete Mathematics 9 (1974), pp. 205-212.

[DK 88a] Dahlhaus, E., and Karpinski, M. *Efficient Parallel Algorithm for Clique Separator Decomposition*, Research Report No. 8531-CS, University of Bonn 1988.

[DK 88b] Dahlhaus, E., and Karpinski, M. *Fast Parallel Decomposition by Clique Separators*, Research Report No. 8525-CS, University of Bonn 1988.

[DK 89] Dahlhaus, E., and Karpinski, M. *An Efficient Parallel Algorithm for the Minimal Elimination Ordering (MEO) of an Arbitrary Graph*, Research Report No. TR-89-024, International Computer Science Institute, Berkeley (June 1989); Extended Abstract to appear in Proc. 30th IEEE FOCS (1989).'

[Di 61] Dirac, A. *On Rigid Circuit Graphs*, Abhandlungen Mathematischer Seminare der Universität Hamburg 25 (1961), pp. 71-76.

[FJ 86] Farber, M., and Jamison, R.E. Convexity in Graphs and Hypergraphs, SIAM J. of Algebraic and Discrete Methods 7 (1986), pp. 433-444.

[Ga 74] Găvril, F. *The Intersection Graphs of Subtrees of Trees are exactly the Chordal Graphs*, Journal of Combinatorical Theory, Series B 16 (1974), pp. 47-56.

[Ga 77] Găvril, F. *Algorithms on Clique Separable Graphs*, Discrete Mathematics 19 (1977), pp. 159-165.

[Ha 88] Hafsteinsson, H. *Parallel Sparse Cholesky Factorization*, PhD Thesis, Cornell University (1988).

[HS 58] Hajnal, A. and Surányi *Über die Auflösung von Graphen in vollständige Teilgraphen*, Ann. Univ. Sci. Budapest Eotvos, Sect. Math. 1 (1958), pp. 113-121.

[HY 88] He, X. and Yesha, Y. *Binary Tree Algebraic Computation and Parallel Algorithms for Simple Graphs*, Journal of Algorithms 9 (1988), pp. 92-113

[HM 86] Helmbold, D. and Mayr, E. *Perfect Graphs and Parallel Algorithms*, In International Conference on Parallel Processing, IEEE (1986), pp. 853-860.

[HL 88] Ho, C.W. and Lee, R.C.T. *Efficient Parallel Algorithms for Finding Maximal Cliques, Clique Trees and Minimum Coloring on Chordal Graphs*, Information Processing Letters 28 (1988), pp. 301-309.

[Kl 88] Klein, Ph. *Efficient Parallel Algorithms on Chordal Graphs*, Proc. 29^{th} IEEE FOCS (1988).

[MR 85] Miller, G. and Reif, J.H. *Parallel Tree Contradiction and its Application* In Proc. 26^{th} FOCS (1985), pp. 478-489.

[MW 86] Monma, C.L. and Wei, V.K. *Intersection Graphs of Paths in a Tree*, Journal of Combinatorial Theory, Series B 41 (1985), pp. 141-181.

[NNS 87] Naor, J., Naor, M., and Schäffer, A. *Fast Parallel Algorithms for Chordal Graphs*, Proc. 19^{th} ACM STOC (1987), pp. 355-364.

[Ta 85] Tarjan, R.E. *Decomposition by Clique Separators*, Discrete Mathematics 55 (1985), pp. 221-232.

[TV 80] Tarjan, R.E. and Vishkin, U. *Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time*, Proc. 25^{th} FOCS (1980), pp. 12-20.

[Wh 84] Whitesides, S.H. *A Method for Solving certain Graph Recognition and Optimization Problems with Applications to Perfect Graphs*, In C.Berge and V.Chvâtal, Editors, *Topics on Perfect Graphs*, Annals of Discrete Mathematics 21 (1984), North-Holland , pp. 281-297.

[Ya 81] Yannakakis, M. *Computing the Minimal Fill-In is NP-Complete*, SIAM Journal of Algorithms and Discrete Methods 2 (1981), pp. 77-79