Parallel Algorithms for Chordal Graphs

Philip N. Klein

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-48
July, 1991

Parallel algorithms for chordal graphs

Philip N. Klein*
Computer Science Department
Brown University
Providence, RI 02912

Abstract

This paper is a version of a chapter to be included in *The Synthesis of Parallel Algorithms*, edited by John H. Reif. We describe algorithms, both sequential and parallel, for *chordal graphs*, a class of graphs including *interval graphs*. We start by defining the notions of an interval graph, a chordal graph, a perfect elimination ordering, and an elimination tree. We illustrate these notions by showing how the elimination ordering can be used to solve optimization problems sequentially on chordal graphs. Then we show how the sequential solutions may be efficiently carried out in parallel. The sequential algorithms are due to Gavril. The parallel algorithms are due to Naor, Naor, and Schäffer, and to Klein.

The key to these efficient sequential and parallel solutions is finding a perfect elimination ordering. In the latter part of this chapter, we define a framework for finding an elimination ordering by successive refinement. Working within this framework, we explain the sequential algorithm due to Rose, Tarjan, and Lueker. Then we describe the parallel algorithm due to Klein.

*The author acknowledges support from ONR Grant N00014-88-K-0243 and DARPA Grant N00039-88-C0113 at Harvard University. Additional support provided by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 6320, Amendment 1, and by NSF Research Initiation Award CCR-9012357. Thanks to Tina Cantor, George Lueker, Mark Novick, Ramamurthy Ravi, John Reif, Daniel Robbins, S. Sairam, David Shmoys, and especially Cliff Stein.

1 Preface

The algorithmic study of special kinds of graphs has yielded insights both combinatorial and algorithmic. In this article, we consider two classes of graphs, chordal graphs and interval graphs, that have been the subject of algorithmic study since the work of Lekkerkerker and Boland in 1962. These classes of graphs are rich enough to find application in diverse areas, from biology to VLSI, but are structured enough that efficient algorithms exist to analyze them. Problems such as maximum-weight clique and minimum coloring, which are NP-complete for arbitrary graphs, can be solved in linear time for chordal and interval graphs.

As motivation for studying chordal graphs, we start in Section 2 by considering the problem of recognizing *interval graphs*. The best algorithms for interval graph recognition involve finding the maximal cliques of the input graph. To do so, they make use of the fact that every interval graph is *chordal*. In Section 3, we show how to identify all the maximal cliques of a chordal graph, by using a node-ordering called a *perfect elimination ordering*, or peo. Next we show that the perfect elimination ordering can also be used to solve combinatorial-optimization problems on chordal graphs. In Section 4, we sketch the sequential algorithms for some of these problems and in Section 5 we describe some parallel algorithms.

Finally, we consider the problem of efficiently finding a perfect elimination ordering. In section 6, we describe a framework for solving this problem. In Section 7, we show how the classic sequential algorithm of Rose, Tarjan, and Lueker can be cast within this framework. In Section 8, we give an efficient parallel algorithm for the problem. The reader is referred to the appendix for some elementary definitions.

2 Interval Graphs

In this section, we introduce interval graphs and one method for determining whether a given graph is an interval graph. A key component of this method is the identification of the maximal cliques of the given graph. In the next section, we show how the maximal cliques can be identified. We only touch on the other component of the recognition method; the reader is directed to other references for more details.

Interval graphs arise in modelling overlapping intervals of a line; each interval is represented by a node, and two nodes are adjacent if the represented intervals overlap. Figure 1 illustrates an example. The biologist Seymour Benzer proposed an examination of the structure of the gene to determine whether its subelements are arranged linearly; the analysis entailed determining whether an experimentally determined graph of correlations between mutations was in fact an interval graph (it was). Motivated by the biological application, Fulkerson and Gross took up the problem of determining whether a graph is an interval graph.

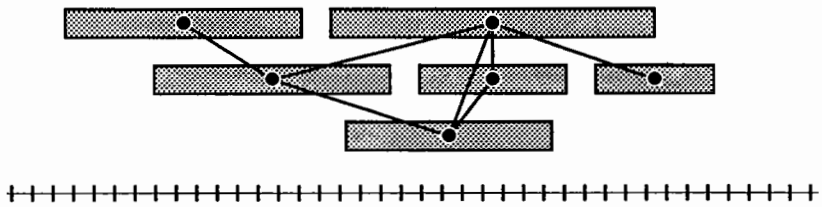

Figure 1: An example of an interval graph. For clarity, the intervals are drawn one above another. Two nodes are adjacent if the corresponding intervals share a point of the real line.

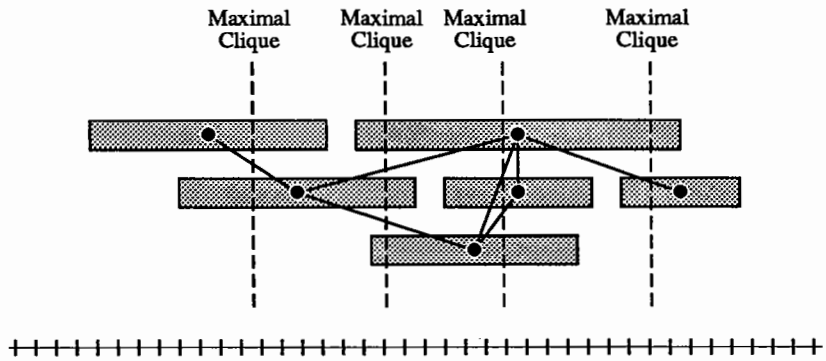

Figure 2: Each maximal clique corresponds to a point on the real line. The ordering of these points corresponds to an ordering of the maximal cliques such that for each node, the maximal cliques containing the node are consecutive.

The intuitive basis for their interval graph recognition algorithm was the following observation. If a graph G corresponds to a collection of overlapping intervals of the real line $\mathbf{R}$, distinct maximal cliques of G correspond to disjoint nonempty intervals of $\mathbf{R}$. Hence the set of maximal cliques of G inherit a natural order from the interval representation of G. Based on this observation, it is not hard to prove the following theorem, illustrated in Figure 2.

Theorem 2.1 (Fulkerson and Gross) *A graph is an interval graph if and only if there is an ordering of its maximal cliques so that for any node of the graph, the maximal cliques containing that node are consecutive.*

Observe that such an ordering of maximal cliques of a graph actually yields an interval representation of the graph. Identify the maximal cliques with distinct points on the real line, and associate with each node v the interval spanning the maximal cliques that contain v. Then the intervals associated with two nodes overlap if and only if there is a maximal clique containing the two nodes, which holds if and only if the two nodes are adjacent.

In using this theorem as the basis for an algorithm, we come up against two problems: finding the maximal cliques, and finding a good ordering for them. The key to finding the maximal cliques is the following property, possessed by all interval graphs:

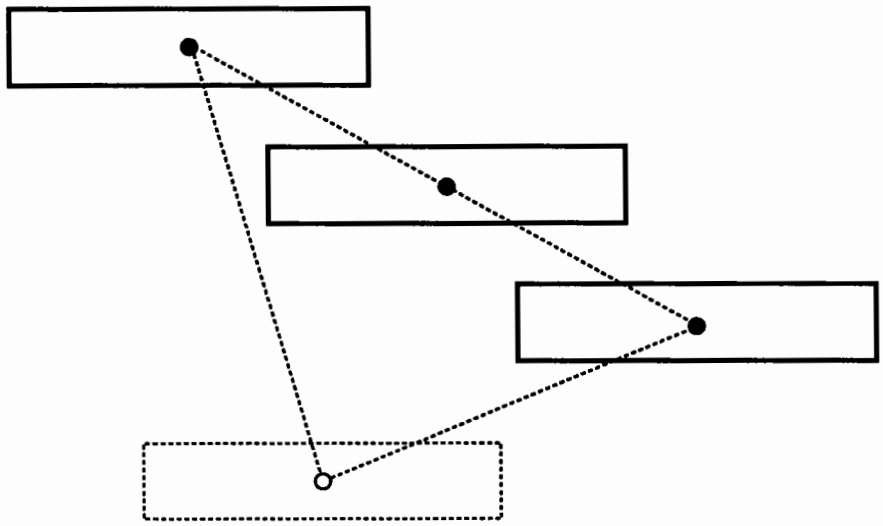

Figure 3: Three intervals are laid down to form a path. A fourth interval can be laid down to join the endpoints of the path only if it intersects all three intervals.

Chordality: Every cycle of length greater than three has a *chord*, an edge connecting two non-consecutive nodes of the cycle.

It is easy to see that every interval graph has this property. First, lay down three (or more) intervals to form a path, as shown in Figure 3. There is no way to lay down a fourth interval overlapping the end-intervals but not the middle one. We shall see in the next subsection how to find the maximal cliques of any graph possessing the chordality property. Of course, if the input graph did not have this property, Fulkerson and Gross could immediately conclude that the input graph was not an interval graph.

Once we have in hand the maximal cliques of the input graph, it remains to determine whether there existed a good ordering of the cliques. The abstract problem is to order some items $C_1, \ldots, C_k$ subject to consecutivity constraints, represented by subsets $S_1, \ldots, S_t$ of the set $\{C_1, \ldots, C_k\}$ of items to be ordered. For each subset S_i, we require that the elements of S_i form a consecutive subsequence of the ordered sequence of items. For example, suppose we wish to order the items A, B, C, D, E, F subject to the constraints $S_1 = \{A, C, E\}$, $S_2 = \{A, C, F\}$, and $S_3 = \{B, F, D\}$. Two consistent orderings are $BDFACE$ and $DBFCAE$; there are six others.

Fulkerson and Gross gave an algorithm based on linear algebra for solving the ordering problem. Their algorithm required $O(k^3)$ time. Years later, Booth and Lueker [6] developed a data structure, called the PQ-tree, for more efficiently solving this problem. The PQ-tree is a tree of size $O(k)$ for implicitly representing all orderings of $C_1, \ldots, C_k$ subject to consecutivity constraints. Given a PQ-tree T and a new constraint S_i, Booth and Lueker showed how to incorporate the new constraint into T, further restricting the set of orderings represented by T, in time proportional to $|S_i|$. Thus, starting from a PQ-tree representing

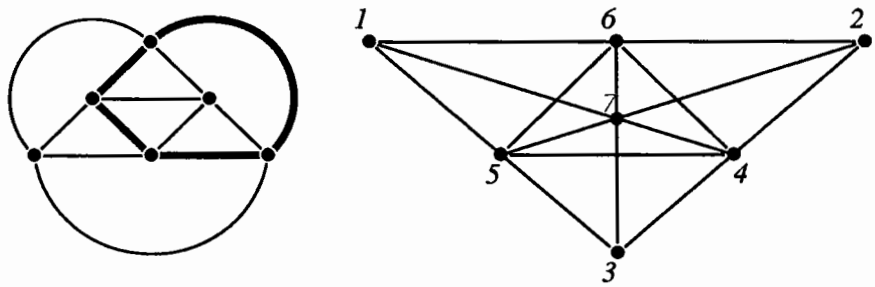

Figure 4: The first graph is nonchordal, and a chordless cycle is highlighted. The second graph is chordal, and a perfect elimination ordering is given. Note that the first graph has no simplicial nodes; in the second graph, nodes 1,2, and 3 are all simplicial. The clique associated with node 1 is $C(1) = \{1, 5, 6, 7\}$. The clique associated with node 5 is $C(5) = \{5, 6, 7\}$. The first clique is maximal; the second is not—it is contained in the first.

all possible orderings, one can introduce the constraints $S_1, \ldots, S_t$ one by one, and obtain after $O(k + \sum_i^t |S_i|)$ time a PQ-tree representing the set of orderings consistent with all the constraints (possibly an empty set). (Klein [28] gave a parallel algorithm for introducing all these constraints at once into a PQ-tree.)

We can apply the PQ-tree to the ordering problem arising in recognizing interval graphs. If the input graph satisfies the chordality property, then, as we shall see, the number k of maximal cliques is at most the number n of nodes of the graph, and the sum of the sizes of the contraints S_i is at most $n + m$. Consequently, use of the PQ-tree yields a linear-time sequential algorithm (and a linear-processor parallel algorithm) for finding a good ordering, if one exists. In the next section, we examine more closely the chordality property, and show how it leads to efficient identification of the maximal cliques.

3 Chordal graphs

The property of chordality seems quite innocent but in fact characterizes a rich and important class of graphs, *chordal* graphs. These graphs are also called *triangulated* graphs [41], because each minimal cycle is a triangle, and *rigid circuit* graphs, because every cycle is (fancifully) made "rigid" by its chords. They are called *perfect elimination* graphs, because they represent the nonzero structure of symmetric matrices for which Gaussian elimination need not introduce a nonzero entry.[1] Figure 4 illustrates a nonchordal and a chordal graph.

Based on a theorem of Dirac on chordal graphs [15], Fulkerson and Gross (and also Rose [40]) gave a characterization of chordal graphs that remains today the most algorithmically useful characterization. Dirac showed that every chordal graph has a *simplicial* node, a

[1]Given an $n \times n$ symmetric matrix A, the corresponding graph has nodes $1, \ldots, n$, and an edge $\{i, j\}$ for each nonzero entry A_{ij}.

node all of whose neighbors form a clique. It follows easily from the chordality property that deleting nodes of a chordal graph yields another chordal graph. That is, any node-induced subgraph of a chordal graph is chordal. In particular, deleting a single node yields a chordal graph. This observation leads to the following algorithm: find a simplicial node, delete it, and recurse on the resulting graph, until no nodes remain. By Dirac's theorem, this procedure successfully terminates on chordal graphs. Conversely, if the graph contains an unchorded k-cycle with $k \geq 4$ (e.g., the first graph of Figure 4), no node in that cycle will ever become simplicial. (For any node in the cycle, its neighbors in the cycle are not adjacent.) Thus the procedure succeeds for chordal graphs, and chordal graphs alone.

But the procedure provides more than a test for chordality; it unfolds the structure of the chordal graph. For example, it provides us with all the maximal cliques. For each node v_i, let $C(v_i)$ be the set consisting of v_i together with v_i's higher-numbered neighbors—those neighbors that are still in the graph when v_i is deleted by the procedure. The procedure ensures that $C(v_i)$ is a clique. It is easy to see that the set of cliques $C(v_i)$ for $1 \leq i \leq n$ include all the maximal cliques. Let C be any maximal clique, and let v_i be the first node of C to be deleted. Then every other node of C is one of the neighbors of v_i remaining in the graph, so C is contained in $C(v_i)$. Since C is maximal, it must be that $C = C(v_i)$. We have proved that every maximal clique occurs as $C(v_i)$ for some v_i. (The reader is encouraged to try applying this argument to any maximal clique in a chordal graph, e.g. the second graph in Figure 4.) Note that some cliques $C(v_i)$ are not maximal. However, it is easy to determine the set of v_i for which $C(v_i)$ is a maximal clique, and we discuss this step in Subsection 5.2.

The above argument shows also that there are at most n maximal cliques in a chordal graph. Moreover, we can infer that if S_i is the set of maximal cliques containing the node v_i, then the sum $\sum_i^n |S_i|$ is at most $n + m$. First, v_i is the minimum-numbered node in at most one maximal clique. Second, the number of maximal cliques of which v_i is a non-minimum member is bounded by the number of edges connecting v_i to a lower-numbered neighbor. The sum of these numbers over all nodes v_i is $n + m$.

The node-ordering provided by the procedure has many other uses as well. For example, Rose used it to establish a connection between chordal graphs and symmetric linear systems. For any $n \times n$ symmetric matrix, there is a corresponding n-node undirected graph—the graph contains the edge $\{i, j\}$ if and only if the ij entry of the matrix is nonzero. Rose showed that if the graph is chordal, a linear system associated with the matrix can be solved via Gaussian elimination *without introducing new nonzero entries.*[2] One simply eliminates variables in in the order in which corresponding nodes are deleted by the procedure. For this reason, such an ordering is called a *perfect elimination ordering* (peo) or perfect elimination scheme.

The original peo algorithm seems to require $\Omega(nm)$ work; at each of n stages, a simplicial node must be located. Rose, Tarjan, and Lueker, however, came up with a linear-time algorithm [41] for constructing a peo. It follows that we can recognize interval graphs in

[2]Rose also required that the matrix be positive semidefinite, so that the elimination process is numerically stable regardless of the order of elimination.

linear time. We use Rose, Tarjan, and Lueker's linear time peo algorithm to help identify the maximal cliques of the input graph, and then use Booth and Lueker's PQ-tree to find a good ordering of the maximal cliques.

We discuss Rose, Tarjan, and Lueker's sequential peo algorithm and a parallel peo algorithm in Sections 7 and 8.

4 Sequential optimization algorithms for chordal graphs

With the maximal cliques provided by the perfect elimination ordering, Fulkerson and Gross were able to carry out interval graph recognition. Gavril [19] carried it a step further. He saw that the perfect elimination ordering was the basis for solving certain optimization problems on chordal graphs. First, observe that included among the maximal cliques is certainly the *maximum* clique. In fact, for any non-negative node-weights, a maximum-weight clique is also among the maximal cliques. Thus having a peo enables one to easily find the maximum-weight clique.

Gavril also gave algorithms, all based on the peo, for finding a maximum independent set, a minimum coloring (i.e. a minimum covering of the the graph by independent sets), and a minimum covering by cliques. If one is given a perfect elimination ordering $v_1 \ldots v_n$, the algorithms are easy to carry out. For the remainder of this section, we assume a perfect elimination ordering has been computed. We call a node *low* or *high* according to its position in the ordering. For example, the first node in the ordering, v_1, is said to be the lowest node, and the other nodes are all higher than v_1.

To find a maximum independent set, use a greedy approach to build up an independent set I node by node. Start by placing the lowest node v_1 in I, and at each subsequent step add to I the lowest remaining node that is not adjacent to any node in I. This procedure always yields a *maximal* independent set; in fact, Gavril proved that it yields a maximum independent set. For whenever a node v_i is added to I, the nodes it rules out—the neighbors of v_i not already ruled out—are higher neighbors, and hence form a clique $C(v_i)$ with v_i. When the procedure terminates, every node has either been placed in I or been ruled out by such a node; thus, we have covered the graph with cliques $C(v_i)$, one for each node v_i added to I. If there were a larger independent set than I, the pigeonhole principle would imply that two of its nodes would land in one of the cliques of our clique cover, a contradiction since no two nodes of an independent set are adjacent. Thus I is a *maximum* clique. By the same token, if there were a smaller clique cover, two of the nodes of I would fall in one of the cliques, so the clique cover we have found is minimum.

This last bit of argument relies on a *min-max inequality*, one of the many that pervade combinatorics:

$$\text{maximum independent set} \leq \text{minimum clique cover} \tag{1}$$

Gavril's algorithm shows that for any chordal graph, (1) is satisfied with equality. In fact,

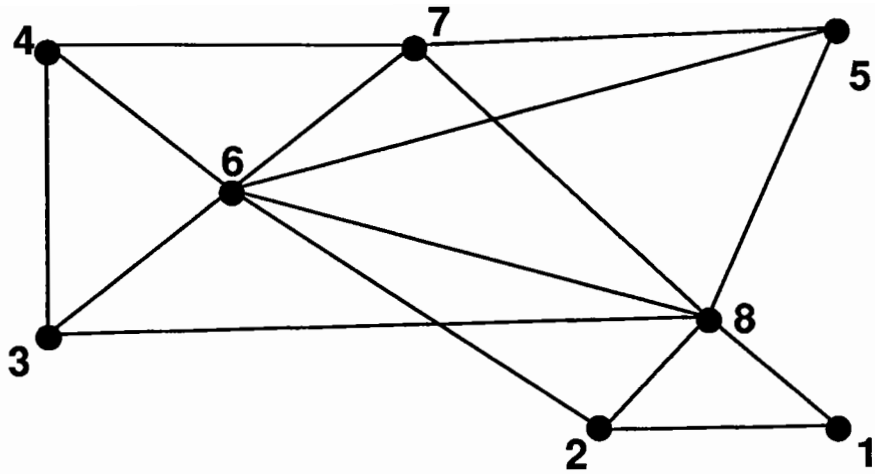

Figure 5: The independent set algorithm applied to the graph shown first places node 1 in I, and deletes 2 and 8. Then it places 3 in I, and deletes 4 and 6. Finally, it places 5 in I, and deletes 7. Thus the final independent set is $I = \{1, 3, 5\}$. The corresponding clique cover is $C(1) = \{1, 2, 8\}$, $C(3) = \{3, 4, 6, 8\}$, and $C(5) = \{5, 6, 7, 8\}$.

if G is a chordal graph, then every node-induced subgraph of G is also chordal, so (1 holds with equality for every node-induced subgraph of G.

A graph all of whose node-induced subgraphs satisfy (1) with equality is called a *perfect* graph. Perfect graphs have long been a focus of combinatorial and algorithmic study. Although we know of no algorithmically useful characterization of perfect graphs, they contain as subclasses a variety of well-studied classes of graphs, including comparability graphs, line graphs, and, of course, chordal and interval graphs. Lovász's celebrated Perfect Graph Theorem states that the node-induced subgraphs of a graph all satisfy (1) with equality if and only if they satisfy the following inequality with equality.

$$\text{maximum clique} \leq \text{minimum covering by independent sets (coloring)} \qquad (2)$$

Since a clique is the complement of an independent set, the Perfect Graph Theorem states that a graph is perfect if and only if its complement is perfect.

Returning to the subclass of interest, chordal graphs, we might be tempted by (2) to conjecture that there is a minimum coloring algorithm for chordal graphs. In fact, Gavril gives such an algorithm, again based on a greedy approach. We use the positive integers as colors. Starting at the last node v_n of the peo, and working backwards, assign to each v_i in turn the minimum color not assigned to its higher neighbors. Suppose the node receiving the highest color k is v_i. Then colors 1 through $k-1$ were already assigned to the higher neighbors of v_i. But then v_i and its higher neighbors form a clique $C(v_i)$ of size k. Since we have found a coloring using only k colors, the inequality (2) implies that our coloring is minimum (and k is the size of the largest clique).

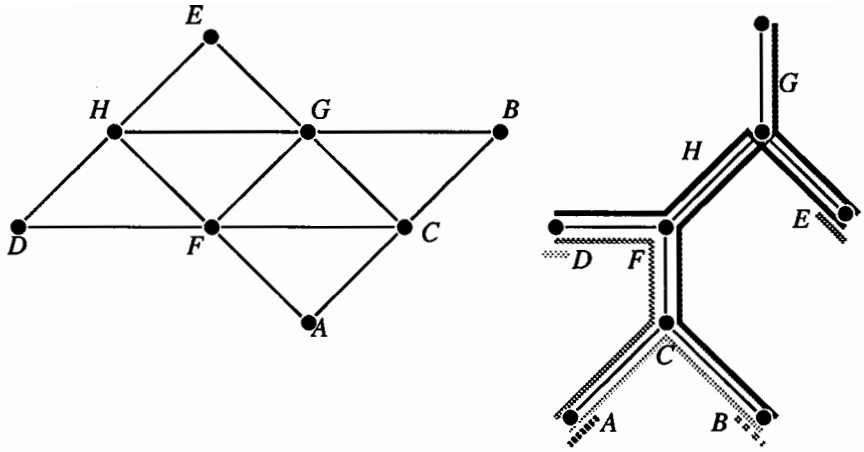

Figure 6: Illustrated are a chordal graph, and a tree whose subtrees represent the chordal graph. Each highlighted subtree represents a node of the chordal graph; two nodes of the chordal graph are adjacent if and only if the corresponding subtrees share nodes of the tree.

Gavril's algorithms for maximum-weight clique, maximum independent set, minimum coloring, and minimum-clique cover can all be implemented in linear time, if a perfect elimination ordering is provided.

5 Parallel optimization algorithms

Naor, Naor, and Schäffer [35] first gave *parallel* algorithms for these optimization problems. They relied, largely, not on the peo, but on another characterization of chordal graphs. It turns out that chordal graphs are exactly the intersection graphs of subtrees of trees. That is, for a tree T and subtrees $T_1, \ldots, T_n$ of T there is a graph whose nodes correspond to subtrees T_i, and where two nodes are adjacent if the corresponding subtrees share a node of T. Such a representation is depicted in Figure 6.

The graphs arising in this way are exactly the chordal graphs, and interval graphs arise when the tree T happens to be a simple path. The *clique tree* of a chordal graph G is a special tree-and-subtrees representation of G, in which there is a node in T for each maximal clique, and, for each node v_i of G, the corresponding subtree T_i in the representation contains the nodes corresponding to the maximal cliques containing v_i.

Naor, Naor, and Schäffer's parallel approach was to first find all the maximal cliques of G, using a simple but expensive divide-and-conquer strategy, and then construct the clique tree representation using the fact that every tree has a good separator. Their algorithm for finding all maximal cliques used n^4 processors to achieve $O(\log^3 n)$ time or n^5 processors to achieve $O(\log^2 n)$ time. Given the maximal cliques, their algorithm for constructing the clique tree used n^3 processors to achieve $O(\log^2 n)$ time on a CREW PRAM. The disadvantage of this

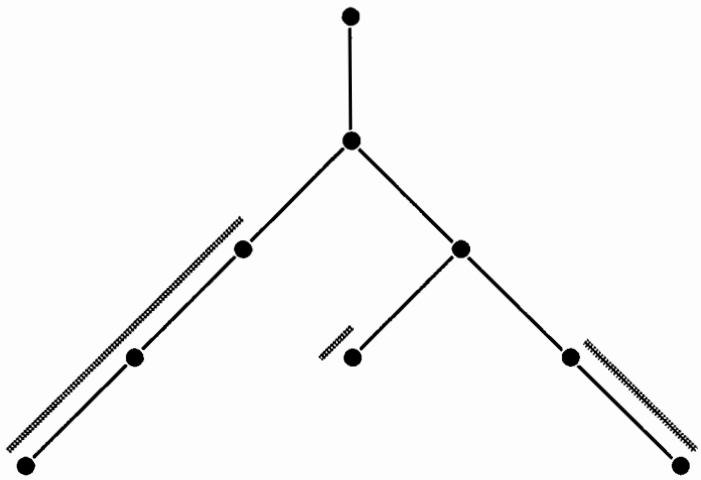

Figure 7: The rooted tree illustrated has three terminal branches, one per leaf, each consisting of the path from the leaf up to but not including the first ancestor with two or more children.

approach is that the number of processors required is quite large.

Naor, Naor, and Shäffer then showed how to use the clique tree as the basis for parallel algorithms to find a perfect elimination ordering, a maximum-weight clique, a minimum coloring, a minimum clique cover, a maximum independent set, and a maximum-weight independent set. The processor bounds range from n^2 for the first problem to n^4 for the fourth.

A key technique they developed was that of processing a tree by *pruning terminal branches*. A *terminal branch* of a tree is a maximal path in the tree consisting of degree-two nodes and a leaf; an example is illustrated in Figure 7. Naor, Naor, and Schäffer observe that removal of all terminal branches from a tree yields a tree with at most half the number of leaves. Thus $\lfloor \log n \rfloor + 1$ stages of terminal branch pruning reduces an n-node tree to nothing. The methodology of Naor, Naor, and Schäffer for solving a particular chordal graph problem was to develop a parallel subroutine for solving the subproblem in the case where the clique tree is a *path* (i.e. the graph is an interval graph), and repeatedly apply this subroutine to each of the terminal branches in parallel. The number of iterations is $O(\log n)$. The idea of pruning terminal branches is simple yet powerful, and will probably find its way into other parallel algorithms that operate on trees.

Klein [29] took a different approach to solving the optimization problems in parallel. The basis for these parallel algorithms, was, as for the sequential algorithms, the perfect elimination ordering. In Section 8, we describe an *efficient* parallel algorithm for finding a peo—the algorithm uses only $(n+m)/\log n$ processors to achieve $O(\log^2 n)$ time on a CRCW PRAM. Thus the total work done by this algorithm is within a logarithmic factor of linear. In this light, the result can be viewed as a parallel analogue to Rose, Tarjan, and Lueker's result on finding a peo in linear time.

In the remainder of this section, we show that, if a peo is provided, the problems of maximum-weight clique, minimum coloring, maximum independent set, and minimum clique cover can all be solved efficiently in parallel, again in $O(\log^2 n)$ time using $(n+m)/\log n$ processors of a CRCW PRAM.[3] Thus Gavril's results also have analogues in the parallel realm.

There is also a parallel analogue of Booth and Lueker's PQ-tree data structure: given a PQ-tree T and some constraints $S_1, \ldots, S_t$, the constraints can be incorporated into T quickly and efficiently in parallel. Combining this algorithm with the parallel peo algorithm yields an interval graph algorithm: given a graph G, the algorithm constructs an interval representation (if one exists) in $O(\log^2 n)$ time using $O((n+m)/\log n)$ processors.

5.1 The elimination tree

The algorithms we describe in this section, like those of Naor, Naor, and Schäffer, tend to rely on a tree to guide the algorithm. In the former case, however, the tree was the *elimination tree* determined by the peo. Given connected graph G and a peo $\sigma = v_1 \ldots v_n$ for G, the elimination tree $T(G_\sigma)$ is defined by choosing a parent $p(v_i)$ for each node v_i of G except the last, v_n. The choice of parent is given by

$$p(v_i) = \text{ lowest neighbor } v_j \text{ such that } j > i \tag{3}$$

The parent has a higher number than the child, so the resulting directed graph is acyclic. Since each node but one has a parent, the directed acyclic graph is indeed a tree.

It turns out that the elimination tree determined by a peo of a graph G was in fact a *depth-first search tree* for the graph G. For any rooted spanning tree T of G, a *cross-edge* of T is an edge of $G - T$ whose endpoints are not ancestors of each other in T. Figure 9 illustrates this definition. If T has no cross-edges, we call T a *depth-first search tree* for G.

Theorem 5.1 *Let G be a chordal graph. The elimination tree $T(G_\sigma)$ determined by a* peo *σ of G has no cross-edges.*

To prove the theorem, we introduce the notion of an end-high path. For a graph G with a total ordering on its nodes, an *end-high path* is a path in G whose endpoints are higher in the total ordering than all internal nodes.

Lemma 5.2 (Rose, Tarjan, and Lueker) *For a chordal graph G with the nodes ordered by a* peo*, every end-high path has adjacent endpoints.*

[3] In fact, the first problem is nearly trivial to solve, once the peo is obtained; the time bound is only $O(\log n)$.

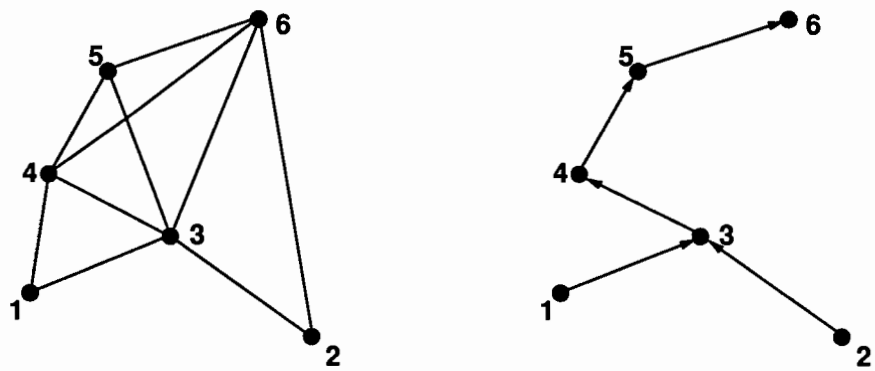

Figure 8: Given a graph G with a peo σ, we construct the elimination tree $T(G_\sigma)$ by choosing a parent for each node according to the rule (3): the parent of v is the lowest neighbor higher than v.

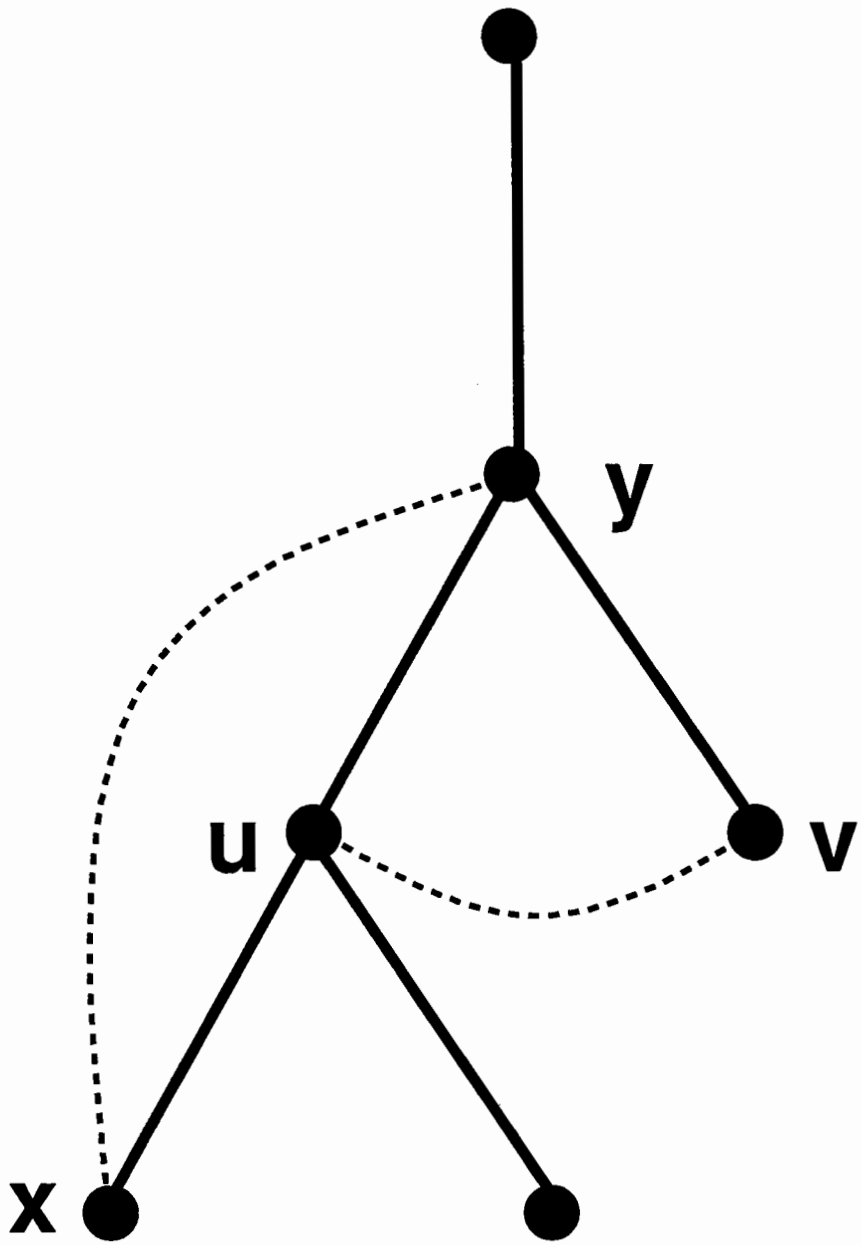

Figure 9: The edge $\{x, y\}$ is not a cross-edge because y is an ancestor of x. The edge $\{u, v\}$ is a cross-edge because neither u nor v is an ancestor of the other.

Proof: Suppose there is an end-high path with endpoints x and y. We must show that x and y are adjacent. Let P be the shortest end-high path with these endpoints. If P consists of a single edge between x and y, we are done. Therefore, assume for a contradiction that P contains internal nodes. Let v be the lowest internal node in P. The node that precedes v in P and the node that follows v in P are clearly neighbors of v, since P is a path. By choice of v, they are higher than v in the peo. Hence by definition of a peo, they are adjacent. The edge between them is a "short-cut" that allows us to skip v. That is, removing v from the path P yields a shorter end-high path. This contradicts the minimality of P. ∎

The proof of Theorem 5.1 follows easily from Lemma 5.2.

Proof of Theorem: Suppose for a contradiction that $\{v, y\}$ is a cross-edge. Let x be the highest ancestor of v that is less than y in the peo. There is an end-high path starting at y, going to v and then up through the tree to x. Hence by Lemma 5.2, x and y are adjacent. Since $\{v, y\}$ is a cross-edge, y is not an ancestor of v. In particular, x's parent $p(x)$ is not y. By choice of x, therefore, $p(x)$ is higher in the peo than y. But $p(x)$ is defined to be the lowest neighbor of x that is higher than x, and hence $p(x)$ must be lower than y. We have a contradiction. ∎

Another important structural property of the elimination tree is stated by the following simple corollary to Lemma 5.2.

Corollary 5.3 *Suppose u is a descendent of v in the elimination tree determined by a* peo. *Then every neighbor of u that is a proper ancestor of v is also a neighbor of v.*

Proof: There is an end-high path starting each such neighbor of u, proceeding through u and up the tree to v. Hence by Lemma 5.2, each such neighbor is adjacent to v. ∎

The fact that an elimination tree is a depth-first search tree makes it a useful tool for choosing a chordal graph separator. In particular, we shall show that a node-separator for the elimination tree corresponds to a clique-separator for the chordal graph. A *clique-separator* for a graph G is a clique K in G such that every component of $G - K$ has no more than half the nodes of G. The fact that every chordal graph has a clique-separator was first shown by Gilbert, Rose, and Edenbrandt [23]; they gave a linear-time sequential algorithm for finding such a separator. Using the elimination tree, Klein showed how to find a clique-separator efficiently in parallel.

The key lemma states that removing a node v and its higher neighbors separates each subtree rooted at a child of v from the rest of the graph. For a rooted tree T containing a node v, let T_v denote the subtree of T rooted at v.

Lemma 5.4 *Let G be a chordal graph. Let $T = T(G_\sigma)$ be the elimination tree determined by a* peo *σ of G. Let $\widehat{v}$ be a node of G, with children $v_1, \ldots, v_k$ in T. Let K be the clique of G consisting of $\widehat{v}$ and its higher neighbors. Then $G[T_{v_i}]$ is a connected component of $G - K$, for $i = 1, \ldots, k$.*

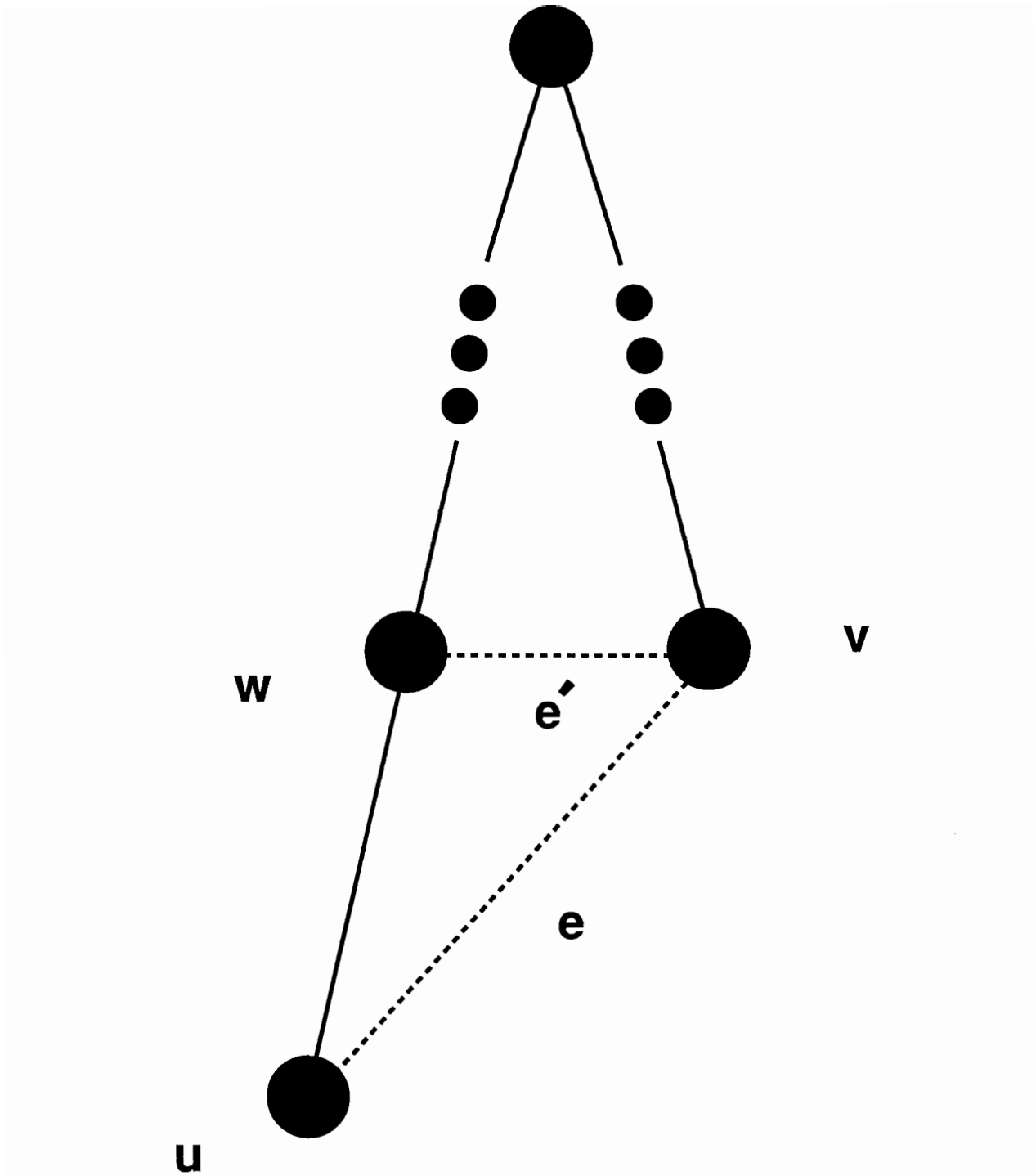

Figure 10: The existence of a cross-edge e connecting u and v implies the existence of a cross-edge e' connecting w and v.

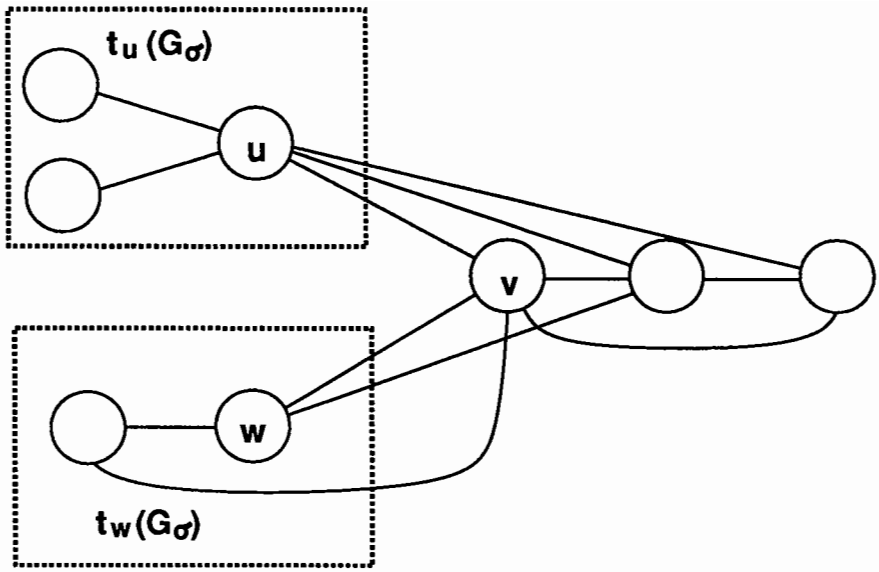

Figure 11: When the node v and its higher neighbors are removed, the subtrees rooted at children of v become separated from each other and from the remainder of the graph.

Proof: : To see that $G[T_{v_i}]$ is connected in $G-K$, note that edges in T_{v_i} are edges in G, and hence T_{v_i} is a spanning tree of $G[T_{v_i}]$. None of the nodes in T_{v_i} are in K, so $G[T_{v_i}]$ remains connected when K is removed from G.

Suppose there is an edge between a node v in T_{v_i} and a node w not in $K \cup T_{v_i}$. The edge cannot be a cross-edge in T, so w must be an ancestor of v; since w is not in T_{v_i}, it must be an ancestor of $\widehat{v}$ as well. Hence by Corollary 5.3, w must be adjacent to $\widehat{v}$, so w belongs to K, a contradiction. ∎

As a corollary to Lemma 5.4, we can show that a chordal graph has a clique whose removal breaks the graph into pieces of at most half the size. Let the node $\widehat{v}$ of Lemma 5.4 be the lowest node in the elimination tree having more than $n/2$ descendents. Then every component $G[T_{v_i}]$ has at most $n/2$ nodes, but together these components comprise at least $n/2$ nodes. Hence the clique consisting of $\widehat{v}$ together with its higher neighbors is a separating clique.

5.2 Parallel recognition of chordal graphs, and identification of maximal cliques

As in the sequential case, the most efficient parallel algorithm for determining whether a graph G is chordal is to apply the parallel peo algorithm to G, and then determine whether the resulting node-ordering is in fact a peo. We carry out the second step using a parallel version of a technique from [41].

To determine whether a given node-ordering σ is a peo of G, we start by constructing the elimination tree $T(G_\sigma)$. Then each node v sends to its parent $p(v)$ in the elimination tree a list of v's higher neighbors (excluding $p(v)$). Finally, each node w sorts the elements of all the lists it received, together with w's own adjacency list, and verifies that it is a neighbor of every node on every list it received.

Claim 5.5 *The node-ordering is a* peo *if and only if no verification step fails.*

Proof: Suppose the node-ordering is a peo. Then for every node v, the higher neighbors of v form a clique. In particular, the parent of v is adjacent to all v's other higher neighbors. Thus every verification step succeeds.

Suppose no verification step fails. We claim that for each node v, the higher neighbors of v form a clique. The proof is by reverse induction on the depth d of v in the tree $T(G_\sigma)$. The claim is trivial for $d = 0$, because the root has no higher neighbors. Suppose the claim holds for d, and let v be a node at depth $d + 1$. By the inductive hypothesis, $p(v)$ and its higher neighbors form a clique K. By the success of $p(v)$'s verification step, every higher neighbor of v is a neighbor of $p(v)$, and hence lies in K. This proves the induction step. ∎

The claim shows that we can determine whether a given ordering is a peo of G. The total number of items in all lists is $\sum_v(d(v)-1)$, where $d(v)$ is the number of higher neighbors of v. Equivalently, , $d(v)$ is the number of edges from v to higher neighbors; hence $\sum_v d(v) \leq m$. We can therefore allocate to each node w a number of processors equal to the number of items w must sort. All the sorts can then be carried out in $O(\log n)$ time. Thus the time for carrying out verification is $O(\log n)$ using $n + m$ processors using a CREW PRAM.

Assuming the node-ordering is in fact a peo, we can use a similar algorithm to find the set of nodes v such that $C(v)$, the set of nodes consisting of v and its higher neighbors, is a maximal clique. Each node v computes the number $d(v)$ of higher neighbors it has, and sends the value $d(v)$ to its parent. Then each node w compares the values it received to $d(w)$, and concludes that $C(w)$ is a maximal clique only if none of the values it received exceeds $d(w)$.

The correctness of this procedure relies on the fact, immediate from Corollary 5.3, that every higher neighbor of a node v is either v's parent $p(v)$ or a higher neighbor of $p(v)$. If the number of higher neighbors of v exceeds the number of higher neighbors of $p(v)$, then the clique $C(p(v))$ is strictly contained in the clique $C(v) = \{v\} \cup C(p(v))$, and is therefore not maximal.

Conversely, suppose some clique C strictly contains $C(w)$; We shall show that w received a value exceeding $d(w)$. Let u be the lowest node in C. If u was higher than w, then C would not include w, and if u equaled w, C would be contained in $C(w)$, as we showed in Section 3. Hence u must be lower than w. Since u and w are together in a clique, there is an edge between them, and so by Lemma 5.1, u is a descendent of w. Let v be the child of w that is an ancestor of u. By Corollary 5.3, the neighbors of u that are proper ancestors of v (which include the nodes of $C(w)$) are also neighbors of v. Hence $d(v) \geq |C(w)| = 1 + d(w)$.

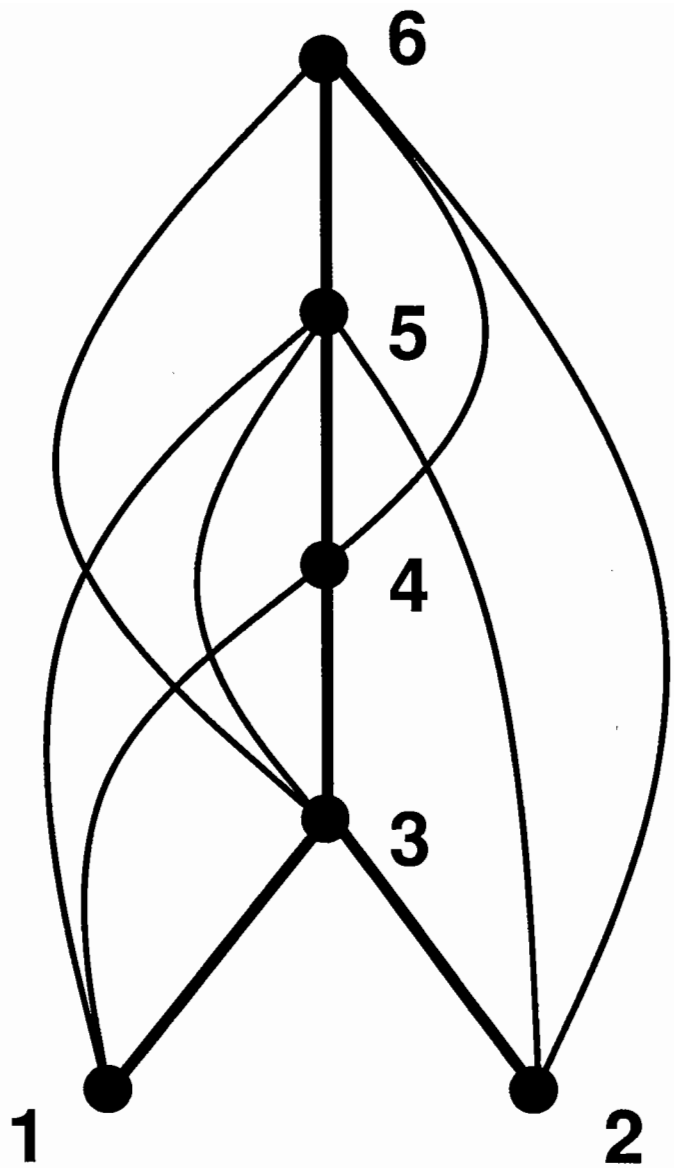

Figure 12: Each node v sends to its parent $p(v)$ a list of v's higher neighbors (excluding $p(v)$). Each node w verifies that it is a neighbor of every node on every list it received. For example, in the graph depicted, 1 sends to its parent 3 the list $\{4, 5\}$, and 2 sends to its parent 3 the list $\{5, 6\}$. The node 3 verifies that 4, 5, and 6 are among its neighbors.

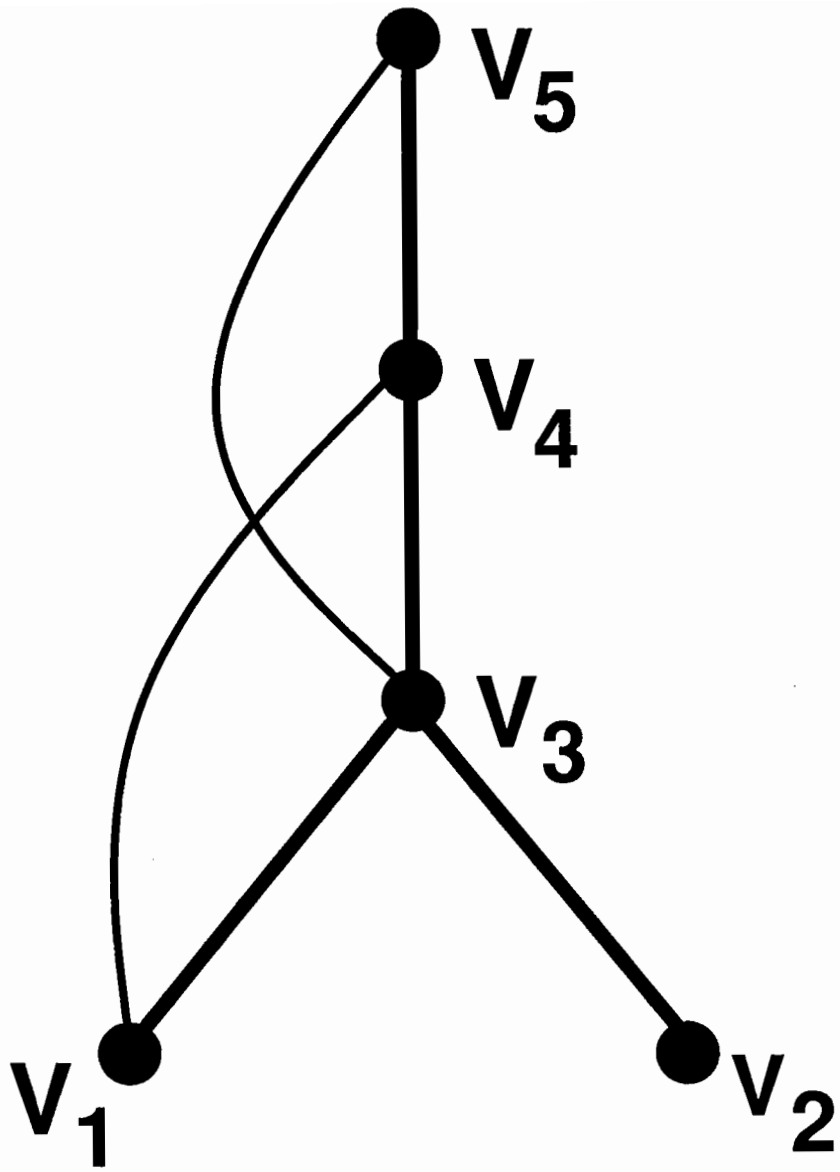

Figure 13: Identification of maximal cliques. For the graph depicted, v_3 has two higher neighbors, v_4 and v_5, so $d(v_3) = 2$. Similarly, $d(v_1)$ is 2, $d(v_2)$ and $d(v_4)$ are 1, and $d(v_5)$ is 0. The node v_3 receives the numbers 2 and 1 from its children v_1 and v_2; neither of these numbers exceeds $d(v_3)$, so we conclude that $C(v_3) = \{v_3, v_4, v_5\}$ is a maximal clique. The nodes v_1 and v_2 have no children, so the condition is trivially satisfied, and we conclude that the cliques $C(v_1) = \{v_1, v_3, v_4\}$ and $C(v_2) = \{v_1, v_3\}$ are maximal.

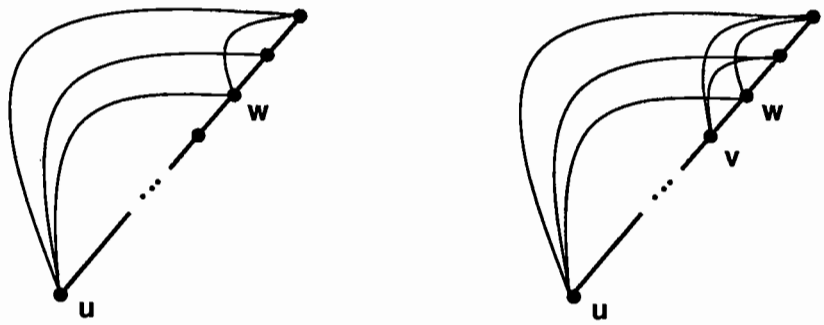

Figure 14: Proof of correctness of the procedure for identifying maximal cliques. If $C(w)$ is not a maximal clique, then w has a descendent u such that $C(u)$ contains $C(w)$. Let v be the child of w that has u as a descendent. Then $C(v)$ contains $\{v\} \cup C(w)$, so $d(v) > d(w)$. We conclude that w receives from one of its children (namely v) a number exceeding $d(w)$.

5.3 Maximum-weight clique

Suppose each node is assigned a non-negative weight. As Gavril observed, any maximum-weight clique is maximal, so finding all the maximal cliques is tantamount to finding a maximum-weight clique. For each clique $C(v)$, compute the total weight of its nodes, and compare the total weights to find the maximum. This procedure takes $O(\log n)$ time using $(\sum_v |C(v)|)/\log n \leq (n+m)/\log n$ processors of a CREW PRAM.

5.4 Breadth-first search trees

To obtain a breadth-first search tree of G, we construct a tree similar to the elimination tree, by choosing the parent of each node v (except the highest node) to be the highest neighbor of v. Let T be the resulting tree, rooted at the highest node, which we shall denote by r. It is easy to see that T can be computed in $O(\log n)$ time using $n+m$ processors of a CREW PRAM. Our proof that T is a breadth-first search tree relies on two claims.

Claim 5.6 *For each node v the shortest path from v to r is monotonically increasing with respect to the* peo.

Proof: Any subpath whose internal nodes are lower than its endpoints can be replaced by a direct edge between the endpoints, by Lemma 5.2. ∎

For the second claim, let $s(v)$ denote the length of the shortest path in G from v to r.

Claim 5.7 *If w is a descendent of v in the elimination tree, then $s(w) \geq s(v)$.*

Proof: The proof is by reverse induction on the position of w in the peo. The basis, in which $w = r$, is trivial. Otherwise, let w' be the second node on a shortest path from w to r, so

$s(w) = 1 + s(w')$. By Claim 5.6, w' is higher than w, hence an ancestor of w by Lemma 5.1. If w' is a descendent of v, then $s(w') \geq s(v)$ by the inductive hypothesis. If w' is an ancestor of v, then there is an end-high path from v back along tree edges to w, and then forward to w', proving by Lemma 5.2 that v is adjacent to w', and hence that $s(v) \leq 1 + s(w')$. ∎

For each node $v \neq r$, let $q(v)$ be the highest neighbor of v. Any other neighbor w of v is a descendent of $q(v)$ in the elimination tree, so $s(w) \geq s(q(v))$ by Claim 5.7. It follows that $q(v)$ is the second node in a shortest path from v to the highest node of G. Thus the tree defined by $q(\cdot)$ is a breadth-first search tree.

5.5 Maximum independent set

As we discussed in Section 4, Gavril showed that a maximum independent set $\mathcal{I}$ of the chordal graph G is obtained by the greedy maximal-independent-set algorithm when applied to nodes in order of the peo $\sigma = v_1 \ldots v_n$. We review his algorithm. First put v_1 into $\mathcal{I}$, and delete v_1 and its neighbors. Next, put the lowest remaining node in $\mathcal{I}$, and delete it and its neighbors, and so forth. Since in each iteration the lowest possible node v_i is placed in $\mathcal{I}$, all v_i's neighbors at that time are *higher* neighbors. Every deleted node, therefore, is either in $\mathcal{I}$ or is a *higher* neighbor of a node in $\mathcal{I}$. Recall that for each node x, $C(x)$ is a clique consisting of x together with its higher neighbors. Consequently, when the algorithm terminates, the family of cliques $\{C(x) : x \in \mathcal{I}\}$ is a clique cover (a set of cliques whose union contains all the nodes). Because any independent set has size at most that of any clique cover, it follows that the above procedure has identified a *maximum* independent set and a *minimum* clique cover.

We want to simulate Gavril's sequential greedy algorithm in parallel. First suppose that the elimination tree $T(G_\sigma)$ determined by the peo σ is a *path* with leaf x. In this case, we give a simple algorithm PMIS for simulating Gavril's algorithm. For each node v, let $b[v]$ be the lowest ancestor of v in $T(G_\sigma)$ that is not adjacent to v in G (or v, if no such ancestor exists).

Claim 5.8 *The greedy independent set consists of $x, b[x], b[b[x]]$, and so on.*

This set can be determined quickly in parallel using standard pointer-jumping techniques. The implementation shown in Figure 15 requires $O(\log n)$ time, m processors, and $O(m \log n)$ space using a CREW PRAM; use of more sophisticated techniques (e.g. [3], [11]) achieve the same time bound using only $m/\log n$ processors and $O(m)$ space on a EREW PRAM.

Proof of Claim: Suppose we put x into $\mathcal{I}$ and delete the neighbors of x. The node $b[x]$ is by definition the lowest undeleted node. Moreover, we assert that for each undeleted node v, $b[v]$ is undeleted. If $b[v]$ were a neighbor of x, then by Corollary 5.3, $b[v]$ would be a neighbor of v, contradicting the definition of $b[v]$. This argument proves the assertion; the claim follows by induction on the length of the elimination path. ∎

PMIS

P1 For each node v, let $b_0[v]$ denote the lowest ancestor of v that is not adjacent to v (or else v).

P2 For stages $k = 0, \ldots, \lceil \log n \rceil - 1$, for each node v, let $b_{k+1}[v] := b_k[b_k[v]]$.

P3 Mark the leaf x as being in the independent set.

P4 For stages $k = \lceil \log n \rceil - 1, \lceil \log n \rceil - 2, \ldots, 0$, for each marked node v, mark $b_k[v]$.

Figure 15: A simple implementation of the algorithm PMIS for finding a maximum independent set when the elimination tree is a path.

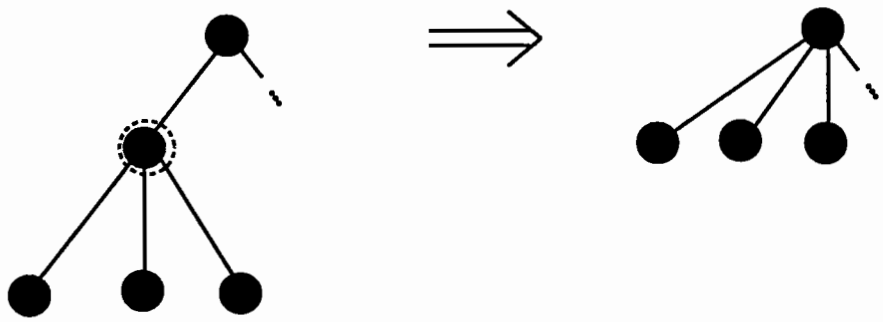

Figure 16: To splice a node out of a tree, remove the node and reattach the node's children to its parent.

To generalize this procedure to the case in which $T(G_\sigma)$ is a tree, we use an idea of Naor, Naor, and Schäffer discussed at the beginning of Section 5: pruning terminal branches. A *terminal branch* of a tree is a maximal path of degree-two nodes ending in a leaf. Naor, Naor, and Schäffer observe that deletion of all terminal branches of a tree yields a new tree with half as many leaves. Therefore, $O(\log n)$ iterations of terminal branch elimination suffice to eliminate the entire tree. By applying this idea to the elimination tree, we obtain a parallel maximum independent set algorithm requiring only $m/\log n$ processors.

Before giving the algorithm, we introduce a bit of tree-surgery, called *splicing*. To *splice* a node v out of a tree T is to remove the node and reattach any children of v to v's parent in T, as illustrated in Figure 16. If a set of nodes are to be spliced from a tree, the resulting tree does not depend on the order in which the nodes are spliced out. In fact, they can all be spliced out at once; for each node v to be spliced out, the children of v are reattached to the lowest ancestor of v that is not to be spliced out.

We now describe the algorithm MIS, shown in Figure 17, for constructing a maximum independent set in a chordal graph G. Let $T(G_\sigma)$ be the elimination tree for G determined

MIS

M1 To initialize, let $\mathcal{I} = \emptyset$ and let T be the elimination tree $T(G_\sigma)$ for G determined by a peo σ.

M2 While T is not empty,

M3 Use the algorithm PMIS to find the greedy maximum independent set $\mathcal{I}_\mathcal{B}$ of the subgraph induced on each terminal branch $\mathcal{B}$ of T.

M4 Add the nodes $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ to $\mathcal{I}$.

M5 Splice out of T the nodes $\bigcup_\mathcal{B}(\mathcal{I}_\mathcal{B} \cup \{\text{neighbors of } \mathcal{I}_\mathcal{B}\})$.

Figure 17: The algorithm MIS to construct a maximum independent set $\mathcal{I}$ in the chordal graph G.

by a peo σ. The algorithm maintains a set $\mathcal{I}$, the independent set under construction, and a tree T, obtained from the elimination tree $T(G_\sigma)$ by splicing out nodes. We prove by induction that the following invariant holds before and after each iteration of the algorithm.

Invariant

(1) $\mathcal{I}$ is an independent set.

(2) Every neighbor of a node of $\mathcal{I}$ is in fact a *higher* neighbor of some node of $\mathcal{I}$.

(3) T is obtained from $T(G_\sigma)$ by splicing out the nodes of $\mathcal{I}$ and their neighbors.

The algorithm terminates when T is empty, at which point $\mathcal{I}$ is an independent set such that every node of G is either in $\mathcal{I}$ or is a higher neighbor of some node in $\mathcal{I}$. Thus, as in Gavril's algorithm, the family of cliques $C(x) = \{x\} \cup \{\text{higher neighbors of } x\}$ for $x \in \mathcal{I}$ is a minimum clique cover, and $\mathcal{I}$ is a maximum independent set.

Initially, $\mathcal{I} = \emptyset$ and T is $T(G_\sigma)$, the elimination tree, so the invariant holds trivially. Suppose the invariant holds through the first k iterations of the algorithm, and consider the $k+1^{st}$ iteration. For each terminal branch $\mathcal{B}$, the algorithm finds a maximum independent set $\mathcal{I}_\mathcal{B}$ of the subgraph induced on $\mathcal{B}$, using PMIS as a subroutine. If two nodes of T lie in different terminal branches, neither is an ancestor of the other in $T(G_\sigma)$, and hence the two nodes are not adjacent, by Lemma 5.1. Thus $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ is an independent set in G, where the union is over all terminal branches of T. Moreover, T contains no neighbors of $\mathcal{I}$ (by part (3) of the invariant), so $\mathcal{I} \cup (\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B})$ is an independent set of G. Thus when the nodes $\bigcup_\mathcal{B} \mathcal{I}_\mathcal{B}$ are added to $\mathcal{I}$ in step M4, part (1) of the invariant remains true.

To show that part (2) remains true, we must prove that every node w that is newly a neighbor of a node in $\mathcal{I}$ is in fact a *higher* neighbor of a node in $\mathcal{I}$. Our simulation PMIS

of Gavril's algorithm on terminal branches $\mathcal{B}$ guarantees this property when w lies on a terminal branch. Suppose therefore that w does not lie on a terminal branch, and let $v \in \mathcal{I}$ be a neighbor of w. Since there are no cross-edges by Lemma 5.1, v is either a descendent or an ancestor of w in $T(G_\sigma)$. The set $\mathcal{I}$ consists only of nodes in terminal branches of T and descendents of such nodes. Hence v must be a descendent of w, and a lower neighbor.

In the last step of an iteration of the algorithm, we splice out of T all nodes newly added to $\mathcal{I}$ and their neighbors. This step ensures that part (3) holds at the end of the iteration.

Having proved that the invariant continues to hold, we now consider the implementation of the algorithm MIS. In step M3, the algorithm must identify the nodes lying in terminal branches of T. An application of the Euler tree technique [45] suffices to determine, for each node v of T, the number of leaf descendents of v. The nodes for which this number is 1 are the nodes in terminal branches. Next, the algorithm must find a maximum independent set in each terminal branch. For each node v in T, $b_0[v]$ is assigned the lowest ancestor w of v in T that is not a neighbor of v, *if* w lies in a terminal branch. Otherwise, $b_0[v]$ is assigned v. As in PMIS, a pointer-jumping technique is then used to mark the nodes $x, b_0[x], b_0[b_0[x]]$, and so on, for all leaves x of T. The marked nodes are added to $\mathcal{I}$ in step M4.

To implement the splicing in step M5, we again use a pointer-jumping technique; for each node v, we compute the lowest ancestor of v in T that is not to be spliced out. Each step can be implemented in $O(\log n)$ time using $m/\log n$ processors and $O(m)$ space. Each iteration removes all nodes in terminal branches and hence reduces the number of leaves in T by a factor of two; consequently, $\lceil \log n \rceil + 1$ iterations suffice, for a total of $O(\log^2 n)$ time.

5.6 Minimum coloring

Gavril showed that applying the greedy coloring algorithm to the nodes of G in reverse order of \$ yields a minimum coloring. In this section, we give a parallel algorithm for minimum coloring. The algorithm is efficient in its use of parallelism—it requires only $(n+m)/\log n$ processors—but takes $O(\log^2 n)$ time on a CRCW PRAM. It requires as input a chordal graph and a perfect elimination ordering.

5.6.1 The color-mapping strategy

To gain intuition, let us consider the problem of coloring a graph G consisting of two overlapping subgraphs $\widehat{H_0}$ and $\widehat{H_1}$. We assume that each edge of G occurs in either $\widehat{H_0}$ or $\widehat{H_1}$—that is, no edge goes between $\widehat{H_0} - \widehat{H_1}$ and $\widehat{H_1} - \widehat{H_0}$. We would like to obtain a minimum coloring of G from minimum colorings of $\widehat{H_0}$ and $\widehat{H_1}$. The difficulty is ensuring that the separate colorings are consistent on the nodes $\widehat{H_0} \cap \widehat{H_1}$ common to the two subgraphs; if this were the case, merging the two colorings would be easy. We would take the color of a node of G to be the color assigned to that node in one of the two separate colorings. The resulting color assignment would not assign the same color to two adjacent nodes, because the nodes are

also adjacent in either $\widehat{H_0}$ or $\widehat{H_1}$.

To make things easier, let us assume that the common nodes form a clique. The advantage is that there is only one way to color the nodes of a clique, up to renaming of colors—each node of the clique must receive a different color.

Our strategy for making the separate colorings consistent, which we call the *color-mapping strategy*, relies on the fact that colors are interchangeable. We start by independently choosing a coloring c_i for each subgraph $\widehat{H_i}$ $(i = 0, 1)$, ignoring the fact that they overlap. We must then carry out a *repair step*, in which the coloring of one subgraph, say $\widehat{H_1}$, is modified to be consistent with the coloring of the other subgraph $\widehat{H_0}$ on the common nodes. The repair step consists in finding a one-to-one mapping of the colors used by c_1 satisfying the following condition:

> *Consistency condition*: the color assigned by c_1 to a common node $v \in \widehat{H_1} \cap \widehat{H_0}$ is mapped to the color assigned to the same node by c_0.

Once we have found such a mapping, we can use it to transform the coloring c_1 of H_1; each color is replaced by its image under the mapping. By the consistency condition, the resulting coloring c_1' assigns the same color to each common node as c_0 does. We can therefore combine the colorings c_1' and c_0 to obtain a coloring c of the entire graph G.

What properties must the one-to-one mapping satisfy in order that the coloring of G be a *minimum* coloring? We must ensure that the transformed coloring c_1' *re-uses* colors used by c_0 as much as possible. Let x_i be the number of colors used by c_i, for $i = 0, 1$. If $x_1 \leq x_0$, we can map every color used by c_1 to a color used by c_0 (while satisfying the consistency condition). In this case, a total of x_0 colors are used by the merged coloring c of G. If $x_1 > x_0$, we must map $x_1 - x_0$ of the colors used by c_1 to colors not used by x_0. In this case, a total of x_1 colors are used. Let us assume inductively that c_0 and c_1 are *minimum* colorings of $\widehat{H_0}$ and $\widehat{H_1}$, respectively. Then since these graphs are subgraphs of G, the minimum number of colors required for G is at least $\max(x_0, x_1)$. The merged coloring c of G achieves this lower bound; hence it is minimum.

5.6.2 A divide-and-conquer approach to coloring

To apply the color-permuting strategy to minimally coloring a chordal graph G, we use divide-and-conquer. For now, we outline the basic approach; we will later consider each step in more detail. Divide the graph G into node-disjoint subgraphs $\widehat{H_0}, H_1, \ldots, H_s$ such that the neighborhood of each H_i for $i = 1, \ldots, s$ is a clique K_i contained in $\widehat{H_0}$. Next, for $i = 1, \ldots, s$, let $\widehat{H_i} = G[H_i \cup K_i]$. The resulting subgraphs $\widehat{H_0}, \widehat{H_1}, \ldots, \widehat{H_s}$, illustrated in Figure 18, satisfy the conditions that make possible the color-mapping strategy.

1. The intersection of each $\widehat{H_i}$ $(i > 0)$ with $\widehat{H_0}$ is a clique.
2. The union of $\widehat{H_0}, \ldots, \widehat{H_s}$ is the whole graph G.

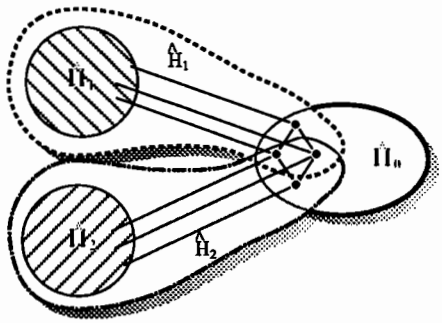

Figure 18: Divide G into disjoint subgraphs $\widehat{H_0}, H_1, H_2, \ldots, H_k$ (here $k = 2$) such that the neighborhood of each H_i ($i > 0$) is a clique contained in $\widehat{H_0}$.

In parallel, recursively find minimum colorings $c_0, c_1, \ldots, c_k$ for the subgraphs $\widehat{H_0}, \widehat{H_1}, \ldots, \widehat{H_s}$. For $i = 1, \ldots, s$ in parallel, find a transformed coloring c_i' of $\widehat{H_i}$. Derive the coloring of G from the colorings $c_0, c_1', \ldots, c_k'$.

The questions that we need to address in order to turn this outline into an algorithm are:

1. How do we choose the decomposition of G into disjoint subgraphs?
2. Given that the subgraphs on which we recurse are not disjoint, how can we achieve a linear processor bound?
3. How do we guarantee small recursion depth?
4. How do we carry out the repair step, in which the colors are mapped?

We first address question 2. It seems that since the subgraphs $\widehat{H_0}, \widehat{H_1}, \ldots, \widehat{H_s}$ on which the algorithm recurses are not disjoint—$\widehat{H_i}$ shares with $\widehat{H_0}$ the edges and nodes in the cliques K_i—we cannot hope to make do with only one processor per edge and one processor per node.

To cope with this difficulty, we use a simple idea called *clique management*. Given the knowledge that K_i is a clique, the algorithm need not inspect the edges between nodes of K_i during the recursive call on $\widehat{H_i} = G[H_i \cup K]$; it "knows" without checking that every pair of nodes in K_i are adjacent. Hence for this recursive call, we do not need to assign processors to the edges of K_i, only to the "relevant edges"—those within H_i and those between H_i and K_i. For distinct subgraphs H_i and H_j, the relevant edge-sets are disjoint. Hence we can carry out the recursive calls while holding to our rule of assigning one processor per edge of the original graph.

The above idea helps us deal with edge duplications—how do we handle node duplications? Because every node of K_i is a neighbor of a node of H_i, every node of K_i is the endpoint of an edge relevant to H_i. We can have each processor assigned to a relevant edge do double-duty—it handles the edge and also handles the endpoint in the clique.

COLOR(G, K_0):
Input: Connected graph G containing a clique K_0, such that every node of K_0 has a neighbor in $G - K_0$.
Output: Optimal coloring of G.

C1 If $G - K_0$ consists of a single node v, then G is a clique; assign the first $|V(G)|$ colors to its nodes, and end.

C2 Otherwise, Break $G - K_0$ into subgraphs $H_0, \ldots, H_s$ such that

- each subgraph has size at most half that of $G - K_0$;
- $H_1, \ldots, H_s$ are distinct components of $G - K_0 - H_0$; and
- for $1 \leq i \leq s$, the neighborhood of H_i in $G - H_i$ is a clique K_i.

C3 For $i = 0, \ldots, s$ in parallel,
call COLOR$(\widehat{H}_i, K_i)$, where $\widehat{H}_i = G[H_i \cup K_i]$, to get an optimal coloring c_i of $\widehat{H}_i$.

C4 For $i = 1, \ldots, s$ in parallel, modify the coloring c_i to be consistent with c_0 on the nodes $V(K_i)$ they have in common.

C5 Merge the colorings to obtain a coloring of G.

Figure 19: The recursive algorithm COLOR for finding an optimal coloring of a chordal graph G.

According to the idea of clique management, the recursive coloring procedure takes *two* inputs, the graph G to be colored and a clique K_0 contained in G. A sketch of the procedure COLOR(G, K_0) is given in Figure 19. Step C1 handles the base case of the recursion, where the graph G is a clique. In this case, it is easy to minimally color G—each node must receive a different color. Otherwise, step C2 finds a decomposition of $G - K_0$ into some number of subgraphs $H_0, \ldots, H_s$ such that for $i = 1, \ldots, s$, the neighborhood of H_i is a clique K_i. (Note that this clique K_i may share nodes with the clique K_0.) Step C3 recursively colors each of the subgraphs $\widehat{H}_i$; during the coloring of $\widehat{H}_i$, the clique K_i is managed.

In order to ensure that the recursion depth is small (question 4), the decomposition of $G - K_0$ is such that each subgraph H_i is at most half the size of $G - K_0$. Thus we measure progress not in terms of the size of the graph G being colored, but in terms of the number of nodes of the graph that are not in the clique being managed. On input G, K_0, the number of levels of recursion is $1 + \log |G - K_0|$. We shall show that each level can be implemented in $O(\log t)$ time using t processors, where t is the number of edges with at least one endpoint in $G - K_0$. To find a coloring in the original graph G, call COLOR$(G, \emptyset)$.

At last we address question 1: how is the decomposition chosen? This is where we use the special properties of the elimininination tree; in fact, this is the only place we use chordality. Inductively we assume we have an elimination tree T for G in which the nodes of K_0 are the highest nodes. Using the Euler-tour technique [45], choose the lowest node $\hat{v}$ in T that has more than $p/2$ descendents, where $p = |G - K_0|$. Let $v_1, \ldots, v_s$ be the children of $\hat{v}$ in T; then

each subtree T_{v_i} has at most $p/2$ nodes. We let K be the clique $\{\hat{v}\} \cup \{\text{higher neighbors of } \hat{v}\}$, and let $H_i = G[T_{v_i}]$. By Lemma 5.4, the subgraphs $H_1, \ldots, H_s$ are connected components of $G - K$. The neighborhood of each H_i in $G - H_i$ is contained in the clique K, and is therefore itself a clique K_i. Let $H_0 = G - K_0 - \bigcup_{i=1}^{s} H_i$. By choice of $\hat{v}$, H_0 has at most $p/2$ nodes.

Step C4, in which we modify the colorings to be consistent, can be implemented using parallel prefix computation. We shall next describe this step in greater detail.

5.6.3 The repair step

It is convenient to use the positive integers as our colors. The repair step consists of finding a mapping ϕ_i from the colors used by c_i (for $i = 1, \ldots, s$) into the set of all colors, and computing the transformed coloring $c_i'(v) = \phi_i(c_i(v))$, such that the following two conditions are satisfied:

> *Consistency condition*: For each $1 \leq i \leq s$, the color assigned by c_i to a common node $v \in \widehat{H_i} \cap \widehat{H_0}$ is mapped to the color assigned to the same node by c_0.
>
> *Parsimony condition*: The maximum color used by the transformed colorings $c_1', \ldots, c_s'$ is no more than the maximum color used by the original recursive colorings $c_0, c_1, \ldots, c_s$.

The consistency condition ensures that each transformed coloring c_i' can be merged with c_0. The parsimony condition ensures that the resulting merged coloring is minimum (if the recursive colorings were minimum.)

To make the repair step easier, we inductively require that for $i = 0, \ldots, s$, the coloring c_i assigns colors 1 through $|K_i|$ to the nodes of the associated clique K_i. This requirement is easy to satisfy at the base of the recursion, step C1. It is automatically preserved in going from one level of recursion to the next higher level: the colors assigned by c to the nodes of K_0 are exactly those assigned by c_0.

Given the above requirement is satisfied, we now show how to transform a coloring c_i ($1 \leq i \leq s$) so that the consistency condition and the parsimony condition are satisfied. Let x_i be the number of colors used by c_i. By the requirement, the colors 1 through $|K_i|$ are assigned by c_i to the nodes of K_i; these are the colors that must be mapped to corresponding colors used by c_0 in order that the consistency condition be satisfied. We therefore partially define the mapping ϕ_i by

$$\forall v \in K_i \ : \ c_i(v) \longmapsto c_0(v) \tag{4}$$

The other colors used by c_i, colors $|K_i| + 1$ through x_i, are assigned to the nodes of H_i. We must map these to other available colors used by c_0 (and use additional colors only when necessary). In order to help us identify the other available colors, we perform a preliminary

computation. For each color q between $|K_i|+1$ and x_i, we count the colors less than q that c_0 uses in coloring K_i: let

$$A_i[q] := |\{c_0(v) < q \ : \ v \in V(K_i)\}|.$$

(The values $A_i[\cdot]$ can be computed using a parallel prefix computation.) A color q between $|K_i|+1$ and x_i is the $q-|K_i|^{\underline{th}}$ color used by c_i on H_i. We want to map it to the $q-|K_i|^{\underline{th}}$ color not already assigned by c_0 to a node of K_i. We therefore extend the definition (4) of the mapping ϕ_i as follows:

$$\forall q(|K_i|+1 \le q \le x_i): \quad q \longmapsto q - |K_i| + A_i[q] \tag{5}$$

Thus the colors of nodes of H_i are mapped to colors starting at 1, with gaps only for colors already assigned to nodes of K_i. We use this mapping to transform the coloring c_i to obtain c_i': for each node $v \in \widehat{H_i}$, define

$$c_i'(v) := \begin{cases} c_0(v) & \text{if } v \in K_i \\ c_i(v) - |K_i| + A_i[c_i(v)] & \text{if } v \in H_i \end{cases}$$

This assignment ensures that, for any node $v \in H_i$, colors 1 through $c_i'(v)$ all appear in the coloring c_i' of $\widehat{H_i}$. As a consequence, we claim, the parsimony condition is satisfied. If the maximum color q used by the transformed coloring c_i' is used for a node of K_i, then q is at most the maximum color used by c_0. Otherwise, colors 1 through q all appear in c_i'. Since c_i' uses the same number of colors as c_i, it follows that c_i uses at least q colors. The claim is proved.

5.6.4 Resource requirements

The only nontrivial computation in implementing step C4 is computing the $A_i[q]$ values. For each $1 \le i \le s$, construct an array of length $x_i \le |H_i|$, and initialize all entries to 1. Then set to zero the $q^{\underline{th}}$ entry for each color q that c_0 assigns to a node of K_i. Finally, perform a parallel prefix sum computation on the array to compute the values $A_i[q]$. The total work is proportional to $\sum_i |K_i| + |H_i|$.

Let t_i be the number of edges that either lie in H_i or connect H_i to K_i. Since H_i is connected, the number of nodes in H_i is at most one more than the number of edges in H_i. Since every node in K_i is a neighbor of some node in H_i, the number of nodes in K_i is at most the number of edges connecting H_i to K_i. Thus $|H_i \cup K_i| \le t_i + 1$. so the total work is proportional to $\sum_i t_i$, which is just the number t of edges that either lie within $G - K_0$ or connect $G - K_0$ to G.

Assume inductively that $O(\log t_i \log |H_i|)$ time and $O(t_i/\log t_i)$ processors are sufficient to recursively color $G[H_i \cup K_i]$. Then $O(t/\log t)$ processors are sufficient to recursively color all the subgraphs $G[H_i \cup K_i]$ in $O(\log t(\log(|G-K_0|)-1))$ time and to combine the colorings in $O(\log t)$ time, for a total of $O(\log t \log |G-K_0|)$ time.

6 The Framework for finding a Perfect Elimination Ordering

6.1 Preliminaries

In this section, we develop two efficient parallel algorithms for finding a perfect elimination ordering, one sequential and one parallel algorithm. The optimization algorithms of the previous section all assume that a perfect elimination ordering (or peo) is part of the input; hence the peo algorithms are crucial for finding a minimum coloring, a maximum independent set, etc., efficiently.

Recall the historically first peo algorithm, that of Fulkerson and Gross: repeatedly find a *simplicial* node (one whose neighbors form a clique), and delete it. This algorithm clearly consists of only n stages for an n-node graph; the disadvantage is that each stage seems fairly expensive. Indeed, even checking whether a given node is simplicial seems to require $\Omega(m)$ time in the worst case.

The theorem of Dirac guaranteeing the existence of a simplicial node in every chordal graph gives us a clue as to how to proceed in order to get an improved sequential algorithm. Dirac's theorem in fact ensures that every graph with at least two nodes has at least *two* simplicial nodes. For any node v, therefore, there is a simplicial node other than v; once it is eliminated, there is remaining another simplicial node other than v, and so on, until there is only v remaining. We see that we can initially set aside v to be eliminated last. This idea suggests we might be able to build a *peo* backwards, first choosing the last node of the peo, then choosing the second-to-last, and so on. The linear-time peo algorithm of Rose, Tarjan, and Lueker does exactly this.

Dirac's theorem ensures that the choice of last node is arbitrary. Once the last node of a *peo* is chosen, however, the choice of the second-to-last node is far from arbitrary, the subsequent choice of the third-to-last node is even more restricted, and so on. We need a condition that guides us in making these choices so that we never make a choice that rules out all peo's.

We shall give a characterization of those partial solutions that can be extended to complete solutions. In order that our characterization yields not only a sequential algorithm but also a parallel algorithm, we use a general model of what constitutes a partial solution. A *semiorder* is a partition of a nodeset into an ordered sequence of classes: $\Psi = (C_1, C_2, \ldots, C_k)$. For example, a total ordering is a semiorder $(\{v_1\}, \{v_2\}, \ldots, \{v_n\})$ in which each class contains only one node. The partial solution in which we have chosen the last node v but have made no other decisions is represented by a semiorder $(C, \{v\})$ consisting of two classes: the second class contains v and the first class contains all other nodes. Figure 20 illustrates some examples of semiorders.

We make progress towards a complete solution by *refining* the semiorder, i.e. further

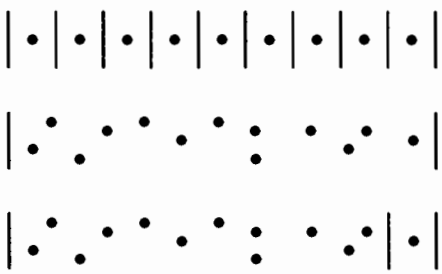

Figure 20: Example semiorders are depicted. The first is a semiorder in which every node is in a separate class, i.e. a total order. The second consists of only a single class; it is effectively a completely unordered set. The third is a semiorder in which the last node has been chosen but no other ordering decisions have been made.

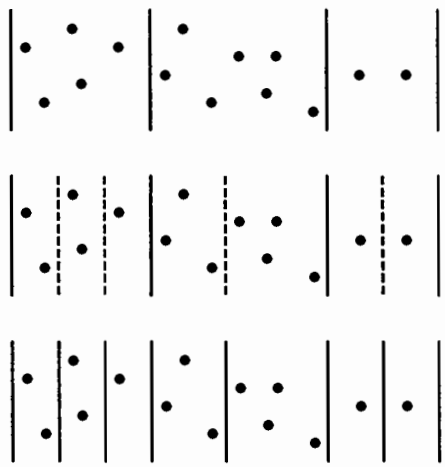

Figure 21: This figure illustrates the process of refining a semiorder. We start with a semiorder with three classes. Each class is divided into subclasses. Finally, these subclasses become the classes of the new semiorder.

dividing its classes into subclasses. Let $\Psi = (C_1, \ldots, C_k)$ be a semiorder on the nodeset V, and let $\Phi = (C_{i1}, C_{i2}, \ldots, C_{ir})$ be a semiorder on a class C_i of Ψ. To *refine* Ψ *by* Φ is to replace the class C_i in Ψ by the subclasses $C_{i1}, C_{i2}, \ldots, C_{ir}$ in order, yielding a new, *refined* semiorder. More generally, to *refine* a semiorder is refine some of its classes, as illustrated in Figure 21.

We say one semiorder is *consistent* with a second if the second can be obtained by refining the first. For example, the semiorder with only one class is consistent with every semiorder, while a total order—a semiorder in which each class contains one node—is consistent only with itself.

For a graph G and a semiorder Ψ, we write G_Ψ for the graph G with each node labelled by the class containing it. For two nodes v and w, we write $\Psi(v) > \Psi(w)$ if v is higher than w. For example, a semiorder Ψ is consistent with another semiorder Φ if $\Psi(v) > \Psi(w)$

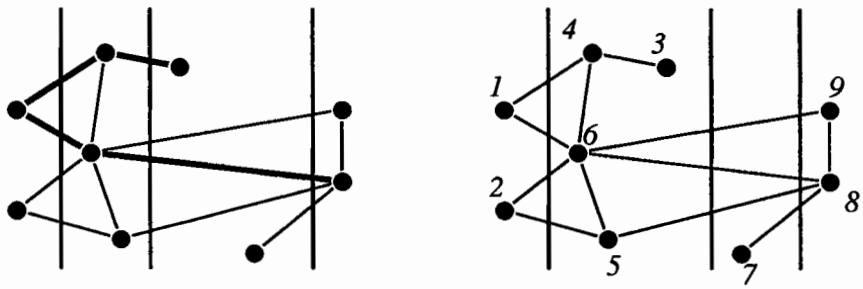

Figure 22: The first semiorder is not valid for the graph shown; the highlighted end-high path, for example, does not have adjacent endpoints. The second semiorder is valid, and the nodes are numbered in perfect elimination order.

implies $\Phi(v) > \Phi(w)$ for all pairs of nodes v, w.

Our notion of an end-high path, defined in Subsection 5.1 for graphs whose nodes are totally ordered, is equally applicable to graphs G_Ψ whose nodes are semiordered: an end-high path is a path in the graph whose endpoints are strictly higher than its internal nodes. We say a semiorder Ψ is *valid* for a graph G if every end-high path in G_Ψ has adjacent endpoints. The reason for this definition is made clear by the following theorem.

Theorem 6.1 (End-high Theorem) *A semiorder on the nodes of a chordal graph is consistent with some* peo *if and only if the semiorder is valid for the graph.*

Figure 22 gives examples of how this theorem is applied. We postpone the proof of the theorem until the end of this section; for now, we consider the special case in which the semiorder is restricted to be a total order.

Lemma 6.2 *A total ordering of the nodes of a chordal graph is a* peo *if and only if it is valid.*

Proof: It follows from Lemma 5.2 of Section 5.1 that a peo is valid. Suppose conversely that the total ordering Ψ is valid, and let v be a node. We must show that the higher neighbors of v form a clique. Let x and y be any two higher neighbors of v. Then xvy is an end-high path, so by validity, its endpoints x and y are adjacent. ∎

Terminology begets terminology. In order to prove that every end-high path has adjacent endpoints, it is easier to focus on a special kind of path, the *chordless* path. A *chord* of a path $P = v_1 \ldots v_k$ in a graph G is an edge $\{v_i, v_j\}$ in G, where $i < j - 1$. We say a path is *chordless* if it has no chord. The following simple observation shows why we can focus exclusively on chordless paths.

Lemma 6.3 (Shortcutting Lemma) *For any path P in a graph G, there is a chordless path P' with the same endpoints, such that $V(P') \subseteq V(P)$.*

ITERATED REFINEMENT

R1 To initialize, let Ψ be the trivial semiorder consisting of one class.

R2 While Ψ is not a total ordering.

R3 Refine Ψ, maintaining validity

R4 Output the total ordering Ψ.

Figure 23: The ITERATED REFINEMENT algorithm for finding a peo.

Proof: If $P = v_1 \ldots v_k$ has a chord $\{v_i, v_j\}$, replace P with the path $v_1 \ldots v_i v_j \ldots v_k$, which has fewer chords; by iterating, we obtain the desired chordless path. ∎

For a graph G with a semiorder on its nodes, we say a path is *weakly end-high* if its endpoints are at least as high as all internal nodes. Contrast this notion with that of a (strictly) end-high path, whose endpoints must be strictly higher than the internal nodes. We say a path is *uniform* if all its nodes except possibly one endpoint belong to the same class.

Lemma 6.4 (Uniformity Lemma) *If Ψ is valid for G, each chordless weakly end-high path P in G_Ψ is uniform.*

Proof: Suppose $P = v_1 \ldots v_k$ is not uniform, and let $v_i \ldots v_j$ be a maximal subpath consisting of nodes lower than the endpoints of P. Then $v_{i-1} \ldots v_{j+1}$ is an end-high path, so by validity of Ψ, $\{v_{i-1}, v_{j-1}\}$ is an edge, contradicting the chordlessness of P. ∎

6.2 Iterated Refinement

We shall next lay out our basic framework for peo algorithms. In the following section, we describe the sequential algorithm of Rose, Tarjan, and Lueker, and show how it can be cast within the framework. In Section 8, we give the parallel algorithm of [29].

Our general framework for finding a peo, which we call *iterated refinement*, is illustrated in Figure 23. We assume at the outset that the input graph is chordal, and show how to construct a total ordering that is a peo if the assumption holds; once the total ordering has been constructed, the method of Subsection 5.2 can be used to verify that it is indeed a peo and hence that the input graph is indeed chordal.

The iterated refinement approach starts with the trivial semiorder, in which all nodes are in the same class, and iteratively refines the semiorder, maintaining validity throughout the process, until the semiorder becomes a total order. The initial, trivial semiorder is trivially

valid. Since we maintain validity throughout, the final, total ordering is valid. But by Lemma 6.2, a valid total ordering is a peo.

The iterated refinement approach is the basis for proving the End-high Theorem.

Proof of the End-high Theorem: The "only if" direction is easy to prove; the proof resembles that of Lemma 5.2. Suppose Ψ is a semiorder on G that is consistent with the peo σ. We need to show that every chordless end-high path P in G_Ψ has adjacent endpoints. If P consists of a single edge $\{x, y\}$, we are done, so assume for a contradiction that P has internal nodes. Let u be the internal node with the minimum σ-number. Then the neighbors of u in P have higher σ-number, so they are adjacent by definition of a perfect ordering, contradicting the chordlessness of P.

To prove the "if" direction, suppose G is chordal. In Subsection 8, we give a procedure REFINE to validly refine any valid semiorder on G that is not a total order. By repeated refinements, we eventually obtain a valid total order, i.e. a peo. ∎

Even aside from the fact that we have not yet given the procedure REFINE or proved its correctness, the proof of the "if" direction may seem to be a cheat. How can we use an algorithm to prove a theorem "about" the algorithm? The answer is that the End-high theorem is not used in proving the correctness of either the *iterated refinement* approach or the REFINE procedure; only the special case of Lemma 6.2 is needed.

The use of an algorithm to prove a combinatorial result is by no means new; illustrious examples include use of a maximum flow algorithm to prove the Max-flow Min-cut Theorem, and use of Edmonds' blossom-shrinking nonbipartite matching algorithm to prove the Gallai-Edmonds Structure Theorem. (For those readers still not satisfied, reference [28] contains a more direct proof of the End-high Theorem.)

A word about implementation is in order. For conceptual simplicity, we use the notion of semiorder refinement. For purposes of implementation, a semiorder must be represented in a way that can be efficiently updated when refinement takes place. In sequential computation, one can use doubly-linked lists, one for each class listing the elements of the class. This representation makes it easy to remove nodes from a class, one at a time, and form new subclasses. This is the representation used in implementing Rose, Tarjan, and Lueker's linear-time sequential algorithm for finding a peo, described in the next section.

In parallel computation, the same representation can be used, but removing nodes from a class is somewhat more involved, because a large collection of nodes may need to be removed all at the same time. This can be accomplished by use of list-ranking techniques. An alternative representation assigns distinct positive integers to each of the classes, in ascending order. Instead of assigning, say, 1 to the first class, 2 to the second, and so on, we can assign $1n$ to the first class, $2n$ to the second, and so on. This gives us enough room for refinement: we can split each class into subclasses and assign each subclass its own integer. After each class has been refined, we can use sorting to bring together all nodes in the same class, and then use a parallel prefix computation to renumber the classes in preparation for

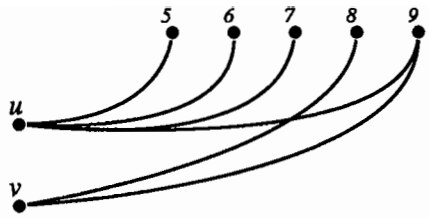

Figure 24: Both u and v are adjacent to 9, but only v is adjacent to 8. Therefore, the neighborhood of node v among the numbered nodes is lexicographically higher than that of u.

the next refinement. For brevity, we will not address these issues of representation when we discuss the parallel algorithm for finding a peo.

7 Rose, Tarjan, and Lueker's sequential algorithm for finding a perfect elimination ordering

The sequential peo algorithm of Rose, Tarjan, and Lueker used a method they called *lexicographic breadth-first search.* The idea is to construct the peo backwards, selecting each node in turn in an order dependent on the node's neighbors among previously selected nodes. The algorithm consists of n stages, one for each node, numbered from n down to 1. In stage i, a node is selected and assigned position number i in the peo being built. The selection is made from the set of as-yet-unnumbered nodes, based on the neighborhoods of these nodes among the numbered nodes. The node selected is one which has the *lexicographically highest* neighborhood. That is, given two distinct subsets S_1 and S_2 of the numbered nodes, let v_k be the highest-numbered node that is in one subset but not in the other; then the subset containing v_k is *lexicographically higher* than the subset not containing v_k. Figure 24 gives an example of this rule being applied.

7.0.1 Linear-time implementation

Rose, Tarjan, and Lueker show that this version of lexicographic breadth-first search can be implemented in linear time. The key is to keep all the unnumbered nodes partitioned into classes according to their neighborhoods among the numbered nodes. That is, each class contains all the unnumbered nodes having a particular neighborhood, and the classes are ordered lexicographically by neighborhood. Each class is represented by a doubly-linked list.

Stage i consists of two phases. First, the algorithm selects an unnumbered node v belonging to the highest class, removes it from the linked list representing that class, and assigns it number i. Second, each class C is partitioned into two subclasses, a higher subclass C^+ and a lower subclass C^-. The higher subclass consists of the nodes in the class that are adjacent

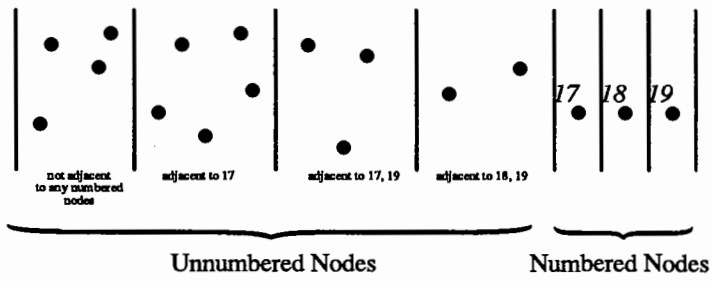

Figure 25: The semiorder consists of a sequence of classes of unnumbered nodes, followed by singleton classes, one for each numbered node. Each of the former classes contains all nodes with a specific neighborhood among the numbered nodes, and these classes are ordered according to the lexicographic ordering of the neighborhoods. The classes containing numbered nodes are ordered according to number.

to the newly numbered node v, and the lower subclass consists of the remaining nodes in the class. The ordered sequence of subclasses is the sequence of classes used for the next stage. That is, if at the beginning of stage i, the sequence of classes was $C_1, C_2, \ldots, C_k$, then at the end the new sequence of classes is $C_1^-, C_1^+, C_2^-, C_2^+, \ldots, C_k^-, C_k^+$. (In general, some of these new classes may be empty; the empty classes are omitted from the sequence.)

In order to carry out the partitioning of the classes, the algorithm examines the adjacency list of the newly numbered node v to determine the set of unnumbered nodes having v as a neighbor. For each such unnumbered node w, the algorithm determines which class C currently contains w, removes w from the linked list representing C, and adds w to a new list representing the higher subclass C^+. Subsequently the elements not removed from C form the lower subclass C^-. Thus, work is only performed for the neighbors of w.

Consequently, the stage in which v is assigned number i requires constant time to remove v from its class, plus time proportional to the number of edges connecting v to unnumbered nodes. Over all n stages, each node is examined once and each edge is examined once, so the total time required is $O(n + m)$.

7.0.2 Correctness

In order to prove that lexicographic breadth-first search yields a peo when the input graph is chordal, we cast the algorithm in the framework of Iterated Refinement, and show that each refinement maintains validity. At each stage in the algorithm, the semiorder consists of a sequence of classes containing the unnumbered nodes, followed by a sequence of single-node classes, one for each numbered node, in increasing order of number.

Stage i consists of two refinements, one for each phase. We must verify that these two refinements maintain validity. We inductively assume at the beginning of the stage that the current semiorder Ψ_i divides the unnumbered nodes into classes according to their neighbor-

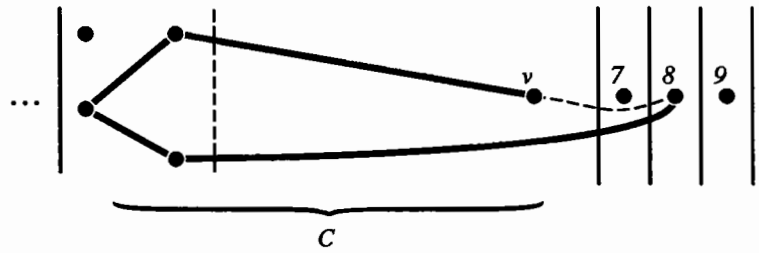

Figure 26: This figure shows the highest class C and the numbered nodes. The end-high path P has endpoints v, the node currently being numbered, and 8, a previously numbered node. Since all nodes of C have the same neighborhood among the previously numbered nodes, v must be adjacent to 8.

hoods among the numbered nodes. Thus all nodes in a class have the same neighborhood. We also assume that the current semiorder Ψ is valid. In the first refinement, we refine only the highest class C containing unnumbered nodes. We select a node v in C, and we partition C into a higher class containing only the node v—this corresponds to assigning the number i to v—and a lower class containing all the other nodes of C. Let Ψ' be the resulting refined semiorder.

Claim 7.1 *The first refinement $\Psi \to \Psi'$ maintains validity.*

Proof: Suppose Ψ is valid, and let P be a chordless end-high path in $G_{\Psi'}$ with endpoints x and y; we must show that x and y are adjacent. If neither of the endpoints is v, then P was an end-high path even before the refinement, in G_Ψ, so x and y are adjacent by the validity of Ψ. Assume therefore that the lower of the endpoints, say x, is v; the other endpoint y must be a previously numbered node.

By the uniformity lemma, the path P is uniform with respect to the old semiorder Ψ. Hence every node of P, except for the higher endpoint y, belonged to the same class C of Ψ as v. We assumed that all the nodes in each class of Ψ have the same numbered neighbors. In particular, the node of P adjacent to y has the same numbered neighbors as v. We have proved that v is adjacent to y. ∎

In the second refinement, we consider each class C containing unnumbered nodes, and we partition its nodes into a higher class C^+ and a lower class C^-, where the higher class contains the nodes of C adjacent to v. Let Ψ'' be the resulting refined semiorder.

Claim 7.2 *The refinement $\Psi' \to \Psi''$ maintains validity.*

Proof: Suppose Ψ' is valid, and let P be a chordless end-high path in $G_{\Psi''}$ that was not end-high with respect to the previous semiorder Ψ'. By the uniformity lemma, all nodes of P except possibly the higher endpoint y, belonged to a single class C of Ψ'. There are four

possibilities for y: it may be a node numbered in a previous iteration, the newly numbered node v, a node of C^+, or a node that belonged to a higher class C' than C in Ψ.

First suppose y is a previously numbered node. We assumed that all the nodes in each class of Ψ' have the same previously numbered neighbors; since the node of P adjacent to y belongs to the class C of Ψ', every node of C, including x, is adjacent to y.

Next suppose y is the newly numbered node v. The other endpoint x belonged to the class C of Ψ', along with the internal nodes of P. Since P is end-high with respect to Ψ'', the endpoint x must belong to the higher subclass C^+. Hence x is adjacent to y.[4]

Now suppose y is adjacent to v; this includes the case in which y is in C^+. By adding the edges $\{x,v\}$ and $\{y,v\}$ to the path P, we obtain a cycle θ. If θ has three edges then $\{x,y\}$ is the third, and we are done; otherwise by chordality θ has a chord. Since P is chordless, θ's chord cannot consist only of nodes of P; it must contain the node v. But the internal nodes of P belong to the lower subclass C^-, and hence are not adjacent to v, so we have a contradiction.

Suppose finally that y belongs to a higher class C' of Ψ' than C, and y is not adjacent to v. Then y has a numbered neighbor v_j that is not a neighbor of any node of C. By adding the edges $\{y,v_j\}$ and $\{x,v\}$ to P, we get a path P' that is end-high with respect to Ψ'. By Claim 7.1, its endpoints v and v_j are adjacent. By adding the edge $\{v,v_j\}$ to P', we obtain a cycle θ, which must have a chord. But v is not adjacent to any node of P except x, and v_j is not adjacent to any node of P except y, so the only possible chord is $\{x,y\}$. ∎

We have proved that the two refinements of stage i, $\Psi \rightarrow \Psi'$ and $\Psi' \rightarrow \Psi''$, maintain validity. The algorithm uses the semiorder Ψ'' for the next stage; hence the algorithm maintains validity throughout, and is therefore correct.

8 A parallel algorithm for finding a perfect elimination ordering

In this section, we show how to find a peo in $O(\log^2 n)$ time using $n+m$ processors.

8.1 The REFINE procedure

In order for an *iterated refinement* algorithm to be a fast parallel algorithm, the number of iterations required must be small, much smaller than the $2n-1$ iterations required by the sequential algorithm of the previous section. We give an algorithm requiring only $O(\log n)$ iterations. The core of the algorithm is a subroutine REFINE which takes as input a graph G

[4]In fact, this case cannot happen; the node of P adjacent to y would be in C^+, which would mean P was not an end-high path after all.

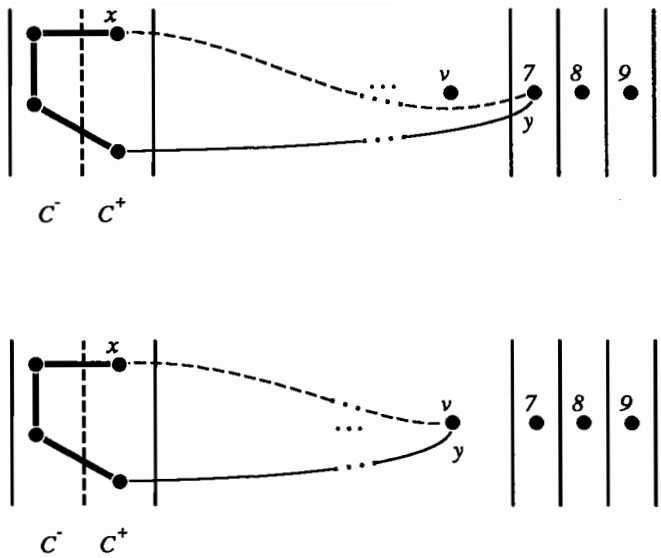

Figure 27: In the first semiorder, y is a previously numbered node; all the nodes in C have the same numbered neighbors, so x is adjacent to y. In the second semiorder, y is the newly numbered node v; since x is in C^+, x is adjacent to v. In the third semiorder, both endpoints are adjacent to v, but none of the internal nodes of the path is. In this case, chordality implies that x and y are adjacent. In the fourth semiorder, x is adjacent to v, and y is not. The node y is adjacent to a numbered node that is adjacent to v but not to x or any internal nodes of the path. Moreover, none of the internal nodes is adjacent to v. Chordality again implies that x and y are adjacent.

and a valid semiorder Ψ and returns a refined valid semiorder Ψ' whose largest nonsingleton class is four-fifths the size of that of Ψ. After t iterations, the largest nonsingleton class has size at most $(4/5)^t n$. Hence only $O(\log n)$ iterations are required before every class is singleton.

To achieve the refinement, the REFINE subroutine operates independently on each of the nonsingleton classes of the input semiorder Ψ, and refines each. *A priori*, we have no reason to believe that such a strategy could work—that we could refine each class independently and expect to thereby obtain a valid semiorder; perhaps some coordination between different classes is necessary. The key to proving that the strategy can be made to work is the Refinement Lemma, which we give presently. It states, intuitively, that a refinement of a class C preserves validity for the entire graph if it preserves validity for the subgraph consisting of C and its higher neighbors. To state the lemma formally, we need to introduce some notation.

Fix a graph G and a valid semiorder Ψ. For a class C, let $\Gamma(C)$ denote the set of higher neighbors of C, let $\widehat{C}$ denote the subgraph of G induced by $C \cup \Gamma(C)$, and let Ψ_C denote the semiorder $(C, \Gamma(C))$ for $\widehat{C}$.

Lemma 8.1 (Refinement Lemma) *Suppose Ψ is a valid semiorder for G. Let Φ be a semiorder for a class C of G_Ψ. Then the following two conditions are equivalent*

(1) the refinement of G_Ψ by Φ is valid

(2) the refinement of $\widehat{C}_{\Psi_C}$ by Φ is valid.

Proof: Let Ψ' be the refinement of G_Ψ by Φ, and let λ be the refinement of $\widehat{C}_{\Psi_C}$ by Φ. We need to show that Ψ' is valid for G if and only if λ is valid for $\widehat{C}$. It suffices to show that the chordless end-high paths in $G_{\Psi'}$ that are not end-high with respect to Ψ are exactly the chordless end-high paths in $\widehat{C}_\lambda$ that are not end-high with respect to Ψ_C.

One direction is easy. Since $\widehat{C}$ is a subgraph of G, a path P in $\widehat{C}$ is also a path in G. Moreover, λ is consistent with the restriction of Ψ' to $\widehat{C}$, so if P is end-high with respect to λ, it is certainly end-high with respect to Ψ'.

Conversely, let P be a chordless end-high path in $G_{\Psi'}$ that was not end-high with respect to Ψ. By the uniformity lemma, P is uniform with respect to Ψ, so its lower endpoint x lies in the same class of Ψ as its internal nodes. Since P is end-high with respect to Ψ', x lies in a higher class of Ψ' than the internal nodes. Since Ψ' is the refinement of Ψ by Φ, it must be that x lies in a higher class of Φ that the internal nodes. Since Φ is only defined for the class C of Ψ, this class must contain x and all the internal nodes.

The other endpoint y of P is a neighbor of a node of C, and hence is either in C itself, or is a higher neighbor of C. In the first case, y is in a higher class of Φ than the internal nodes because P' is strictly end-high. In the second case, y is in a higher class of Ψ_C than the internal nodes. In either case, we conclude that P' is an end-high path in $\widehat{C}_\lambda$. ∎

The Refinement Lemma implies that to validly refine a class C in G_Ψ, we need only validly refine it in $\widehat{C}_{\Psi_C}$. In fact, we observe next that all the classes of G_Ψ may be thus refined simultaneously and independently, as far as validity is concerned. The reason is that for any class C, $\widehat{C}_{\Psi_C}$ remains unchanged when classes other than C are refined. More formally, fix a semiorder Φ on C. By applying the lemma to Ψ, we see that the following two conditions are equivalent:

(1) the refinement of G_Ψ by Φ maintains validity

(2) the refinement of $\widehat{C}_{\Psi_C}$ by Φ maintains validity.

Now suppose Ψ' is obtained from Ψ by refining classes other than C. By applying the lemma to Ψ', we see that condition (2) is equivalent to the following condition:

(3) the refinement of $G_{\Psi'}$ by Φ maintains validity.

We conclude that conditions (1) and (3) are equivalent, and hence that the refinement of C is independent of the refinement of other classes. Thus the REFINE subroutine can refine each class independently and still be assured of maintaining validity for the entire graph.

We make one more useful observation before giving the REFINE procedure. We shall see at the end of this subsection that it is easier to work with *connected* classes, i.e. classes C whose induced subgraphs $G[C]$ are connected. The following corollary shows that we can assume without loss of generality that this condition holds.

Corollary 8.2 *Refining a class by dividing it into its connected components always maintains validity.*

Proof: No end-high path can be created by such a refinement, because an end-high path would have to cross components. More formally, let Ψ be a valid semiorder for G, and let C be a class of Ψ. Let Φ be a semiorder on C consisting of the connected components of $G[C]$ in any order. Suppose we refine $\widehat{C}_{\Psi_C}$ by Φ, obtaining $\widehat{C}_\lambda$. If we can prove that no new end-high paths result, the corollary will follow via the Refinement Lemma. Assume for contradiction that there is an end-high path P in $\widehat{C}_\lambda$ that was not end-high in $\widehat{C}_{\Psi_C}$. Then all the internal nodes and at least one endpoint x must be in C. Since they are connected by a path, these nodes all lie in the same connected component of $G[C]$. In Φ and λ, therefore, x is no higher than the internal nodes, contradicting our assumption that P was end-high. ∎

The procedure REFINE is given in Figure 28. The first step refines Ψ to ensure that each of its classes is connected. The validity of this refinement follows from Corollary 8.2. The main work of REFINE is carried out in the subroutine STRATIFY(G_Ψ, C), which validly refines a connected class C into subclasses such that each nonsingleton connected component of each subclass has size at most four-fifths the size of C. These connected components themselves become classes during the first step of the next iteration. Thus in successive calls to STRATIFY, the maximum class size goes down by a factor of four-fifths.

Procedure REFINE(G_Ψ):

For each nonsingleton class C, call STRATIFY(G_Ψ, C) to refine C.
(Follow-up step) For each resulting class C, refine C by dividing it into its connected components.

Figure 28: The procedure REFINE for refining a semiorder; it reduces the size of each class by a factor of four-fifths.

8.2 Resource Requirements: the return of clique management

There is an apparent obstacle to achieving an efficient algorithm. The second step of REFINE operates independently on each class. To validly refine a class C, one must examine the structure of $\widehat{C}$, the subgraph consisting of C *and* its higher neighbors; these higher neighbors belong to different classes. If we are to refine all classes simultaneously, we must operate on mutually *overlapping* subgraphs; how can we do this with only a linear number of processors?

The solution is *clique management*, the idea we used for minimum coloring in Subsection 5.6.

Lemma 8.3 *Suppose Ψ is a valid semiorder for G, and a class C forms a connected subgraph of G. Then the higher neighbors of C form a clique.*

Proof: For every pair of higher neighbors x and y of C, there is an end-high path through C with endpoints x and y; hence every pair of higher neighbors of C are adjacent. ∎

As in the minimum coloring algorithm, once we have identified a clique, we need not assign processors to the edges between its nodes. Consequently, we can carry out STRATIFY(G_Ψ, C) in work proportional to the number of edges with at least one endpoint in C; we need not do work for the edges between C's higher neighbors. Summing over all classes C, we find that each edge of G is charged at most twice, so the total work is $O(m)$.

The parallel time to carry out STRATIFY(G_Ψ, C) is $O(\log n)$ using t processors, where t is the number of edges with at least one endpoint in C. By applying STRATIFY to all classes in parallel, we achieve $O(\log n)$ time using $2m$ processors. The preliminary step in which we divide each class into its connected components can be carried out within the same bounds, using a standard connectivity algorithm.

8.3 Stratification of a class

We will presently give a procedure for *stratifying* a class: breaking it into subclasses while maintaining validity. Given a validly numbered graph G_Ψ and a nonsingleton connected class C of G_Ψ, our goal is to refine C into subclasses such that each nonsingleton subclass has at most four-fifths the nodes of C.

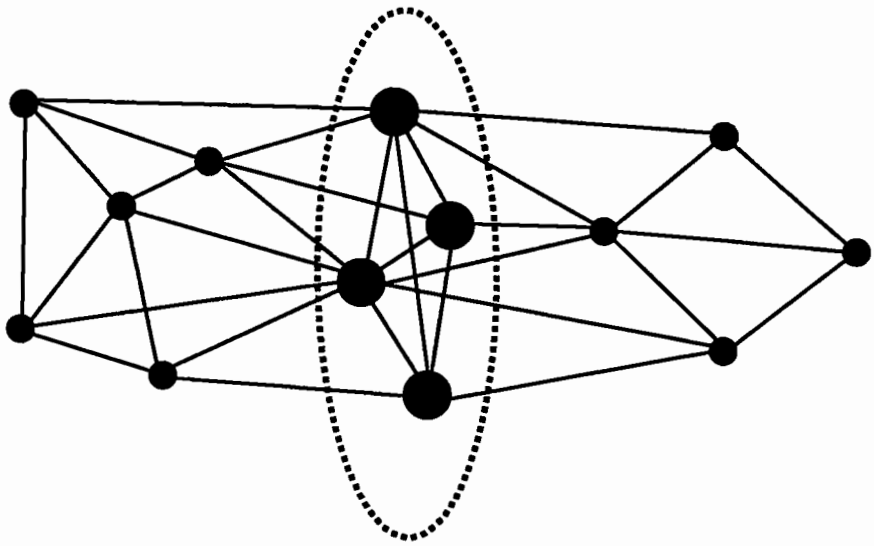

Figure 29: A minimal separator is a clique.

Note that because of the follow-up step of the procedure REFINE, presented in Subsection 8.1, we need only refine C into subclasses such that each *connected component* of each subclass is a constant fraction smaller than C. We call such a refinement a *well-stratification* of C.

Theorem 8.4 *Suppose Ψ is a valid numbering of a chordal graph G, and C is a class of G_Ψ. Valid well-stratification of C can be done in $O(\log k)$ time using $O(k)$ processors of a CRCW P-RAM, where k is the number of edges with at least one endpoint in C. Hence, a* peo *of the chordal graph G can be found in $O(\log^2 n)$ time using $O(m)$ processors.*

In this subsection, we present some results used in proving the correctness of the procedure. In subsequent subsections, we present parts of the procedure.

We start with a lemma of Dirac [15].

Lemma 8.5 (Dirac) *If S is a minimal set of nodes whose removal separates a chordal graph into exactly two connected components, then S is a clique.*

Corollary 8.6 *In a chordal graph, the common neighbors of two nonadjacent nodes form a (possibly empty) clique.*

Proof: In the induced subgraph consisting of the two nonadjacent nodes x and y, and their common neighbors, the common neighbors form a minimal separator between x and y. ▮

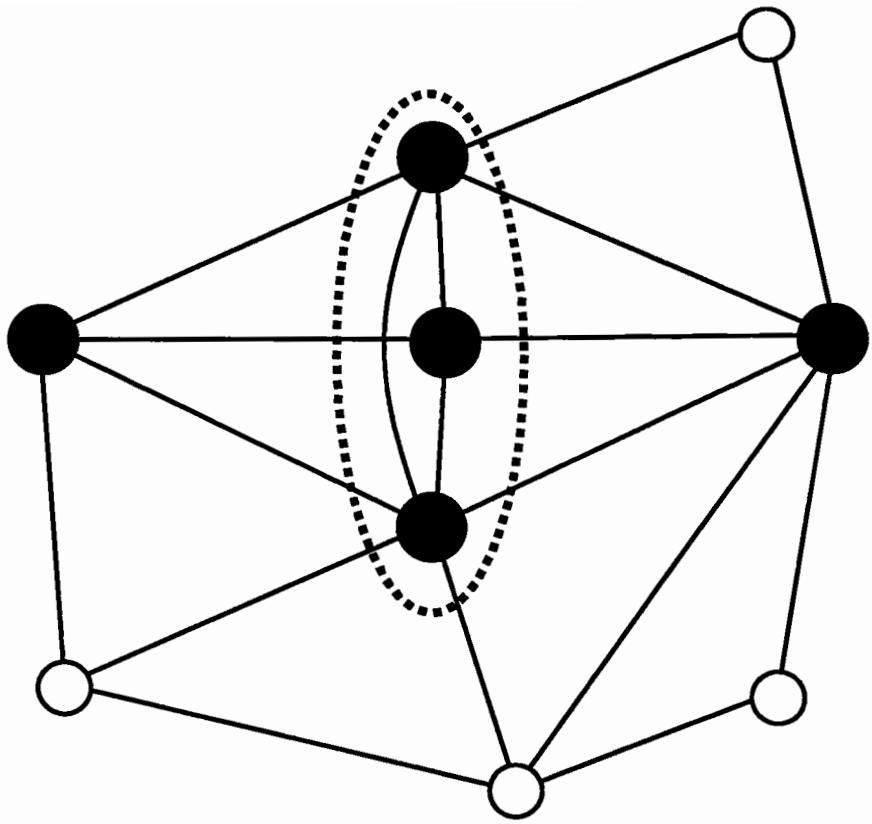

Figure 30: Common neighbors of nonadjacent nodes form a clique.

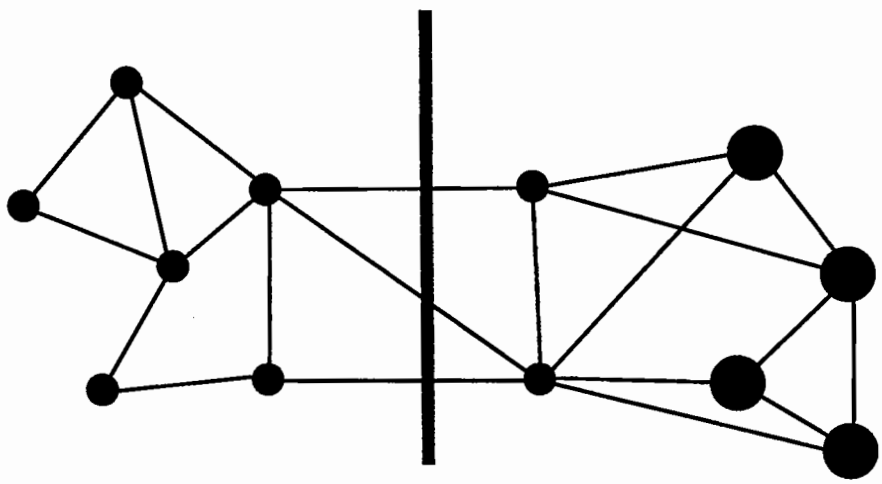

Figure 31: If H is connected, the semiorder $\Phi = (G - H - \{\text{neighbors of } H\}, H \cup \{\text{neighbors of } H\})$ is valid.

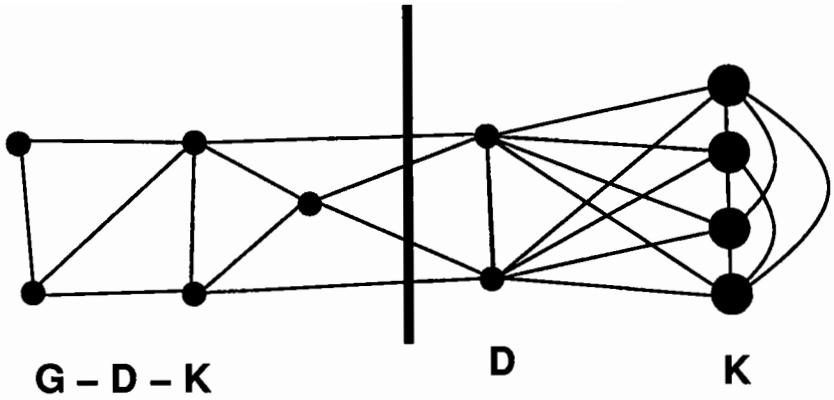

Figure 32: If K is a clique and D contains the nodes adjacent to every node of K, then the semiorder $\Phi = (G - K - D, K \cup D)$ is valid.

Corollary 8.7 *Let H be a connected subgraph of a chordal graph G. Then the semiorder $\Phi = (G - H - \{\text{neighbors of } H\}, H \cup \{\text{neighbors of } H\})$ is valid.*

Proof: Let P be an end-high path in G_Φ, and let K be the component of the lower class that contains the internal nodes of P. Let D be the set of higher neighbors of nodes in K. In the subgraph induced on $K \cup D \cup H$, the nodes of D form a minimal separator between C and H, so D is a clique. The endpoints of P are in D, so they are adjacent. ∎

Lemma 8.8 *Let K be a clique in a chordal graph G. Let $D = \{v \in G : v \text{ adjacent to every node of } K\}$. Then the semiorder $\Phi = (G - K - D, K \cup D)$ is valid.*

Proof: The proof is by induction on $|K|$. The base case, in which $|K| = 1$, follows from Corollary 8.7. Suppose $|K| > 1$, and let v be a node of K. Let A consist of the nodes of $K - \{v\}$ and the nodes adjacent to all of $K - \{v\}$. Let α be the semiorder $(G - A, A)$. By the inductive hypothesis, α is valid. Because K is a clique, v is in A. Next we define a semiorder for A, $\beta = (A - \{v\} - \{\text{neighbors of } v\}, \{v\} \cup \{\text{neighbors of } v\}$. By Corollary 8.7, β is a valid semiorder for $G[A]$. Let G_γ be the refinement of G_α by β. The nodes of A have no higher neighbors in G_α, so $\hat{A} = A$. Hence by the Refinement Lemma, γ is a valid semiorder for G. But γ is a refinement of the numbering Φ defined in the statement of the lemma, so Φ is also valid. This completes the inductive step. ∎

Lemma 8.9 *Let α be any valid semiorder of a graph G, and let K be a clique contained in the highest class C of G_α. Suppose γ is obtained from α by first refining C by $(C - K, K)$ and then refining K arbitrarily. Then γ is valid.*

Proof: The only end-high paths introduced have endpoints in the clique K. ∎

The following is a special case of the Refinement Lemma, where all nodes in the class C have the same higher neighbors. In this special case, there is no need to consider $\hat{C}$.

Lemma 8.10 *Let Ψ be a valid semiorder for a graph G. Suppose C is a class of G_Ψ such that $G[C]$ is connected, and all nodes in C have the same higher neighbors in G_Ψ. Let Φ be a valid semiorder for C. Then the refinement of G_Ψ by Φ is valid.*

Proof: By the Refinement Lemma, we need only show that the refinement λ of $\widehat{C}_{\Psi_C}$ by Φ preserves validity. Assume Ψ is valid, so the nodes of $\widehat{C}$ not in C form a clique by Lemma 8.3. Let P be an end-high path in $\widehat{C}_\lambda$ with endpoints x and y. We must show that x and y are adjacent. If both endpoints are in C, they are already adjacent by the validity of Φ. If neither is in C, they belong to a clique, and hence are adjacent. Suppose therefore that x is in C and y is not. Since P is a weakly end-high path in $\widehat{C}_{\Psi_C}$ and an endpoint lies in C, all the internal nodes of P must also lie in C. Hence y is a higher neighbor of C in $\widehat{C}_\Psi$. Since all nodes in C have the same higher neighbors, it follows that y is a neighbor of x. ▮

8.4 The procedure STRATIFY

In this subsection, we outline the procedure STRATIFY for well-stratifying a class. The main work of the procedure is done in subprocedures which are described in succeeding subsections.

The procedure STRATIFY(G_Ψ, C) appears in Figure 33. If C is a connected nonsingleton class of G_Ψ, the procedure well-stratifies C. The procedure takes $O(\log n)$ time using t processors, where t is the number of edges of G with at least one endpoint in C. To achieve this processor bound, the procedure first identifies these edges by inspecting the adjacency lists of nodes in C, and subsequently never examines any other edges of G.

While inspecting the adjacency lists, the procedure also identifies the set B of higher neighbors of C in G_Ψ. The procedure uses the fact that if Ψ is a valid numbering of G, then the set of nodes B form a clique, by Lemma 8.3. The procedure assumes the existence of edges between nodes in B without ever checking for their presence. Specifically, in computing connected components of a graph involving nodes of B, the procedure uses an efficient connectivity algorithm such as that of Shiloach and Vishkin [44] or that of Gazit [21], suitably modified to take into account the fact that any two nodes of B are adjacent.

Namely, whenever we need to compute connectivity in a subgraph that includes nodes of B, we start by constructing a star graph containing all such nodes. The star graph connects up all these nodes using $k-1$ artificial edges, where k is the number of such nodes of B. Because we know that every two nodes of B are adjacent in the true graph, the artificial edges stand for true edges. The number of processors needed for handling these artificial edges is no more than the number of processors needed to handle the nodes of B. Hence the total number of processors needed for computing connected components in a node-induced subgraph of $G[C \cup B]$ is proportional to the number of nodes in the subgraph, plus the

Procedure STRATIFY(G_Ψ, C):

S1 For each node v in C, identify those edges connecting v to other nodes in C, and those edges connecting v to higher nodes.

S2 Let B be the set of higher neighbors of C.

S3 If B is empty, call NONE(G_Ψ, C), and end.

S4 If every node in B has at least $\frac{2}{5}|C|$ neighbors in C, call HIGHDEGREE(G_Ψ, C, B), and end.

S5 If there are nodes in B with fewer than $\frac{2}{5}|C|$ neighbors in C, call LOWDEGREE(G_Ψ, C, B).

Figure 33: The main procedure for finding a valid well-stratification.

number of edges with at least one endpoint being a node of C. The processor bound does not depend on the number of edges with both endpoints in B.

Depending on the nodes in B, the procedure STRATIFY(G_Ψ, C) calls one of three subprocedures, in which most of the work is done. In each procedure, we make use of *parallel prefix computation*, due to Ladner and Fischer [32]. In the next subsection, we describe the simplest of these procedures, called NONE, which is applicable when the class has no higher neighbors (i.e. when B is empty). This procedure illustrates the techniques used in refinement, and should be sufficient for all but the most dedicated and thorough reader. For that reader, however, we have described the other two procedures in the subsequent subsection.

8.5 The subprocedure NONE for stratifying a class with no higher neighbors

We now show that STRATIFY(G_Ψ, C) succeeds in finding a valid refinement that well-stratifies C. We shall refer to the nodes of C as *crimson* nodes, and the nodes of B as *blue* nodes. We consider three cases, corresponding to the three procedures. In this subsection, we consider only the first and simplest case, case I, when there are no blue nodes.

In this case, procedure NONE(G_Ψ, C), shown in Figure 36, is called. The procedure first identifies the set D of high-degree nodes: $D = \{v \in C \ : \ v \text{ has } > \frac{3}{5}|C| \text{ neighbors in } C\}$. The procedure then branches according to the following cases:

Subcase (1): Some component H of $C - D$ has size $> \frac{4}{5}|C|$. In this case, the procedure finds a connected subgraph H' of H such that the set $A = H' \cup \{\text{neighbors of } H' \text{ in C}\}$ includes between $n/5$ and $4n/5$ nodes. Then C is refined by the semiorder $(C - A, A)$, resulting in a well-stratification of C. The validity of the refinement follows from Corollary 8.7.

The procedure chooses H' as follows. First, a spanning tree T of H is constructed. Next, the nodes of H are arranged in some order consistent with their distance in T from the root: $v_1, \ldots, v_k$. This order has the property that any initial subsequence $v_1, \ldots, v_j$ induces

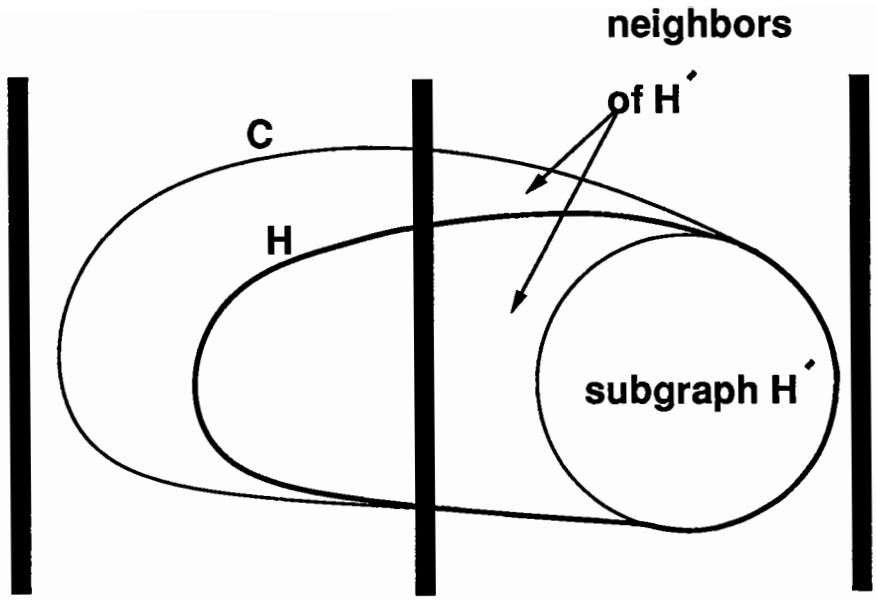

Figure 34: In Case I, subcase (1), we find a connected subgraph H' of H such that $H' \cup \{\text{neighbors of } H'\}$ has between $|C|/5$ and $4|C|/5$ nodes. Let A be $H' \cup \{\text{neighbors of } H'\}$. We refine C by the semiorder $(C - A, A)$. This refinement is valid because the neighborhood in A of any component of $C - A$ is a clique, by Corollary 8.6.

a connected subgraph of H.

Next, we carry out a parallel prefix computation. The goal is to identify the maximum m j such that the nodes $v_1, \ldots, v_j$ and their neighbors comprise at most $\frac{4}{5}|C|$ nodes of C.

For each node $v \neq v_1$ in C, let *earliest-nbr*(v) be the minimum i such that v is adjacent to v_i, or *undefined* if v has no neighbors among $v_1, \ldots, v_k$. Let *earliest-nbr*$(v_1) = 1$.

Now let us consider the inverse of the relation *earliest-nbr*. For each node v_i in T, *earliest-nbr*$^{-1}(v_i)$ is the set of neighbors of v_i in C that are not neighbors of any lower-numbered node. Let $f(v_i) = |\textit{earliest-nbr}^{-1}(v_i)|$, and let $g(\ell) = \sum_{i=1}^{\ell} f(v_i)$ for $\ell = 1, \ldots, k$. Then $g(\ell)$ is the number of nodes comprised by $v_1, \ldots, v_\ell$ and their neighbors in $\widehat{C}$. The function $g(\cdot)$ can be computed from $f(\cdot)$ by a parallel prefix computation.

After $g(\cdot)$ has been computed, it is easy to find the largest index j such that $g(j) \leq \frac{4}{5}|C|$. Let $\hat{\jmath}$ denote that index. That is, $\hat{\jmath}$ is the maximum j such that the nodes $v_1, \ldots, v_j$ and their neighbors comprise at most $\frac{4}{5}|C|$ nodes of C.

Finally, we let H' be the subgraph consisting of $v_1, \ldots, v_{\hat{\jmath}}$, and we let A be the subgraph consisting of H' and its neighbors. As mentioned before, the class C is then refined by $(C - A, A)$.

We need to show that the result is a well-stratification, i.e. that each of the subclasses is significantly smaller than the class. It is clear that A has at most $\frac{4}{5}|C|$ nodes. We must

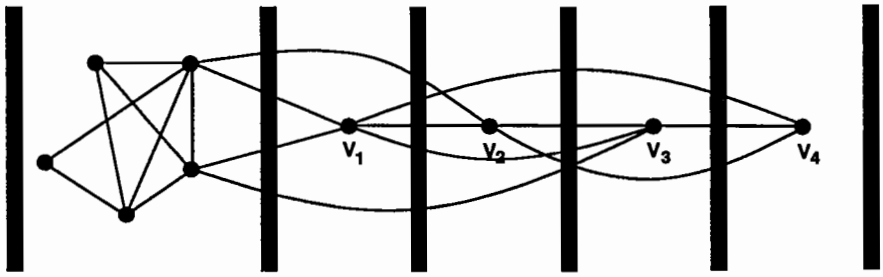

Figure 35: In Case I, subcases (2) and (3), we identify a clique $\{v_1, \ldots, v_k\}$. Then we place each node of the clique in a subclass by itself: we refine C by the semiorder $(C - \{v_1, \ldots, v_k\}, \{v_1\}, \{v_2\}, \ldots, \{v_k\})$. This refinement is valid because the neighborhood in $\{v_1, \ldots, v_k\}$ of any component of $C - \{v_1, \ldots, v_k\}$ is a clique.

show that it has at least $\frac{1}{5}|C|$ nodes, for then it will follow that $C - A$ has at most $\frac{4}{5}|C|$ nodes. Observe that by choice of $\hat{\jmath}$, nodes $v_1, \ldots, v_{\hat{\jmath}+1}$ and neighbors comprise *more than* $\frac{4}{5}|C|$ nodes. Not all these neighbors are neighbors of H'—some are neighbors of $v_{\hat{\jmath}+1}$, which is not in H'. But $v_{\hat{\jmath}+1}$ has at most $\frac{3}{5}|C|$ neighbors, since it is not in D. Hence leaving out the neighbors of $v_{\hat{\jmath}+1}$, we get at least $\frac{1}{5}|C|$ nodes in A.

Subcase (2): Each component of $C - D$ has size at most $\frac{4}{5}|C|$, and D is a clique. Let $D = \{v_1, \ldots, v_k\}$. In this case, we refine C by the semiorder $(C - D, \{v_1\}, \ldots, \{v_k\})$, placing each node of D in its own class. The components induced by the remaining nodes of C are small, so we have well-stratified C. The validity of the refinement follows from Lemma 8.9.

Subcase (3): D is not a clique. In this case, we choose two nonadjacent nodes x and y in D. At most $\frac{2}{5}|C|$ nodes of C are not neighbors of x, and so at least $\frac{1}{5}|C|$ neighbors of y are also neighbors of x. Let these common neighbors be $v_1, \ldots, v_k$. By Lemma 8.6, the common neighbors form a clique. We therefore proceed as in Subcase (2): we refine C by the semiorder $(C - \{v_1, \ldots, v_k\}, \{v_1\}, \ldots, \{v_k\})$, putting each of the common neighbors in its own class. Since there are so many common neighbors, the remaining nodes form a small subclass, and we have well-stratified C. The refinement is valid by Lemma 8.9.

8.6 The other subprocedures used in STRATIFY

In this subsection, we describe the other two subprocedures. The subprocedures and their justification are considerably more complicated than that described in the previous subsection. It is therefore suggested that this subsection be skipped on first reading.

There are two remaining cases to consider, cases II and III, corresponding to the two remaining subprocedures.

Case II: The set B of higher neighbors (called blue nodes) is not empty, and each blue node is adjacent to at least $\frac{2}{5}|C|$ crimson nodes. In this case, the procedure HIGHDEGREE(G_Ψ, C, B) of Figure 37 is called. The procedure arbitrarily orders the blue nodes: $B = \{v_1, \ldots, v_k\}$. For

Procedure NONE(G_Ψ, C):

N1 Let $D = \{v \in C : v \text{ has } > \frac{3}{5}|C| \text{ neighbors in } C\}$.

N2 Find a spanning forest of $C - D$.

N3 If some component H of $C - D$ has at least $\frac{4}{5}|C|$ nodes,

N4 Let T be the spanning tree of H.

N5 Arrange the nodes of H in some order consistent with their distance in T from the root: $v_1, \ldots, v_k$.

N6 For $1 \leq j \leq k$, let A_j denote the set consisting of $v_1, \ldots, v_j$ and neighbors of these nodes in C.

N7 Using parallel prefix computation, choose $\hat{\jmath} = \max\{j : |A_j| \leq \frac{4}{5}|C|\}$.

N8 Refine C by the semiorder $(C - A_{\hat{\jmath}}, A_{\hat{\jmath}})$.

N9 Otherwise (if every component of $C - D$ has fewer than $\frac{4}{5}|C|$ nodes),

N10 If D is a clique,

N11 Let $v_1, \ldots, v_k$ be the nodes of D.

N12 Refine C by the semiorder $(C - D, \{v_1\}, \ldots, \{v_k\})$.

N13 Otherwise,

N14 Let x and y be two nonadjacent nodes of D.

N15 Let $v_1, \ldots, v_k$ be their common neighbors. (They form a clique.)

N16 Refine C by the semiorder $(C - \{v_1, \ldots, v_k\}, \{v_1\}, \ldots, \{v_k\})$.

Figure 36: The subprocedure used to well-stratify C if C has no higher neighbors.

$1 \leq j \leq k$, let F_j be the set of nodes $v \in C$ such that v is adjacent to *all* the nodes $v_1, \ldots, v_j$. Then $\hat{\jmath}$ is chosen to be the maximum j such that F_j contains at least $|C|/5$ crimson nodes. (This step is similar to a step in the procedure NONE, described in the previous subsection, and can be implemented in essentially the same way.) The class C is then refined by the semiorder $(C - F_{\hat{\jmath}}, F_{\hat{\jmath}})$. The validity of the resulting numbering follows from Lemmas 8.8 and 8.9. At most $\frac{4}{5}|C|$ nodes of C remain in the lower subclass $C - F_{\hat{\jmath}}$. However, the set $F_{\hat{\jmath}}$ may be quite large. We consider two subcases.

Subcase (1): $\hat{\jmath} < k$. In this case, by choice of $\hat{\jmath}$, the set of crimson nodes adjacent to all the nodes $v_1, \ldots, v_{\hat{\jmath}+1}$ is less than $|C|/5$. Since there are at most $\frac{3}{5}|C|$ crimson nodes not adjacent to $v_{\hat{\jmath}+1}$, it follows that the number of nodes adjacent to all the nodes $v_1, \ldots, v_j$ is less than $\frac{1}{5}|C| + \frac{3}{5}|C| = \frac{4}{5}|C|$. Thus C has been well-stratified in this case.

Subcase (2): $\hat{\jmath} = k$. In this case, every node in $F_{\hat{\jmath}}$ is adjacent to every blue node. The procedure finds the largest component C' of $G[F_{\hat{\jmath}}]$, and calls NONE(G_Ψ, C'), which stratifies C' as if C' had *no* higher neighbors. The validity of the resulting numbering of $\widehat{C}$ follows from Lemma 8.10 and the Refinement Lemma. Since C' is well-stratified and $|C - C'| \leq \frac{4}{5}|C|$, C is well-stratified.

Procedure HIGHDEGREE(G_Ψ, C, B)
(Each node of B has at least $\frac{2}{5}|C|$ neighbors in C.)

H1 Arbitrarily order the nodes of B: $v_1, \ldots, v_k$.

H2 for $1 \leq j \leq k$, let F_j denote the set of nodes of C adjacent to *all* of the nodes $v_1, \ldots, v_j$.

H3 Using parallel prefix computation, choose $\hat{\jmath} = \max\{j \ : \ |F_j| \geq \frac{1}{5}|C|\}$.

H4 Refine C by the semiorder $(C - F_{\hat{\jmath}}, F_{\hat{\jmath}})$.

H5 Let C' be the largest component of $G[F_{\hat{\jmath}}]$.

H6 If $\hat{\jmath} = k$, then call NONE(G_Ψ, C').

Figure 37: The procedure HIGHDEGREE used to stratify C in case every higher neighbor of C has high degree in C.

Case III: The set B of higher neighbors is not empty, and Case II does not hold. In this case, the procedure LOWDEGREE(G_Ψ, C, B) in Figure 39 is called. The procedure defines D to be the set of nodes in $C \cup B$ having more than $\frac{3}{5}|C|$ crimson neighbors. The procedure then finds a spanning tree T of the (unique) component H of $G[(C \cup B) - D]$ containing blue nodes, and roots T at a blue node. The nodes of T are arranged in some order consistent with their distance from the root: $v_1, \ldots, v_k$. For $1 \leq j \leq k$, let A_j be the set consisting of $v_1, \ldots, v_j$ and neighbors of these nodes in $\widehat{C}$. Note that since v_1 is blue, and the blue nodes form a clique, every A_j contains all the blue nodes. The procedure chooses $\hat{\jmath}$ to be the maximum j such that A_j includes at most $\frac{4}{5}|C|$ crimson nodes. (This step is similar to a step in the procedure NONE, described in the previous subsection, and can be implemented in essentially the same way.)

Next, C is refined by the semiorder $(C - A_{\hat{\jmath}}, A_{\hat{\jmath}} \cap C)$, in step L7. To see that the resulting semiorder θ of $\widehat{C}$ is valid, first consider the intermediate semiorder $(C - A_{\hat{\jmath}}, A_{\hat{\jmath}})$ of $\widehat{C}$. Since $G[\{v_1, \ldots, v_{\hat{\jmath}}\}]$ is connected, the intermediate semiorder is valid by Corollary 8.7. To obtain the final semiorder θ of $\widehat{C}$ produced in step L7 from the intermediate numbering, we need only refine the class $A_{\hat{\jmath}}$ by the semiorder $(A_{\hat{\jmath}} - B, B)$; this refinement is valid by Corollary 8.9, since the blue nodes form a clique.

The higher subclass $A_{\hat{\jmath}} \cap C$ of C has size at most $\frac{4}{5}|C|$. However, the lower subclass $C - A_{\hat{\jmath}}$ may be quite large. The procedure finds the largest component C' of $C - A_{\hat{\jmath}}$, and proceeds according to the following two cases.

Subcase (1): $|C'| \leq \frac{4}{5}|C|$. In this case, we are done; C has been well-stratified.

Subcase (2): $|C'| > \frac{4}{5}|C|$. First, we observe that in this case, $\hat{\jmath} = k$. To see this, suppose $\hat{\jmath} < k$. Then the number of crimson nodes among $\{v_1, \ldots, v_{\hat{\jmath}+1}\}$ and neighbors is more than $\frac{4}{5}|C|$. Since $v_{\hat{\jmath}+1}$ is adjacent to at most $\frac{3}{5}|C|$ crimson nodes, it follows that the number of nodes whose numbers have increased is more than $\frac{4}{5}|C| - \frac{3}{5}|C| = \frac{1}{5}|C|$, which contradicts the fact that $|C'| > \frac{4}{5}|C|$.

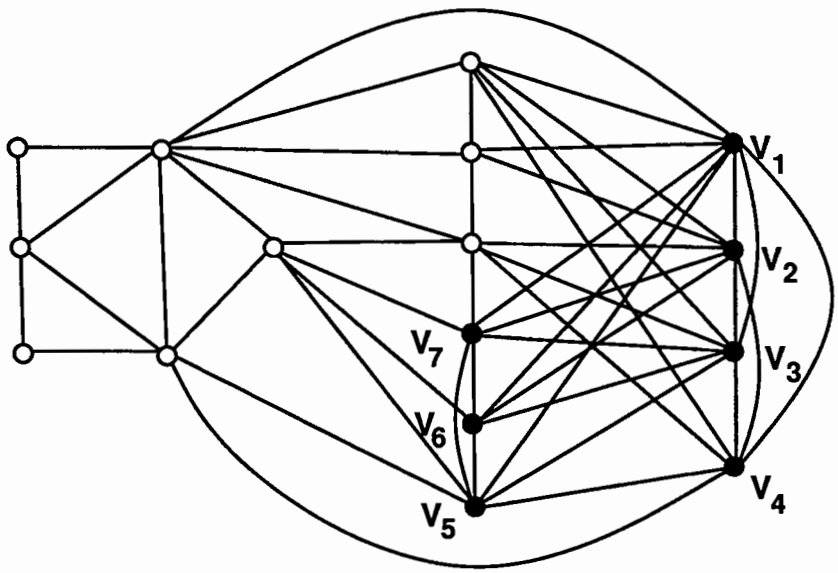

Figure 38: In case II, we start by finding a subset $\{v_1, \ldots, v_j\}$ of blue nodes such that at least one-fifth of the nodes of C are adjacent to *all* the nodes of the blue subset. For the graph illustrated, the subset would be $\{v_1, v_2, v_3, v_4\}$.

Procedure LOWDEGREE(G_Ψ, C, B)
(There exists a node in B having fewer than $\frac{2}{5}|C|$ neighbors in C.)

L1 Let $D = \{v \in C \cup B : v \text{ has } > \frac{3}{5}|C| \text{ neighbors in } C\}$.

L2 Let H be the connected component of $G[C \cup B - D]$ containing nodes of B.

L3 Find a spanning tree T of H, rooted at a node of B.

L4 Arrange the nodes of T in some order consistent with their distance in T from the root: $v_1, \ldots, v_k$.

L5 For $1 \leq j \leq k$, let A_j denote the set consisting of $v_1, \ldots, v_j$ and neighbors of these nodes in C.

L6 Using parallel prefix computation, choose $\hat{\jmath} = \max\{j \ : \ |A_j \cap C| \leq \frac{4}{5}|C|\}$.

L7 Refine C by the semiorder $(C - A_{\hat{\jmath}}, A_{\hat{\jmath}} \cap C)$.

L8 Let C' be the largest component of $C - A_{\hat{\jmath}}$.

L9 If $|C'| > \frac{4}{5}|C|$, then call STRATIFY(G_Ψ, C').

Figure 39: The procedure LOWDEGREE used to stratify C in case some higher neighbor of C has low degree in C.

Since $\hat{\jmath} = k$, every crimson node in T and every crimson neighbor of T ended up in the higher subclass. Therefore, C' contains no nodes of T and no neighbors of T. Every node in D has more than $\frac{3}{5}|C|$ crimson neighbors, and all but at most $\frac{1}{5}|C|$ of the crimson nodes are in C', so every node in D has more than $\frac{2}{5}|C|$ crimson neighbors in C'. Thus every higher neighbor of C' is adjacent to at least $\frac{2}{5}|C|$ nodes of C'. This shows that the recursive call to STRATIFY in step L9 results in a call to HIGHDEGREE, and not in a call to LOWDEGREE. Thus no further recursive calls occur. The recursive call well-stratifies C', and hence C as well, since $|C - C'| \leq \frac{1}{5}|C|$. The validity of the resulting numbering follows from the Refinement Lemma.

This completes the description of the algorithm STRATIFY for valid well-stratification of a class. At most one recursive call is made, as we have shown. The time for the algorithm is dominated by the time to compute connected components and find spanning trees, which is $O(\log |C \cup B|)$ using the algorithm of [44]; as discussed in Subsection 8.3, we need only $|C \cup B| + |E(C)| + |B| - 1$ processors, which is bounded by at most twice the number of edges with at least one endpoint in C. We have thus proved Theorem 8.4.

Using our algorithm for valid well-stratification in the procedure ITERATED REFINEMENT, we can therefore find a peo of a graph G in $O(\log^2 n)$ time using $O(n+m)$ processors.

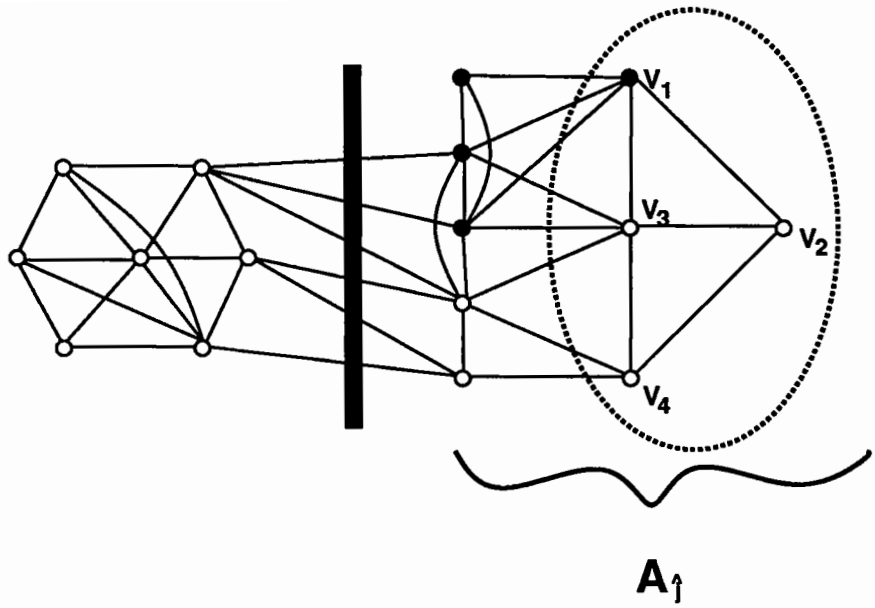

Figure 40: In Case III, we proceed similarly to Case I, subcase (1). We find a connected subgraph containing a blue node such that the subgraph together with its neighbors, collectively denoted $A_{\hat{\jmath}}$, contains at most four-fifths of the nodes of C. We next refine C by the semiorder $(C - A_{\hat{\jmath}}, A_{\hat{\jmath}} \cap C)$.

Appendix: Some elementary definitions

Let G be a graph. Throughout the paper, we let n and m denote the number of nodes and edges in G, respectively.

A path or cycle is *simple* if each node occurs only once. A k-cycle is a cycle consising of k nodes.

Let H be a subgraph of G or a subset of the nodes of G. We let $G[H]$ denote the subgraph of G consisting of nodes of H and edges both of whose endpoints are in H. We call $G[H]$ the subgraph of G *induced* by nodes of H. A *node-induced subgraph* of G is a subgraph of the form $G[H]$. We let $G - H$ denote the subgraph of G induced by nodes not in H. The *neighbors* of a subgraph H of G are the nodes of $G - H$ connected to H by edges of G. In case the nodes of G have weights, the weight of a subgraph is the sum of the weights of its nodes.

A graph H is a *clique* if there is an edge between every pair of nodes, and an *independent set* if it has no edges. We say H is a clique of G if H is a clique and a subgraph of G. We say H is a *maximal clique* of G if H is a clique of G and no clique H' of G strictly contains H. We say H is a *maximum clique* of G if H is a largest clique of G. We call $H_1, \ldots, H_k$ a *clique-cover* of G if each H_i is a clique of G and every node of G is in some H_i. A minimum clique-cover is one with a minimum number of cliques. We can assume without loss of

generality that the H_i's are node-disjoint. The above definitions also apply to independent sets. An independent set cover of G is called a *coloring* of G if the independent sets are node-disjoint. A minimum coloring is also called an optimal coloring.

Let P be a problem for which the best known sequential algorithm takes time $T(n)$ on instances of size n. A parallel algorithm to solve P is called *efficient* if the product of its processor requirement with its time requirement is within a polylogarithmic factor of $T(n)$. For example, if the parallel algorithm requires $T(n)$ processors and $O(\log^2 n)$ time, it is efficient.

References

[1] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts (1974).

[2] A. Aggarwal and R. Anderson, "A random NC algorithm for depth first search," *19th STOC* (1987), pp. 325-334.

[3] R. J. Anderson and G. L. Miller, "Optimal parallel algorithms for list ranking," extended abstract, 1986.

[4] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, "On the desirability of acyclic database schemes," *J. ACM 30* (1983), pp. 479-513.

[5] C. Berge, *Graphs and Hypergraphs*, North-Holland, Amsterdam (1973).

[6] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *J. Comp. Sys. Sci. 13:3* (1976), pp. 335-379.

[7] P. Buneman, "A characterization of rigid circuit graphs," *Discrete Math. 9* (1974), pp. 205-212.

[8] N. Chandrasekharan and S. S. Iyengar, "NC algorithms for recognizing chordal graphs and k-trees," Tech. Rept. #86-020, Dept. of Comp. Sci., Louisiana State Univ. (1986).

[9] C. J. Colbourn, K. S. Booth, "Linear time automorphism algorithms for trees, interval graphs, and planar graphs," *SIAM J. Comp. 10:1* (1981), pp. 203-225.

[10] R. Cole, "Parallel merge sort," *27th FOCS* (1986), pp. 511-516.

[11] R. Cole and U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree, and graph problems," *27th Annual Symp. on Foundations of Computer Science* (1986), pp. 478-491.

[12] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *19th STOC* (1987), pp. 1-6.

[13] E. Dahlhaus and M. Karpinski, "The matching problem for strongly chordal graphs is in NC," Tech. Rept. 855-CS, Institut für Informatik, Universität Bonn, 1986.

[14] E. Dahlhaus and M. Karpinski, "Fast parallel computation of perfect and strongly perfect elimination schemes," Tech. Rept. 8519-CS, Institut für Informatik, Universität Bonn, 1987, and IBM Research Report RJ5901, October 1987.

[15] G. A. Dirac, "On rigid circuit graphs," *Abh. Math. Sem. Univ. Hamburg 25* (1961), pp. 71-76.

[16] A. Edenbrandt, *Combinatorial Problems in Matrix Computation*, TR-85-695 (Ph.D. Thesis), Department of Computer Science, Cornell University (1985).

[17] A. Edenbrandt, "Chordal graph recognition is in NC," *Inf. Proc. Let. 24* (1987), pp. 239-241.

[18] D. Fulkerson and O. Gross, "Incidence matrices and interval graphs," *Pac. J. Math 15* (1965), pp. 835-855.

[19] F. Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph," *SIAM J. Comp. 1* (1972), pp. 180-187.

[20] F. Gavril, "The intersection graphs of subtrees in trees are exactly the chordal graphs," J. Combin. Theory B 16 (1974), pp. 47-56.

[21] H. Gazit, "An optimal randomized parallel algorithm for finding connected components in a graph," *27th FOCS* (1986), pp. 492-501.

[22] H. Gazit, G. L. Miller, "An improved parallel algorithm that computes the BFS numbering of a directed graph," *Inf. Proc. Let. 28* (1988), pp. 61-65.

[23] J. R. Gilbert, D. J. Rose, and A. Edenbrandt, "A separator theorem for chordal graphs," *SIAM J. Alg. Disc. Meth. 5* (1984), pp. 306-313.

[24] P. C. Gilmore and A. J. Hoffman, "A characterization of comparability graphs and of interval graphs," *Can. J. Math 16* (1964), pp. 539-548.

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

[26] C.-W. Ho and R. C. T. Lee, "Efficient parallel algorithms for finding maximal cliques, clique trees, and minimum coloring on chordal graphs," *Inf. Proc. Lett. 28* (1988), pp. 301-309.

[27] C.-W. Ho and R. C. T. Lee, "Counting clique trees and computing perfect elimination schemes in parallel," *Inf. Proc. Lett. 31* (1989),pp. 61-68.

[28] P. N. Klein, *Efficient parallel algorithms for planar, chordal, and interval graphs*, TR-426 (Ph.D. thesis), Laboratory for Computer Science, MIT (1988).

[29] P. N. Klein, "Efficient parallel algorithms for chordal graphs," *Proc. 29th Symp. Found. of Comp. Sci.* (1989), pp. 150-161.

[30] P. N. Klein and J. H. Reif, "An efficient parallel algorithm for planarity," *J. Comp. Sys. Sci.37* (1988), pp. 190-246. A preliminary version appeared in 27th FOCS (1986), pp. 465-477.

[31] D. Kozen, U. Vazirani, and V. Vazirani, "NC algorithms for comparability graphs,and testing for unique perfect matching," *Proc. 5th Symp. Found. of Software Technology and Theor. Comp. Sci.*, published as *Lecture Notes in Computer Science 206*, Springer-Verlag, New York (1985), pp. 496-503.

[32] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM 27:4* (1980), pp. 831-838.

[33] G. S. Lueker and K. S. Booth, "A linear time algorithm for deciding interval graph isomorphism," *J. ACM 26:2* (1979), pp. 183-195.

[34] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," *26th FOCS* (1985), pp. 478-489.

[35] J. Naor, M. Naor, and A. A. Schäffer, "Fast parallel algorithms for chordal graphs," *19th STOC* (1987), pp. 355-364; also submitted to SIAM J. Comput.

[36] M. B. Novick, personal communication (1987).

[37] M. B. Novick, personal communication (1988).

[38] M. B. Novick, 'Logarithmic time parallel algorithms for recognizing comparability and interval graphs," manuscript.

[39] J. H. Reif, "An optimal parallel algorithm for integer sorting," *Proc. 26th FOCS*, pp. 496-503.

[40] D. J. Rose, "Triangulated graphs and the elimination process," *J. Math Anal. Appl. 32* (1970), pp. 597-609.

[41] D. J. Rose, R. E. Tarjan, and G. S. Lueker, "Algorithmic aspects of vertex elimination on graphs," *SIAM J. Comp. 5* (1976), pp. 266-283.

[42] J. E. Savage and M. G. Wloka, "A parallel algorithm for channel routing," *Proceedings of WG '88, Graph-theoretic Concepts in Computer Science*, published as *Lecture Notes in Computer Science*, Springer-Verlag, New York (1988).

[43] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Trans. on Mathematical Software 8:3* (1982), pp. 256-276.

[44] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *J. Algorithms 3* (1982), pp. 57-67.

[45] R. E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time," *25th FOCS* (1984), pp. 12-22.

[46] J. R. Walter, *Representations of Rigid Circuit Graphs*, Ph.D. thesis, Wayne State Univ. (1972)

[47] J. R. Walter, "Representations of chordal graphs as subtrees of a tree," *J. Graph Theory 2* (1978), pp. 265-267.