An algorithm for computing an elimination tree of minimum height for a tree

Fredrik Manne

Abstract

The elimination tree is a rooted tree that is computed from the adjacency graph of a symmetric matrix A. The height of the elimination tree is one restricting factor when solving a sparse linear system $Ax = b$ on a parallel computer using Cholesky factorization. An efficient algorithm is presented for the problem of ordering the nodes in a tree G so that its elimination tree is of minimum height. Its running time is $O(n \log n \log d)$ where n is the number of nodes in G and d the maximum degree of any node in G. The number of fill edges caused by this algorithm is less than n.

We also show that there exists a minimal separator ordering on any matrix such that the resulting elimination tree is of minimum height. Implications of these results are given for the computation of elimination trees of minimum height for more general classes of graphs.

1 Introduction and motivation

Consider using Cholesky factorization to solve the linear system $Ax = b$, where A is an $n \times n$ sparse symmetric positive definite matrix. On a sequential computer this is usually done in four separate stages:

1. *Ordering.* Determine a permutation matrix P so that the Cholesky factor L of PAP^T will suffer little fill.

2. *Symbolic factorization.* Determine the structure of the nonzeros of L and set up a data structure in which to store A and compute the nonzero entries of L.

3. *Numeric factorization.* Insert the nonzeros of A into the data structure and compute the numeric values of L.

4. *Triangular solution.* Solve $Ly = Pb$ and $L^T z = y$, and then set $x = P^T z$.

Both the ordering of A and the symbolic factorization are independent of the numerical entries of A and can be performed completely on the adjacency graph G of A [20, 22]. Determining a permutation matrix is equivalent to finding an ordering on the nodes of G, while the symbolic factorization uses a graph elimination process to compute the zero-nonzero structure of L. The reason for finding a permutation matrix for A in the first stage so that L suffers little fill is that the amount of work and storage needed in the subsequent stages depends on the number of nonzeros in L.

Most parallel algorithms for performing sparse Cholesky factorization operate in the same four stages. In the first stage, the only difference when ordering A, is that not only must we keep the fill low, but we must also consider whether PAP^T is suitable for parallel methods. For a more detailed overview of the different aspects of parallel sparse Cholesky factorization see [8].

What is unique to sparse Cholesky factorization compared to dense Cholesky factorization, is that each column does not generally depend on all the previous columns. The elimination tree (etree) [17, 24] is a data structure that describes the dependencies among the columns of A when factoring A into LL^T. This is true both for the symbolic factorization and for the numeric factorization. For a dense matrix the etree would be a single path indicating that each column is dependent on all the previous columns. For a sparse matrix one can expect the etree to contain branching and therefore to be lower.

The fact that disjoint parts of the etree can be factored independently of each other gives a potential for a high-level parallelism for sparse matrices that does not exist for

dense matrices. This is also referred to as *large grained parallelism* [15]. Exploitation of this fact is essential in producing good parallel algorithms. The lower the etree and the more branching it contains, the more parallelism there is to exploit. Therefore it would be desirable to find a P that both lowers the etree of PAP^T and keeps the Cholesky factor L of PAP^T sparse. It is however not difficult to show that these two requirements might be in conflict with each other. To make matters worse, both the problem of minimizing fill and the problem of finding the lowest possible etree are known to be NP-hard [21, 26].

There are methods for finding orderings that give low etrees and few fill edges. Nested dissection is a method for ordering G that was developed to reduce fill [5, 6] and has also been shown to produce low etrees [13]. Another approach is first to compute a fill-reducing ordering P and then to find an equivalent ordering (giving the same fill edges) of the adjacency graph G^* of $L + L^T$ that produces a low etree. The main motivation behind this approach is that one knows how to efficiently compute an ordering for G^* such that its etree is of minimum height under the restriction that no new fill edges are introduced [10, 16, 14].

Little is known about how to compute etrees of minimum height for classes of graphs when additional fill is allowed, and how much fill this might cause. We show that for any graph, the class of minimal separator orderings contains an ordering giving an etree of minimum height.

However, the main result presented in this paper is an efficient algorithm that solves the minimum height problem when the original graph is itself a tree. The algorithm will be shown to have time complexity $O(n \log n \log d)$, where d is the maximum degree of any node in G. This is the first efficient algorithm for computing an etree of minimum height for a nontrivial class of graphs. We will also show that when solving this problem we introduce at most $n - 1$ fill edges.

A subexponential algorithm for solving the minimum height etree problem for trees was first presented in [18]. It was based on the following idea: Let x be the root of an etree where each subtree of the etree hanging from x is of minimum height. It was then shown that at most $\log n$ etrees with this property have to be created before an etree of minimum height is found. However, the time complexity of this algorithm was $O(n^{\log \log n})$. In this paper we will demand not only that each subtree of the etree hanging from the root is of minimum height, but also that each subtree is *tilted*. As shown in subsequent sections, such an etree can be reordered into an etree of minimum height in time proportional to the height of the etree. The idea of tiltedness is similar to the use of leftist heaps [3, 12] to facilitate efficient merging of heaps.

It is true that few matrices of practical importance have adjacency graphs that are trees. Still, as we shall discuss, being able to compute etrees of minimum height for trees points to the possibility of solving this problem for more general classes of graphs such as chordal graphs. We will also show how the algorithm developed in this paper can be generalized to a heuristic for reducing the height of an etree for a general graph.

The outline of this paper is as follows. In Section 2 we show how the etree of A is constructed. We also present the notation that will be used in this paper. Nested dissection is presented in Section 3 along with a discussion of how it works on trees. The main algorithm is presented in Section 4 along with a short motivation. Some preliminary results on the height of an etree of minimum height are given in Section 5. In Sections 6 and 7 we prove that the algorithm presented in Section 4 produces an etree of minimum height for a tree. Some of the details of the implementations of the algorithm in Section 4, which depend on the fact that the algorithm produces an etree of minimum height, are given in Section 8. The time complexity of the algorithm is discussed in Section 9 and the amount of fill edges it causes is discussed in Section 10. In Section 11 we consider the problem of computing low etrees for more general graphs than trees. Finally, in Section 12 we summarize.

2 Etrees and notation

We now explain how the elimination tree of A is constructed, and also present some graph notation that might be unfamiliar to the reader. The reader is assumed to be familiar with standard graph notation. For a more thorough introduction to elimination trees see [17].

The adjacency graph $G = (V, E)$ of A is formed by taking n nodes and adding an edge (i, j) for each $a_{ij} \neq 0$, $i < j$. The set of nodes of G is denoted by $V(G)$ and the set of edges by $E(G)$. The adjacency graph of $L + L^T$ is called the filled graph and is denoted by G^*. It can be computed from G by the following algorithm:

Filled_Graph(G)

Set all nodes unmarked.

Iterate the following step n times:

Select an unmarked node v and add edges to G such that all v's unmarked neighbors are adjacent (i.e. the unmarked neighbors form a *clique*), and then mark v.

End Filled_Graph

The order in which the nodes are selected when forming G^* is called an *elimination ordering*, and the edges added to G to form G^* are called *fill edges*. If a node v is selected before a node w in an elimination ordering we write $v < w$. A perfect elimination ordering is an ordering such that $G = G^*$. The class of chordal graphs is exactly the class of graphs for which perfect elimination orderings exist [22]. If G is a tree then a perfect elimination ordering can be found by selecting a leaf in the unmarked graph in each iteration of the above algorithm.

The elimination tree T of G is a directed graph such that $V(T) = V(G)$ with the directed edge $\langle i, j\rangle \in E(T)$ if and only if $j = \min\{k \mid (i,k) \in E(G^*), k > i\}$. If $\langle i, j\rangle \in E(T)$ then j is the parent of i and is denoted by $p(i)$.

If A is irreducible then G is connected and T is a rooted tree. Throughout this paper we will assume that A is irreducible.

The root of T is the node with only incoming edges and is denoted by $root(T)$. By $T[x]$ we mean the subtree of T induced by x and all its descendants in T; x is the root of this subtree.

The height of a leaf in T is 0. The height of a node v in T is denoted by $h(v)$ (or $h(T[v])$) and is defined as $\max\{h(w) + 1 \mid p(w) = v\}$. The height of T is the height of the root and is denoted by $h(T)$. The depth of a node $x \in V(T)$ is the length of the path from x to $root(T)$.

If T is an etree for G such that for any other etree T' for G we have $h(T) \leq h(T')$, we then say that T is an etree of minimum height. If T is an etree of minimum height we denote its height by $mh(G)$.

We do not distinguish between the nodes in T and G. This means that if B is a connected component of G such that $V(T[x]) = V(B)$ we write both $mh(B)$ and $mh(T[x])$ for the height of an etree of minimum height for B. If $h(T[x]) = mh(B)$ we say that $T[x]$ is of minimum height.

A topological ordering of T is an ordering where each descendant of node x is ordered before x. If T' is found by a topological ordering of T we do not distinguish between T and T'. This is because T and T' have the same structure, and the filled graphs of T' and T have the same structure.

Let $K = \{w_1, w_2, ..., w_k\}$ be a set of nodes in T such that $p(w_i) = w_{i+1}$ and w_i is the only child of w_{i+1}, $1 \leq i < k$. Then the nodes in K induce a *chain* in T.

A monotone path in G is a path $w_1, w_2..., w_l$ such that $w_j < w_{j+1}$ for $1 \leq j < l$. If S is a set of nodes, $G - S$ is the induced subgraph of G containing all nodes not in S. We write $G - v$ if $S = \{v\}$. If $G - S$ contains more than one component then S is a *separator*.

The neighbors of a node v are denoted by $adj(v)$. The degree of v is $|adj(v)|$. If S is a set of nodes we write $adj(S) = \{x \mid (x,v) \in E(G) \text{ for some } x \notin S \text{ and } v \in S\}$.

If G is a tree we write $bw(x,v)$ to denote the component B of $G - \{x,v\}$ such that $x, v \in adj(B)$. Thus $bw(x,v)$ is the unique component of G "between" x and v. If $(x,v) \in E(G)$ then $bw(x,v) = \emptyset$. If G is a tree and $x, v \in V(G)$ we denote the length of the unique path from x to v by $dist(x,v)$.

3 Some aspects of computing an etree

In this section, we study a top-down approach for computing an etree as opposed to the one given in Section 2. This leads us to characterize elimination orderings in terms of separators in G. This is of importance not only as a tool, but as we show, the class of minimal separator orderings always contain an ordering giving an etree of minimum height. We first present the following lemma [19]:

Lemma 3.1 *Let u be the lowest common ancestor in T of two given nodes x and y. Let S be the set of nodes on the path $u, w_1, w_2, ..., w_l = root(T)$ in T. Then $T[x]$ and $T[y]$ are disjoint if and only if x and y are in different connected components of $G - S$.* □

Let T be an etree for a graph G and let S be the set of nodes on the chain $w_1, w_2, ..., w_l = root(T)$ in T. From Lemma 3.1 it follows that if B is a nonempty component of $G - S$ then there exists a subtree $T[z]$ such that $p(z) = w_1$ and $V(T[z]) = V(B)$. Furthermore, from the same lemma it also follows that the nodes in $V(T[x])$ induce a connected component of G for any node $x \in V(G)$.

Using these observations we now give a top-down approach to computing T directly from G. We let t be a dummy node such that $p(root(T)) = t$. If G has already been ordered, then the call Order(G,t), will compute the etree of G.

Order(B,v)
  $u :=$ The highest numbered node in B;
  $p(u) := v$;
  For each connected component C of $B - u$
    Order(C,u);
End Order

Note that the order in which the components of $B - u$ are processed by Order() has no effect on the structure of the resulting etree. If the ordering of G is not given in advance, Order() can be used to compute an ordering on G by choosing any node $u \in V(B)$ to be the highest numbered node in $V(B)$.

If the removal of the node u in Order() does not disconnect B then there is at most one component of $B - u$. Thus one can alter the algorithm to keep on choosing nodes $K = \{u_1, u_2, ..., u_l\}$ until either $B - K$ contains at least two connected components or until $K = B$. Then we will have $p(u_j) = u_{j-1}$, $2 \leq j \leq l$, and the recursive call would be for each component of $B - K$, with the node u_l passed along to the next level. We call the resulting ordering a *separator ordering*. If K has been chosen so that the remaining components of $B - K$ are balanced, meaning that no component of $B - K$ contains more than $c|V(B)|$ nodes for some constant $c < 1$, we get a *nested dissection* ordering.

For any tree G of n nodes there is a node v such that no component of $G - v$ contains more than $\lfloor n/2 \rfloor$ nodes [2, 11]. Thus for a tree, nested dissection can be used to find an etree of height at most $\lfloor \log n \rfloor$. But this might still be far from an optimal solution. It has been shown [9] that there for $k > 0$ exists a tree G with $n = 2^{2^k}$ nodes such that $mh(G) = k$ where nested dissection would give an etree of height 2^k.

We now look at another class of orderings for general graphs that can be obtained by modifying Order(). This will be used in the proof of Lemma 9.1. If instead of imposing constraints on the size of each component of $B - K$, we require that each chosen K is a minimal separator in G, we get a *minimal separator ordering*. As the following lemma and theorem show there exists a minimal separator ordering on any graph giving an etree of minimum height.

Lemma 3.2 *Let G be a graph with etree T. Then there exists a minimal separator ordering on G with resulting etree T' such that $h(T') \leq h(T)$.*

Proof: The proof is by induction on the number of nodes in G. The result is trivially true if $|V(G)| = 1$. Assume that the result is true for graphs with fewer than n nodes and that $|V(G)| = n > 1$. There are two cases to consider, depending on the structure of T.

The first case is when T is a chain. If G is a clique any etree for G must be a chain and the result follows. If G is not a clique then there exists some minimal separator C in G. Order C last and the nodes in $G - C$ in any order. This will not increase the height of the etree and the result follows by applying the induction hypothesis on each component of $G - C$.

The second case is when T is not a chain. Let K be the uppermost maximal chain in T. Note first that reordering the nodes in K will not increase the height of the elimination tree. Since $G-K$ is disconnected, K is a separator in G and must contain a set of nodes C such that $G-C$ is disconnected and no subset of nodes in C disconnects G. Then C is a minimal separator. Reorder K such that the nodes in C are ordered last and the nodes in $K-C$ second to last leaving the rest of the etree unchanged. The height of the elimination tree has not increased and a minimal separator is now eliminated last. The result now follows from the induction hypothesis. □

Since Lemma 3.2 is true for any etree for a graph we have the following result:

Theorem 3.3 *Let G be a graph. Then there exists a minimal separator ordering on G with resulting etree T such that $h(T) = mh(G)$.* □

Although Theorem 3.3 is unconstructive, it still points to how etrees of minimum height can be constructed for restricted classes of graphs. This will be discussed further in Section 11.

We now present another observation that can be made from Order(). This result will be used in the proof of Lemma 9.1. Let T be the etree of a general graph G. Let $x \in V(T)$ and let B be the component of G induced by the nodes in $T[x]$. Then x will have as many children in T as there are connected components of $B-x$. Since $B-x$ cannot contain more connected components than the number of neighbors of x in G we have the following result:

Lemma 3.4 *Let G be a graph and $v \in V(G)$ a node of degree d. Then v can have no more than d children in any etree for G.* □

4 A reordering algorithm

In this section, we describe an algorithm for computing a low etree for a tree G. As will be shown in the subsequent sections, this algorithm in fact computes an ordering on G that results in an etree of minimum height.

The first step of the algorithm is to order G by any perfect elimination ordering and to compute the etree T for G. The algorithm then tries to reduce the height of T through a series of local reordering steps called *rotations*. These rotations are analogous to the rotations used to maintain balanced binary trees as described in [25]. For the rest of this section we will assume that G is a tree with $|V(G)| > 1$.

We start by defining a rotation. Let T be an etree for G such that $v = root(T)$ and let x be a child of v in T. Let $B = bw(x, v)$. If $(x, v) \notin E(G)$ then B is nonempty. In this case let z be the root of the subtree in T consisting of the nodes in $V(B)$. Then $p(z) = x$ in T. The etree T is as shown in Figure 1a. Without loss of generality we assume that x and v are consecutively ordered. Then a rotation with respect to x, consists of letting x and v switch places in the elimination ordering. From the nested dissection view, this is equivalent to choosing x as the first separator instead of v. The node v is then chosen first among the nodes in the component of $G - x$ that contains v. This means that $p(v) = x$ after the rotation. Since B and v are now in the same component of $G - x$ we have $p(z) = v$ after the rotation. All other parts of T remain unchanged in the new etree. The new etree is as shown in Figure 1b.

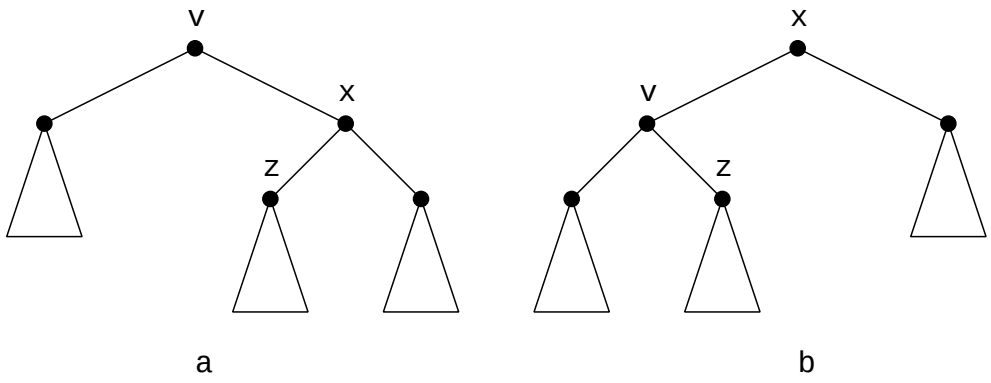

Figure 1:

We now present the code to perform a rotation. For the time being we assume that if $(v, x) \notin E(G)$ then the function $s(x)$ returns a pointer to the node z. If $(v, x) \in E(G)$ then $T[z]$ is empty. In that case we assume that $s(x) = x$. In Section 8 we will show how to initialize and maintain $s(x)$ when the etree is reordered.

Rotate(x)
    $v := p(x)$;
    $p(x) := p(v)$;
    $p(v) := x$;
    If $s(x) \neq x$
        $p(s(x)) := v$;
End Rotate

It is possible to perform Rotate(x) also when v is not the root of T. Thus we can perform a rotation with respect to any node in T other than $root(T)$. Also note that

the effect of performing Rotate(x) can be reversed by performing Rotate(v). Thus Rotate(x) followed by Rotate(v) leaves the etree unchanged.

We now study how rotations can be used to reduce the height of an etree. Let T be as in Figure 1a and let T' be the etree after Rotate(x). Then T' is as in Figure 1b. The rotation is called *successful* if $h(T') < h(T)$ and *unsuccessful* if $h(T') \geq h(T)$. Let $h(T) = k$ and assume that x is the only child of v in T of height $k-1$. Then $h(T'[v]) \leq k-1$ and each child $y \neq v$ of x in T' satisfies $h(T'[y]) \leq k-2$. Thus $h(T') \leq h(T)$. Since v is the only child of x in T' that can be of height $k-1$ it follows that the rotation is successful if and only if $h(T'[v]) \leq k-2$.

Even if $h(T'[v]) = k-1$ we can still hope to find an etree for G of lower height than k. This can be accomplished if we can reorder $T'[v]$ into an etree of height $\leq k-2$. Based on this idea we now present the outline of a simple algorithm for reducing the height of an etree. It takes an etree $T[v]$ as input.

The algorithm operates by performing a sequence of rotations $r_1, r_2, ..., r_l$ with respect to the current highest child of v. The rotations are performed until either v is a leaf in the etree or until v has at least two children of maximum height. Finally the algorithm reverses a maximal sequence of rotations $r_l, r_{l-1}, ..., r_k$ such that each r_i, $k \leq i \leq l$ was unsuccessful.

Note that reversing the rotations r_l through r_k can be done by performing Rotate(v) $k-l+1$ times.

Let T_i be the etree after rotation r_i, $0 \leq i \leq l$, ($T = T_0$). Then the height of $T_{i-1}[v]$ will be reduced if and only if $T_i[v]$ is ordered into an etree of height less than $h(T_{i-1}[v]) - 1$. Thus by a simple inductive argument we have that if r_i is the first successful rotation then $h(T_j) < h(T_0)$, $k \leq j \leq i$. Since the algorithm returns T_{k-1} where either $k = 1$ or r_{k-1} is the last successful rotation, it follows that if at least one rotation is successful then the algorithm will be able to reduce the height of $T[v]$.

Before performing each rotation in the algorithm we store the current height of v in a stack S. This way we can test if the rotation was successful by comparing the topmost element of S with $h(v)$. We use the operation *push* to place a new element on top of the stack and the operation *pop* to remove the topmost element and return its value. We assume that each node has at least two children in T. This is accomplished by adding dummy nodes of height -1 as children to each node with fewer than two children. To avoid having to treat $root(T)$ separately we assume that there exists a special node $t \notin V(G)$ such that $p(root(T)) = t$. In the algorithm we assume that T is

the data structure containing the etree.

Tilt(v)
  x := A highest child of v in T;
  y := A second highest child of v in T;
  S := {};

  While ($h(v) > 0$) **and** ($h(x) > h(y)$)
    push($S, h(v)$);
    Rotate(x);
    x := A highest child of v in T;
    y := A second highest child of v in T;
  End While

  While ($|S| > 0$) **and** ($h(v) =$ **pop**(S) - 1)
    Rotate(v);
End Tilt

We now give an example showing how Tilt can be used to reduce the height of an etree.

Example 4.1 Consider the tree G in Figure 2a. Figure 2b shows a possible etree T for G. Note that the only tree edge which is not in G is $\langle f, b \rangle$. Figure 2c through 2e show the effect of performing Tilt(b) on T. As can be seen, three rotations are performed with respect to the current highest child of b. Only the first rotation is successful. Thus the algorithm backtracks and returns the etree in Figure 2c.

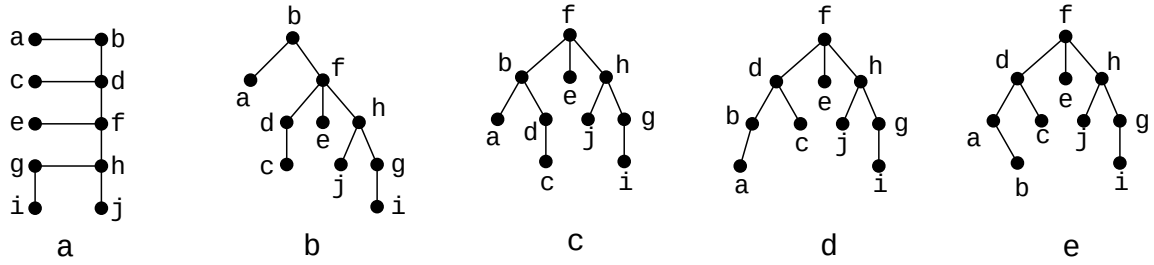

Figure 2:

Note that Tilt can also be implemented as a recursive procedure. If $h(v) > 0$ and $h(x) > h(y)$ then Tilt would perform Rotate(x) thereby reordering $T[v]$ into $T'[x]$. Tilt then calls itself recursively with $T'[v]$ as argument. Let T'' be the resulting etree from this call and let r be the child of x in T'' such that $V(T''[r]) = V(T'[v])$. Then if

$h(T''[r]) = h(T[v]) - 1$ no successful rotation has been performed and each but the first rotation has been reversed. Thus $r = v$ and the first rotation is reversed by performing Rotate(r). If $h(T''[r]) < h(T[v]) - 1$ then at least one rotation has been successful and T'' is left unchanged. In Section 7 we will use this recursive formulation of Tilt to prove that Tilt reorders a special kind of etree into an etree of minimum height.

We now state our main algorithm for computing a low etree for the tree G. It is basically a driver routine for Tilt.

Min_Height
Order G by any perfect elimination ordering;
Construct T;

Perform a postorder traversal of T and for each visited node $v \neq t$ do:
Tilt(v);
End Min_Height

The discussion on the time complexity of the Min_Height algorithm is postponed until Section 9. This is because the time complexity is strongly dependent on the height of v when performing Tilt(v). Note however that the initial ordering of G and the construction of T can be done in $O(n)$ time. Sections 5 through 10 will concentrate on showing that Min_Height produces an etree for G of minimum height and in doing so it does not result in more than $n - 1$ fill edges.

5 Minimum height etrees

In this section we present a lemma on $mh(G)$ and $mh(B)$ where B is a subgraph of a general graph G, along with two corollaries. These results will be used when proving that Min_Height finds an etree of minimum height for a tree.

Lemma 5.1 *Let G be a connected graph and B a connected subgraph of G. Then $mh(B) \leq mh(G)$.*

Proof: Let α be an ordering on G that gives an etree of minimum height. Order the nodes in B in the same relative order as in α. Let $(x, y) \in E(B^*)$ be a fill edge in B^*. By the path lemma [23] we know that there exists a path from x to y in B through nodes numbered lower than both x and y. Since the nodes in B are ordered in the same relative order as in G we see that this path must also exist in G and that $(x, y) \in E(G^*)$.

Thus it follows that $E(B^*) \subseteq E(G^*)$. It is known [16] that the height of the etree is the length of the longest monotone path in the filled graph. Since $E(B^*) \subseteq E(G^*)$, we see that any monotone path in B^* must also exist in G^*. Thus it follows that B has an etree that is no higher than $mh(G)$. □

The first corollary shows how close the height of an etree is to the minimum height when each subtree hanging from the root of the etree is of minimum height.

Corollary 5.2 *Let G be a graph with etree T where each component of $G - root(T)$ is a subtree of T of minimum height. Then $h(T) = mh(G)$ or $h(T) = mh(G) + 1$.*

Proof: By Lemma 5.1 each component B of $G - root(T)$ has $mh(B) \leq mh(G)$. Thus $h(T) \leq mh(G) + 1$. The result follows since $mh(G) \leq h(T)$. □

The next corollary shows that a certain type of etree is of minimum height.

Corollary 5.3 *Let G be a graph with etree T where each component of $G - root(T)$ is a subtree of T of minimum height. If the two highest children of v in T are of equal height then T is of minimum height.*

Proof: Since v has at least two children of minimum height k, $G - v$ must contain two connected components B and C such that $mh(B) = k$ and $mh(C) = k$. Thus any etree with v as root will be of height $\geq k + 1$. Let T' be an etree for G where $x = root(T')$ and $x \neq v$. The component D of $G - x$ that contains v also contain either B or C as a subgraph. By Lemma 5.1 $mh(D) \geq k$ giving $h(T') \geq k + 1$. □

6 Tilted etrees

In this section, we will define a special kind of etree for a tree, called a *tilted etree*. We will show that a tilted etree is well defined and that it is always of minimum height. As we shall see in Section 7, the algorithm Find_Min produces a tilted etree. Before the definition we need some intermediate results. The first result concerns which node can be the root of an etree of minimum height. From this section through Section 10 we will assume that G is a tree on n nodes.

Lemma 6.1 *Let T and T' be two etrees for G such that $h(T) = h(T') = mh(G)$. Then for each node w_j on the path $root(T), w_1, w_2, ..., w_l$,$root(T')$ in G, there exists an etree T_j for G, such that $root(T_j) = w_j$ and $h(T_j) = mh(G)$.*

Proof: Let $k = mh(G)$, $x = root(T)$ and $y = root(T')$. Let w_j be a node on the path $x, w_1, w_2, ..., w_l, y$ in G. Since $h(T) = k$ and $h(T') = k$ it follows that each component B of $G - x$ satisfies $mh(B) \leq k - 1$ and that similarly each component C of $G - y$ satisfies $mh(C) \leq k - 1$. Let B_1 be the component of $G - x$ containing w_j (and y) and let C_1 be the component of $G - y$ containing w_j (and x). Then G is as in Figure 3.

Let D_1 be any component of $G - w_j$ such that $x \notin V(D_1)$. Since G is a tree it follows that $D_1 \subset B_1$. Since $mh(B_1) \leq k - 1$ we see from Lemma 5.1 that $mh(D_1) \leq k - 1$. Let D_2 be the component of $G - w_j$ such that $x \in V(D_2)$. Then $D_2 \subset C_1$ and it follows from Lemma 5.1 that $mh(D_2) \leq k - 1$. Thus each component D of $G - w_j$ satisfies $mh(D) \leq k - 1$. An etree T_j, where $root(T_j) = w_j$ and where each subtree hanging from w_j in T_j is of minimum height, will then have $h(T_j) = k$. □

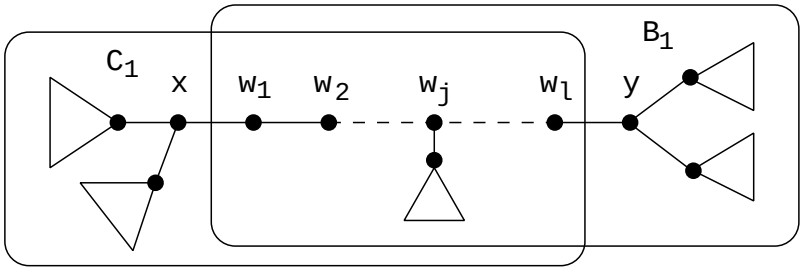

Figure 3:

The following corollary is a direct consequence of Lemma 6.1:

Corollary 6.2 *The set of nodes that can be chosen as the root of an etree of minimum height for G, form a connected component in G.* □

We are now in a position to define the node that will be the root of a tilted etree.

Definition 6.3 ($c_G(x)$) *For a given node $x \in V(G)$, let $c_G(x)$ be a node in $V(G)$ satisfying the following two conditions:*

1. *There exists an etree T for G such that $h(T) = mh(G)$ and $root(T) = c_G(x)$.*
2. *Over all nodes in $V(G)$ satisfying condition 1, $dist(c_G(x), x)$ is minimum.*

The node $c_G(x)$ is thus a closest node to x in G that can be chosen as the root of an etree of minimum height. Note that if there is an etree T of minimum height such that $root(T) = x$ then $c_G(x) = x$. Given x, we now show that there is exactly one node in $V(G)$ satisfying the conditions of Definition 6.3.

Lemma 6.4 *Given x, the node $c_G(x)$ exists and is unique.*

Proof: Let S be the set of nodes that can be the root of an etree of minimum height for G. Since there exists an etree of minimum height, S is nonempty. Assume that there exists at least two distinct nodes $y, z \in S$ such that among the nodes in S, they are both of minimum distance from x in G. Then there exist unique paths $y, v_1, v_2, ..., v_l = x$ and $z, w_1, w_2, ..., w_l = x$ in G such that $v_i, w_i \notin S$, $1 \leq i \leq l$. Let $u = \min\{i \mid v_i = w_i\}$. Note that u exists since $w_l = v_l = x$. From Corollary 6.2 we know that there exists a path from y to z in G consisting entirely of nodes in S. This path together with the path from y to u and the path from u to z give a cycle in G, contradicting the fact that G is a tree. It follows that there exists exactly one node in S of minimum distance from x in G. □

Now we are ready to define a tilted etree.

Definition 6.5 (Tilted etree) *Let T be an etree for G and let x be any node in $V(G)$. Then T is tilted towards x if the following two conditions are satisfied:*

1. *The node $c_G(x)$ is the root of T.*
2. *Each component B of $G - root(T)$ is ordered so that its elimination subtree T_B of T, is tilted towards the node $z \in V(B)$ such that $dist(z, x)$ is minimum over all nodes in $V(B)$.*

Note that if $x \in V(B)$ in Definition 6.5 then $z = x$, and if $x \notin V(B)$ then z is the node in $V(B)$ such that $(z, c_G(x)) \in E(G)$. Just as we did for $c_G(x)$, we now show that the etree tilted towards x is well defined.

Lemma 6.6 *Let x be a node in $V(G)$. Then there exists a unique etree T that is tilted towards x.*

Proof: The proof is by induction on the number of nodes in G. The result is trivial if $|V(G)| = 1$. Assume that the result is true for all trees with less than n nodes and that $|V(G)| = n > 1$. From Lemma 6.4 we know that $c_G(x)$ exists and is unique. Let B be a component of $G - c_G(x)$. Because G is a tree there exist a unique node $z \in V(B)$ such that $dist(z, x)$ is minimum over all nodes in B. Since $|V(B)| < n$ it follows from the induction hypothesis that there exists a unique etree T_B for B that is tilted towards z. Thus both $c_G(x)$ and T_B exists and are unique and the result follows. □

As promised we now show that a tilted etree is of minimum height.

Lemma 6.7 *Let x be a node in $V(G)$. Then the etree T tilted towards x is of minimum height.*

Proof: The proof is by induction on the number of nodes in G. The result is trivial if $|V(G)| = 1$. Assume that the result is true for all trees with less than n nodes and that $|V(G)| = n > 1$. Let T be an etree for G that is tilted towards x and let T' be an etree of minimum height where $root(T') = c_G(x)$. Let B be a component of $G - c_G(x)$ and let T_B be its subtree of T and T'_B its subtree of T'. Since $|V(B)| < n$ it follows from the induction hypothesis that $h(T_B) = mh(B)$. This implies that $h(T_B) \leq h(T'_B)$ and that $h(T) \leq h(T')$. Since $h(T') = mh(G)$ we must have $h(T) = mh(G)$. □

It should be made clear that the converse of Lemma 6.7 is not true in general: $h(T) = mh(G)$ does not imply that T is tilted towards some node.

7 Merging tilted elimination trees

In this section, we will show that Min_Height returns an etree of minimum height. This is done by first showing that the procedure Tilt presented in Section 4 produces a tilted etree if it is given a *semi-tilted* etree. Then we show that the etrees that Min_Height passes on to Tilt are semi-tilted. A semi-tilted etree is a relaxed version of a tilted etree. If x is a node in $V(G)$ then in a semi-tilted etree we have x as the root instead of $c_G(x)$. This is the only difference between a semi-tilted etree and an etree tilted towards x. The formal definition is as follows:

Definition 7.1 (Semi-tilted) *An elimination tree T is semi-tilted if each component B of $G - root(T)$ is ordered so that its elimination subtree T_B of T is tilted towards the unique node $u \in V(B)$ such that $(u, root(T)) \in E(G)$.*

If we remove the root x of a semi-tilted etree T we get a number of tilted subtrees. As we shall see, the effect of Tilt(x) is that the tilted subtrees hanging from x in T are merged together with x into a new etree that is tilted towards x. From Corollary 5.2 we know that if T is semi-tilted then either $h(T) = mh(G)$ or $h(T) = mh(G) + 1$.

In Lemmas 7.2 through 7.7, we will show a number of properties of semi-tilted etrees. These properties will be used to show that Tilt produces a tilted etree if it is given a semi-tilted etree. In each lemma we assume that T is a semi-tilted etree of height $k > 0$ and that x is a highest child of $v = root(T)$ in T. If v has more than one child in T let y be a second highest child of v in T. Thus $h(T[x]) = k - 1$ and if

y exists $h(T[y]) \leq k-1$ (otherwise assume $h(y) = -1$). Also let T' be the etree after performing Rotate(x).

The following names will be used for different components of G. Let B be the component of $G - x$ containing v. Let C be the component of $G - v$ containing x. Let $D = bw(x, v)$. If $(x, v) \notin E(G)$ let $v, w_1, w_2, ..., w_l, x$ be the nodes on the path from v to x in G. Then G is as in Figure 4.

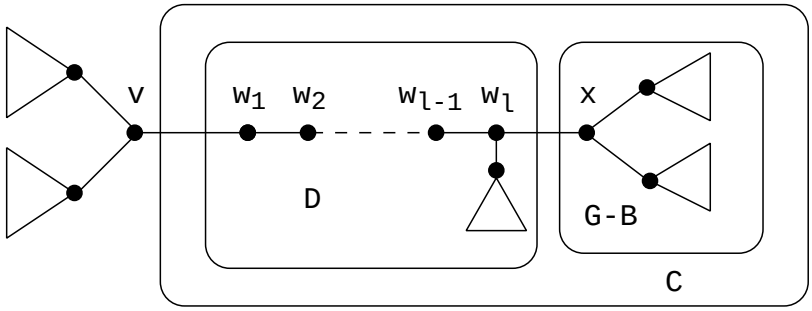

Figure 4:

Lemma 7.2 *If $h(x) = h(y)$ in T then $mh(G) = k$.*

Proof: From Definition 7.1 the etrees $T[x]$ and $T[y]$ are both tilted. From Lemma 6.7 we know that $T[x]$ and $T[y]$ are of minimum height. From Corollary 5.3 it then follows that if $h(T[x]) = h(T[y])$ then $mh(G) = k$. □

Lemma 7.3 *If $mh(G) = k$ then T is tilted towards v.*

Proof: Note first that $mh(G) = k$ implies that T is of minimum height. Since v is the root of T it follows that $c_G(v) = v$. Let F be a component of $G - v$ and let T_F be its elimination subtree of T. Since T is semi-tilted, T_F is tilted towards the node $z \in V(F)$ such that $(z, v) \in E(G)$. Since G is a tree it follows that z is the closest node to v in G among the nodes in $V(F)$. Thus T_F satisfies condition 2 of Definition 6.5. This shows that if $mh(G) = k$ then T is tilted towards v. □

Lemma 7.4 $mh(G - B) = k - 1$.

Proof: The etree $T[x]$ is tilted towards the node u such that $(u, v) \in E(G)$. Thus $T[x]$ is of minimum height $k - 1$. If $x = u$ (i.e., $D = \emptyset$), the result then follows immediately from the fact that $V(T[x]) = G - B$. Suppose therefore that $x \neq u$. Then $u = w_1$. Since $V(T[x]) = V(C)$ and $D \subset (C - \{x\})$ we must have $mh(D) \leq k - 2$. Note

that $dist(w_1, w_l) < dist(w_1, x)$ and that $c_C(w_1) = x$. This implies that there does not exist an etree for C of height $k-1$ with w_l as its root. This again implies that some component F of $C - w_l$ must have $mh(F) = k-1$. Let H be any component of $C - w_l$ such that $x \notin V(H)$. Then since $H \subset D$ and $mh(D) < k-1$ we see from Lemma 5.1 that $mh(H) < k-1$ and it follows that $H \neq F$. Since $G - B$ is the component of $C - w_l$ containing x, we must have $F = G - B$ and $mh(G - B) = k-1$. □

Lemma 7.5 *If $mh(G) = k-1$ then $c_G(v) = x$.*

Proof: Note that x is the closest node in $V(G - B)$ to v. First we will show that no node in $V(B)$ can be the root of an etree for G of height $k-1$. Then we will show that if there exists an etree of height $k-1$ with a node from $V(G - B)$ as root, then there must also exist an etree of height $k-1$ with x as its root. From this it will follow that if $mh(G) = k-1$ then $c_G(v) = x$.

Let T_1 be an etree for G where $z = root(T_1)$ and $z \in V(B)$. Then there exists a component F of $G - z$ such that $(G - B) \subseteq F$. From Lemma 7.4 we know that $mh(G - B) = k-1$. Thus from Lemma 5.1 $mh(F) \geq k-1$. This implies that $h(T_1) \geq k$ and that no node in $V(B)$ can be the root of an etree of height $k-1$.

Suppose now that there exists an etree T_2 of height $k-1$ where $z = root(T_2)$ and $z \in V(G - B)$. Then each component F of $G - z$ satisfies $mh(F) \leq k-2$. Let H be the component of $G - z$ containing v. Since $B \subseteq H$ it follows from Lemma 5.1 that we must also have $mh(B) \leq k-2$. Since $h(T[x]) = k-1$ each component I of $(G-B) - x$ has $mh(I) \leq k-2$. Now we have shown that each component of $G - x$ has an etree of height $\leq k-2$. It follows that there exists an etree for G of height $k-1$ with x as its root.

Thus we can conclude that if $mh(G) = k-1$ then $c_G(v) = x$. □

Lemma 7.6 *The etree $T'[v]$ is semi-tilted.*

Proof: We must show that each subtree $T'[u]$ where $p(u) = v$ in T', is tilted towards the node s in $V(T'[u])$ such that $(s, v) \in E(G)$. Let $T'[u]$ be a subtree of T' such that $p(u) = v$ in T' and $V(T'[u]) \neq V(D)$. Then $T'[u] = T[u]$ and $p(u) = v$ in T. Since T is semi-tilted, the etree $T'[u]$ is tilted towards the node $s \in V(T'[u])$ such that $(s, v) \in E(G)$.

If D is nonempty the nodes in D form a subtree $T[z]$ of T and a subtree $T'[z]$ of T' such that $T[z] = T'[z]$. Moreover, $p(z) = x$ in T and $p(z) = v$ in T'. (See Figure 1 a and 1 b.) The node z is the only child of v in T' that was not a child of v in T. It

remains to show that $T'[z]$ (and thus $T[z]$) is tilted towards the node $w_1 \in V(D)$. The subtree $T[x]$ is tilted towards w_1. Since $w_1 \in V(D)$ it follows from Definition 6.5 that $T[z]$ (and $T'[z]$) is also tilted towards w_1. Thus $T'[v]$ is semi-tilted. □

Lemma 7.7 *$mh(G) = k-1$ if and only if $mh(T'[v]) \leq k-2$.*

Proof: ($\Rightarrow$) From Lemma 7.5 we know that if $mh(G) = k-1$ then $c_G(v) = x$. Thus if $mh(G) = k-1$ each component F of $G - x$ satisfies $mh(F) \leq k-2$. The result follows since $T'[v]$ consists of all the nodes in B and B is the component of $G - x$ that contains v.

($\Leftarrow$) From Lemma 7.5 we know that if $mh(G) = k-1$ then there exists an etree of height $k-1$ with x as its root. Since $h(T[x]) = k-1$, each child $z \neq v$ of x in T' satisfies $h(T'[z]) \leq k-2$. Thus $mh(G) = k-1$ if $mh(T'[v]) \leq k-2$. □

Now we are in a position to show the following lemma on the effect of Tilt on a semi-tilted etree.

Lemma 7.8 *Let T be an etree with $v = root(T)$. If T is semi-tilted then Tilt(v) will reorder T into an etree that is tilted towards v.*

Proof: The proof is by induction on the height of T. Let x be a highest child of v in T and y a second highest child of v in T ($h(y) = -1$ if v only has one child). As discussed in Section 4, Tilt can be implemented as a recursive procedure that performs Rotate(x) if $h(v) > 0$ and $h(x) > h(y)$ and then calls itself with v as argument. Depending on the outcome of this call the etree is either left unchanged or T is restored by a call to Rotate(v).

The induction hypothesis is: If T is semi-tilted and $h(T) < k$ then Tilt(v) returns an etree tilted towards v.

If $h(T) = 0$ then Tilt leaves T unchanged. It is easy to see that T is tilted towards v in this case.

Assume now that the induction hypothesis is true and that $h(T) = k > 0$. If $h(T[x]) = h(T[y])$ then Tilt exits without altering T. From Lemma 7.2 it follows that T is of minimum height and from Lemma 7.3 it follows that T is tilted towards x. The induction hypothesis is thus proved true in this case.

Assume that $h(T[x]) > h(T[y])$. Tilt then performs a rotation with respect to x to get a new etree T'. Since x is the only child of v of height $k-1$ in T we have $h(T'[v]) < k$. From Lemma 7.6 it follows that $T'[v]$ is also semi-tilted. From the induction hypothesis

we know that Tilt will find an etree for $T'[v]$ that is tilted towards v, and therefore also of minimum height. Let T'' be the new etree after reordering $T'[v]$ into an etree that is tilted towards v and let r be the child of x in T'' such that $v \in V(T''[r])$. Now we look at two different cases depending on the height of $T''[r]$.

First we consider the case when $h(T''[r]) = k - 1$. In this case Tilt will perform Rotate(r) and exit. From Lemma 7.7 we know that $h(T''[r]) = k - 1$ implies that $mh(G) = k$. Since $mh(G) = k$ we see from Lemma 7.3 that T is tilted towards v. Thus we must show that Rotate(r) restores T. Since $mh(G) = k$ we must have $h(T'[v]) = k - 1$ or else $h(T') < k$. Thus $T'[v]$ is of minimum height. From Lemma 7.6 we know that $T'[v]$ is semi-tilted. Since $T'[v]$ is of minimum height and semi-tilted it follows from Lemma 7.3 that $T'[v]$ is tilted towards v. Thus $T'[v] = T''[r]$ and by performing Rotate(r) Tilt restores T.

Assume now that $h(T''[r]) < k - 1$. Then Tilt exits without altering T''. We must show that T'' is tilted towards v. From Lemma 7.7 it follows that if $h(T''[r]) < k - 1$ then $mh(G) = k - 1$. From Lemma 7.5 we know that $c_G(v) = x$. Since $root(T'') = x$ it remains to show that each subtree hanging from x in T'' is tilted towards the node z that is closest to v in G. We already know from the induction hypothesis that $T''[r]$ is tilted towards v. Let s be a child of x in T'' other than r. Then $T[s] = T''[s]$ and $p(s) = x$ both in T and in T''. The etree $T[x]$ is tilted towards the node $q \in V(T[x])$ such that $(q, v) \in E(G)$. If $q \neq x$ then $q \in V(T'[v])$ and $q \in V(T''[r])$. Thus $T''[s]$ (and $T[s]$) is tilted towards the node $z \in V(T''[s])$ such that $(z, x) \in E(G)$. Since G is a tree, z is the closest node to x, q and v in G among the nodes in $V(T''[s])$. We have now shown that T'' satisfies each condition in Definition 6.5 and is therefore tilted towards v. □

Now we can show the main result of this section.

Theorem 7.9 *Min_Height(G) returns an etree of minimum height for G.*

Proof: Min_Height initially constructs an etree T such that for each node $x \neq root(T)$ we have $(x, p(x)) \in E(G)$. The nodes are then visited in postorder and for each node v the procedure Tilt(v) is called. We now prove by induction on the height of v in T that the etree rooted at v is semi-tilted when Tilt(v) is called.

If $h(v) = 0$ then $T[v]$ is clearly semi-tilted. Assume that the induction hypothesis is true for each node of height $< k$ in T. Let v be a node of height k in T, $k > 0$. Each child x of v in T is of height $< k$ in T and is visited before v. From the induction

hypothesis and Lemma 7.8 we know that when v is visited $T[x]$ has been reordered into an etree that is tilted towards x. Since $(x, v) \in E(G)$ it follows that v is the root of a semi-tilted etree $T'[v]$ before Tilt(v) is called. Thus Tilt(v) reorders $T'[v]$ into an etree that is tilted towards v and the induction hypothesis is proved true.

The last call to Tilt is done with argument $v = root(T)$. The node v is the root of a semi-tilted etree containing each node in $V(G)$. Since the induction hypothesis is proven true it follows that the resulting etree is tilted towards v and therefore also of minimum height. □

8 Some details of the implementation

In this section we will look at some of the details of the procedure Tilt as presented in Section 4. Specifically we will show how to implement the two functions $s()$ and $h()$. This must be done before we can give an analysis of the time complexity of Min_Height. The reason for giving these details here rather than in Section 4 is that the implementation of the function $h()$ depends on the fact that Min_Height returns an etree of minimum height.

First we look at $s()$. Let T be the current etree at the time of calling Tilt(v). Let x be a highest child of v in T. If x is the only child of v in T of height $h(T[v]) - 1$ then Tilt will perform Rotate(x). If $bw(x, v) \neq \emptyset$ then prior to performing Rotate(x), $s(x)$ must point to the root of the subtree of T consisting of the nodes in $bw(x, v)$. Since $T[v]$ is semi-tilted this is achieved if $s(x)$ points to the child z of x in T such that z is an ancestor of the node that $T[x]$ is tilted towards. If $bw(x, v) = \emptyset$ then $s(x)$ must point to x. We assume that this is true for each node in $T[v]$ except v, at the time of calling Tilt(v). The code we will give here ensures that this will be true for each node originally in $T[v]$ when Tilt exits.

The following code is added right after Rotate(x) inside the first while loop of Tilt:

$s(x) := v;$
$s(p(x)) := x;$

Note that $s(p(x)) = v$ is true before each but the first rotation in the first while loop. After the execution of the first while loop, $s()$ will be correct for each node originally in $T[v]$ except v. In the second while loop of Tilt we might want to reverse some of the rotations performed in the first while loop. This is done by executing Rotate(v) a number of times. Let $x = p(v)$ prior to such a rotation. Also let z be the root of the

subtree of the current etree consisting of the nodes in $bw(x, v)$. To be able to locate z we use a temporary pointer $t()$ associated with x. Prior to Rotate(x) in the first while loop we add:

$t(x) := s(x)$;

Then if $bw(x, v) \neq \emptyset$, $t(x)$ will point to z before executing Rotate(v), and to x if $bw(x, v) = \emptyset$. We also rewrite the body of the second while loop as follows:

$x := p(v)$;
If $t(x) = x$
  $s(v) := v$
else
  $s(v) := t(x)$;
Rotate(v);
$s(x) := t(x)$;
$s(p(x)) := v$;

This ensures that $s()$ will be correct for each node in $T[v]$ except v when the second while loop is done. When Tilt exits, the etree rooted at v is tilted towards v. We therefore add:

$s(v) := v$;

after the second while loop. Then if $s()$ is correct for each node in $T[v]$ except v when Tilt(v) is called, $s()$ will be correct for each node originally in $T[v]$ when Tilt exits. To initialize $s()$ we set $s(x) = x$ in Min_Height for each node such that $h(x) = 0$ in the original etree.

We now discuss how to maintain the height of each node as Tilt reorders a semi-tilted etree into a tilted etree. Let $ch(v)$ be a function that returns $\max\{h(x) + 1 \mid p(x) = v\}$.

Note first that if we initially know the height of each node in $T[v]$ then the first while loop in Tilt will function correctly even if we do not update the heights of v and $p(v)$ after each rotation. However, we must use a test other than $h(v) > 0$ in order to determine whether v is a leaf in the etree. This can be done by testing whether v has any children in the etree. If we do not update any heights in the first while loop, the only heights that might be incorrect at the start of the second while loop are those of v and its ancestors. For the second while loop to function correctly we only need to keep the height of v updated. To do this we rewrite the second while loop as follows:

$h(v) := ch(v)$;
$h(p(v)) := ch(p(v))$;

While ($|S| > 0$) **and** ($h(v) =$ **pop**(S)-1)
    $x := p(v)$;
    Rotate(v);
    $h(x) := ch(x)$;
    $h(v) := ch(v)$;
    $h(p(v)) := ch(p(v))$;
End While

Note that this does not include the extra code to maintain $s()$. After executing Tilt the only heights that might be incorrect are those of the ancestors of $p(v)$. To assure that these are correct we add the following code in the first while loop after Rotate(x):

$h(v) := ch(v) - 1$;
$h(x) := ch(x)$;

To see that this is sufficient for computing the correct height of each node originally in $T[v]$ we need the following lemma.

Lemma 8.1 *The height of each node initially in $T[v]$ will be correct when Tilt exits.*

Proof: As discussed we need to show only that the height of the ancestors of $p(v)$ are correct when Tilt exits. Let x be a node originally in $T[v]$ that is an ancestor of $p(v)$ when Tilt exits. At some stage of the first while loop Rotate(x) must have been performed. Let T' be the resulting etree from this rotation. The height of x is now computed under the assumption that $h(T'[v])$ is one lower than what it actually is. Let B be the component of G consisting of the nodes in $V(T'[v])$. Since x is an ancestor of $p(v)$ when Tilt exits we must have been able to find an etree T''_B for B such that $h(T''_B) < h(T'[v])$. Since $T'[v]$ is semi-tilted it follows from Corollary 5.2 that $h(T''_B) = h(T'[v]) - 1$. Thus it was correct to assume that $h(T'[v])$ was one lower than it actually was at the time of computing $h(x)$. □

From the above discussion we see that if we can find $ch()$ in constant time then keeping track of the height of each node will not increase the asymptotic running time of Tilt. To be able to implement $ch()$ efficiently we assume that each node y has its current children in the etree, in a heap ordered by decreasing height. The heap is a

standard heap as described in [25]. The heap is updated after a rotation that involves y is performed and the new height of y has been computed. If $v = p(x)$ and $u = p(v)$ then after performing Rotate(x) we must update the heaps of x, v and u. Since the height of v and $p(v)$ are recomputed prior to the second while loop in Tilt we also update the heap of $p(v)$ just before the second while loop.

9 Time complexity

In this section, we give an upper bound on the time complexity of Min_Height. We start by giving a time bound on Tilt. Let d be the maximum vertex degree of a node in G.

Lemma 9.1 *Let T be the etree at the time of calling Tilt(v). Then the time complexity of Tilt(v) is $O(h(T[v]) \log d)$.*

Proof: In the first while loop of Tilt a number of rotations are performed on the current highest child of v until either v is a leaf or v has at least two children of maximal height. In each rotation the height of v is reduced by at least one. Thus we perform at most $h(T[v])$ rotations in the first while loop. In the second while loop we perform Rotate(v) until either v is not the only child of $p(v)$ of maximum height or until $T[v]$ is restored. Each rotation reverses a rotation performed in the first while loop. Thus we do not perform more than $2h(T[v])$ rotations in each call to Tilt. After each rotation three heaps must be updated. A heap update is also performed before the second while loop. Thus we do not perform more than $6h(T[v]) + 1$ heap updates. From Lemma 3.4 we know that no node has more than d children in the etree. Thus it follows from [25] that each update of the heap can be done in $O(\log d)$ time giving a total time of $O(h(T[v]) \log d)$ for Tilt. □

Now we can show the main result of this section.

Theorem 9.2 *The time complexity of Min_Height(G) is $O(n \log n \log d)$*

Proof: The initial ordering of G and construction of T can be done in linear time. When Tilt(v) is called the etree rooted at v is semi-tilted. This implies that $h(T[v]) \leq \lfloor \log n \rfloor + 1$. The result now follows from Lemma 9.1 since Tilt is called once in Min_Height for each node in G. □

It is also clear from the code in Section 4 and Section 8 that Min_Height does not require more space than a constant times the number of edges and nodes in G. Thus Min_Height requires $O(n)$ space.

10 Fill-in

In this section, we will show that if G is ordered so that its etree T is tilted then the number of fill edges in G^* is at most $n - h(T) - 1$. Note first the following lemma from [17] concerning fill edges.

Lemma 10.1 *Let H be a general graph with elimination tree T and let x be a node in $V(H)$. Then there is a fill edge between a node $y \in T[x]$ and x if and only if $(x, y) \notin E(H)$ and there exists a node $z \in T[y]$ such that $(z, x) \in E(H)$.* □

Let T be the etree of G and let $v, x \in V(G)$ such that $p(x) = v$ in T. Then since G is a tree there is exactly one node z in $T[x]$ such that $(z, v) \in E(G)$. From Lemma 10.1 it follows that there is a fill edge between each proper ancestor of z in $T[x]$ and v, and that no other node in $T[x]$ has a fill edge to v. Thus if z has k proper ancestors in $T[x]$ then there are exactly k fill edges between v and the nodes in $T[x]$. Note that $k \leq h(T[x])$.

We now define a partitioning of the nodes in a tilted etree into paths in the etree. This partitioning will be used to give a bound on the number of fill edges if G is ordered so that its etree is tilted. For the rest of this section we assume that T is tilted (towards some node).

Definition 10.2 (t-path) *Let $T[u]$ be a subtree of T such that $T[u]$ is tilted towards u. Let further $u = w_0, w_1, ..., w_l$ be a maximal path in T such that $p(w_i) = w_{i+1}$, $0 \leq i < l$, and $T[w_i]$ is tilted towards u for $0 \leq i \leq l$. Then $w_0, w_1, ..., w_l$ is a t-path in T.*

To see that the t-paths actually partitions $V(T)$ we need the following lemma:

Lemma 10.3 *Each node in $V(T)$ is on exactly one t-path.*

Proof: Let x be a node in $V(T)$. Then $T[x]$ is tilted towards some node $u \in V(T[x])$. This implies that x can be on at most one t-path. We will show that $T[w_i]$ is tilted towards u, for each node w_i on the path $u = w_0, w_1, ..., w_l = x$ in T. This will then imply that x is on a t-path. The proof is by induction on the distance of each w_i from

x in T. We already know that $T[x]$ is tilted towards u. Assume that $T[w_i]$ is tilted towards u for some $i > 0$. Let B be the component of G containing the nodes in $T[w_i]$. Then the nodes in $T[w_{i-1}]$ form a connected component C of $B - w_i$ containing u. Since $u \in V(T[w_{i-1}])$ and $dist(u, u) = 0$ it follows from Definition 6.5 that $T[w_{i-1}]$ is tilted towards u. Thus the induction hypothesis is proved true and the result follows. □

Let $T[v]$ be a subtree that is tilted towards u and let y be a child of v in T that is not an ancestor of u. Then from Definition 6.5, $T[y]$ is tilted towards the node z in $V(T[y])$ that is closest to u in G. Since G is a tree it follows that $(z, v) \in E(G)$. Since $T[y]$ and $T[v]$ are tilted towards two different nodes, y must be the topmost node of a t-path $z = w_0, w_1, ..., w_l = y$. Thus there is a fill edge from v to each w_i, $1 \leq i \leq l$, and to no other node in $V(T[y])$.

We now show that the each non-leaf node in T has a child of maximum height that is the topmost node of some t-path.

Lemma 10.4 *Let w_i be a node on a t-path $w_0, w_1, ..., w_l$ in T, where $h(w_i) = k > 0$. Then w_i has a child y in T such that $h(T[y]) = k - 1$ and y is the topmost node of a t-path.*

Proof: If $i = 0$ the result is obviously true since each child of w_0 is the topmost node in a t-path. Assume therefore that $i > 0$. Let B be the component of G that contains the nodes in $T[w_i]$. Let C be the component of $B - w_i$ that contains w_0 and let $D = B - C$. Then from Lemma 7.4, $mh(D) = k > 0$. Thus there must exists a nonempty component E of $D - w_i$ such that $mh(E) = k - 1$. In T, the nodes in E consist of a subtree hanging from w_i of height $k - 1$. Let y be the root of this subtree. Then since $y \neq w_{i-1}$ the node y is the topmost node of a t-path. □

We can now give a bound on the number of fill edges between the nodes in any subtree of T. For ease of notation we will denote $|V(T[v])|$ by $f(v)$.

Lemma 10.5 *Let w_i be a node on a t-path $w_0, w_1, ..., w_l$ in T. Then there are at most*

$$f(w_i) - h(w_i) - (i + 1) \tag{1}$$

fill edges between the nodes in $T[w_i]$.

Proof: The proof is by induction on the height of w_i. If $h(w_i) = 0$ then $i = 0$ and $f(w_i) = 1$. Substituting this into (1) there can be no fill edges between the nodes in $T[w_i]$ if the induction hypothesis is to hold. But since $T[w_0]$ consists of a single node when $h(w_0) = 0$ this is obviously true. Assume that the lemma is true for all nodes x where $0 \leq h(x) < k$, and let $h(w_i) = k$. From Lemma 10.4 it follows that w_i has a child x such that $h(x) = k - 1$ and x is the topmost node of a t-path $u_0, u_1, ..., u_r = x$ in T. From the induction hypothesis we know that there are at most $f(x) - (k-1) - (r+1)$ fill edges between the nodes in $T[x]$. There are r fill edges between w_i and the nodes in $T[x]$ giving a total of no more than

$$f(x) - k \tag{2}$$

fill edges involving w_i and the nodes in $T[x]$. Let $y_1, y_2, ..., y_m$ be children of w_i in T such that for $1 \leq j \leq m$, $y_j \neq x$ and if $i > 0$ then $y_j \neq w_{i-1}$. Then each y_j is the topmost node of a t-path. From the same argument used on $T[x]$, we see that there are at most

$$f(y_j) - h(y_j) - 1 < f(y_j) \tag{3}$$

fill edges involving w_i and the nodes in $T[y_j]$. If $i > 0$ then w_{i-1} is also a child of w_i in T. From the induction hypothesis there are at most $f(w_{i-1}) - h(w_{i-1}) - i$ fill edges among the nodes in $T[w_{i-1}]$. There are no more than $h(w_{i-1})$ fill edges between w_i and the nodes in $T[w_{i-1}]$. Thus there are at most

$$f(w_{i-1}) - i \tag{4}$$

fill edges involving w_i and nodes in $T[w_{i-1}]$.

If $i = 0$ then $f(x) + \sum_{j=1}^{m} f(y_j) = f(w_0) - 1$. Combining this with (2) and (3) we see that there are no more than $f(w_0) - k - 1$ fill edges in $T[w_0]$. Substituting $i = 0$ in (1) we see that the induction hypothesis is true in this case.

If $i > 0$ then $f(x) + f(w_{i-1}) + \sum_{j=1}^{m} f(y_j) = f(w_i) - 1$. From (2), (3) and (4) it follows that there can at most be $f(w_i) - k - (i+1)$ fill edges in $T[w_i]$. This is the same as (1) and the induction hypothesis is proved true. □

Now we can give an upper bound on the number of fill edges in T.

Theorem 10.6 *If T is tilted towards some node x then G^* contains at most $n-h(T)-1$ fill edges.*

Proof Let $w_0, w_1, ..., w_k$ be the t-path in T such that $w_k = root(T)$. The result follows from Lemma 10.5 since $f(root(T)) = n$ and $n - h(T) - (k+1) \leq n - h(T) - 1$. □

Since $h(T) \geq 0$ it follows from Theorem 10.6 that there can never be more than $n-1$ fill edges in G^*. To see that the bound from Theorem 10.6 is sharp consider a graph G on $n = 2^k$ nodes on a single path, where $v \in V(G)$ is a node of degree one. If T is tilted towards v then $h(T) = k$ and there are $n-k-1$ fill edges in G^*. The proof of this is left as an exercise for the reader.

11 Other applications

In this section, we consider how the results in this paper might help us find low etrees for more general classes of graphs. We consider the problem of computing an etree of minimum height for a chordal graph. But first we mention a possible generalization of rotations to general graphs.

One way in which rotations can be generalized to etrees for general graphs is to perform rotations on maximal chains in the etree instead of on single nodes. Let T be an etree for a general graph G and let $K = \{k_1, k_2, ..., k_s\}$ be the uppermost maximal chain in T and $L = \{l_1, l_2, ..., l_t\}$ a maximal chain so that $p(l_t) = k_1$ in T. Without loss of generality we can assume that L and K are ordered consecutively. A generalized rotation consists of letting the nodes in K and L switch places in the elimination ordering while maintaining the internal order among the nodes in K and L.

The effect of a rotation is more complicated for a general graph than for a tree. Let T' be the etree after performing the rotation. Then the structure of each subtree of T consisting of nodes in $G - K - L$ will remain unchanged. The structure of L will also remain unchanged except that $l_t = root(T')$. The only structural differences between T and T' apart from $p(l_t)$, involve K and $p(x)$ where x is a child of l_1 in T. Without going into the details we note that it is possible that K will break into more than one connected component. Each such component will hang from some l_j in T'. It is also possible that some children of l_1 in T become children of nodes in K in T'. In terms of the height of T' it is favorable if K breaks up and if the children of l_1 in T remain as children of l_1 in T'. The worst possible scenario in terms of the height of T' is when K is a chain that hangs from l_1 in T'. Let x be the highest child of l_1 in T and y the

highest child of k_1 in T. Then if $h(T[y]) \leq h(T[x])$ we get $h(T') \leq h(T)$ even if K is connected in T'.

Whether generalized rotations can be used in a practical algorithm for reducing the height of an etree for a general graph needs further study.

We now consider whether it is possible to compute an etree of minimum height for a chordal graph based on the results presented in this paper. There are three key reasons why it is possible to compute an etree of minimum height for a tree efficiently. The first is that in a tree G there are a limited number of easily identifiable separators. The second is that each connected subgraph of G is also a tree. The third is that if for some node $v \in V(G)$ we know $mh(B)$ for each component B of $G - v$, then we can decide in which component of $G - v$ to look for the root of an etree of minimum height. The introduction of tiltedness improves the efficiency of this search.

We investigate how well these properties extend to chordal graphs. From Theorem 3.3 we know that there for any graph exists a minimal separator ordering giving an etree of minimum height. From [22] it is known that every minimal separator K of a chordal graph G is a clique and that every connected component of $G - K$ is chordal. Thus we only have to consider separators that are cliques and every subgraph that needs to be considered will also be chordal. There are at most $n-1$ minimal separators in a chordal graph [4]. By the use of *clique trees* these can be identified in linear time [14]. From this we see that the first two conditions extend to chordal graphs. However, as shown below the third condition does not necessarily extend.

Let K be a minimal separator of the chordal graph G and let B be a component of $G - K$ such that $mh(B)$ is maximal over all components of $G - K$. There exists then an etree T for G of height $|K| + mh(B)$. From this one might be lead to believe that the top separator for an etree of minimum height for G must be found among the nodes in B and K. However, this is not true in general. If a minimal separator $C \not\subseteq \{K, B\}$ is used as the topmost separator then some component D of $G - C$ will contain B as a subgraph. But if $|C| < |K|$ then the only lower bound one can give on the height of an etree with C as the topmost separator is $|C| + mh(B)$. If $|C| < |K|$ it is possible that an etree with C as the topmost separator can have height $< |K| + mh(B)$. For the same reason one cannot prove the obvious analogue of Corollary 5.3 one might hope would hold true. Thus to find an etree of minimum height for G we must not only search K and B for the topmost separator, but we must also consider separators for G not in K and B, that are of smaller size than K.

These observations indicate that the problem of computing an etree of minimum

height for a chordal graph cannot be solved by a simple extension of the algorithm presented in this paper. More insight is needed before one can determine whether an efficient algorithm for this problem can be found.

We note that the recent development of a method [1] which computes an etree of minimum height for interval graphs. This points to the possibility that there are other classes of graphs for which the minimum height etree problem can be solved efficiently.

12 Conclusion

In this paper, we presented an algorithm for computing an etree of low height for a graph that is a tree. It uses local rotations in order to reduce the height of the etree. We then defined and analyzed a theory for a special kind of etrees of minimum height called tilted etrees. Through a series of lemmas we were able to show that our main algorithm returns a tilted etree. We showed that the running time of the algorithm is $O(n \log n \log d)$ and that it introduced at most $n - 1$ fill edges. We also pointed to a possible generalization of rotations to general graphs and discussed the possibility of being able to compute etrees of minimum height for more general graphs than trees.

As to the efficiency of the Min_Height algorithm we note that in [18] a nested dissection algorithm for trees was presented which computes an etree of height no more than $\lfloor \log n \rfloor$. This algorithm also had time complexity $O(n \log n \log d)$.

Gilbert has conjectured [7] that any graph has an etree of minimum height such that the number of fill edges is at most a constant times the number of edges in G^* where G is ordered by a minimum fill ordering. This paper shows that the conjecture is true for trees.

References

[1] B. Aspvall and P. Heggernes, *Finding minimum height elimination trees for interval graphs in polynomial time*, Tech. Report CS-93-80, Department of Informatics, University of Bergen, Norway, 1993.

[2] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks, *Approximating treewidth, pathwidth and minimum elimination tree height*, Tech. Report CSL-90-10, Xerox Palo Alto Research Center, 1991.

[3] C. A. Crane, *Linear lists and priority ques as balanced binary trees*, Tech. Report CS-72-259, Computer Science Dept., Stanford Univ., 1972.

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

[5] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[6] A. George and J. W. H. Liu, *An automatic nested dissection algorithm for irregular finite element problems*, SIAM J. Numer. Anal., 15 (1978), pp. 1053–1069.

[7] J. R. Gilbert, 1991. Personal comunication.

[8] M. T. Heath, E. Ng, and B. Peyton, *Parallel algorithms for sparse linear systems*, SIAM Rev., 33 (1991), pp. 420–460.

[9] P. Heggernes, *Minimizing fill-in size and elimination tree height in parallel Cholesky factorization*, master's thesis, Department of Informatics, University of Bergen, Norway, 1992.

[10] J. A. G. Jess and H. G. M. Kees, *A data structure for parallel L/U decomposition*, IEEE Trans. Comput., C-31 (1982), pp. 231–239.

[11] C. Jordan, *Sur les assemblages de lignes*, Journal Reine Angew. Math., 70 (1869), pp. 185–190.

[12] D. E. Knuth, *Sorting and Searching*, vol. 3 of The Art of Computer Programming, Addison-Wesley, 1973.

[13] C. E. Leiserson and J. G. Lewis, *Orderings for parallel sparse symmetric factorization*, in Parallel Processing for Scientific Computing, G. Rodrigue, ed., SIAM, 1989, pp. 27–32.

[14] J. G. Lewis, B. W. Peyton, and A. Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1146–1173.

[15] J. W. H. Liu, *Computational models and task scheduling for parallel sparse Cholesky factorization*, Parallel Comput., 3 (1986), pp. 327–342.

[16] ——, *Reordering sparse matrices for parallel elimination*, Parallel Comput., 11 (1989), pp. 73–91.

[17] ——, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[18] F. Manne, *Minimum height elimination trees for parallel Cholesky factorization*, master's thesis, Department of Informatics, University of Bergen, Norway, 1989. (In Norwegian).

[19] ——, *Reducing the height of an elimination tree through local reorderings*, Tech. Report CS-91-51, University of Bergen, Norway, 1991.

[20] S. Parter, *The use of linear graphs in Gauss elimination*, SIAM Rev., 3 (1961), pp. 119–130.

[21] A. Pothen, *The complexity of optimal elimination trees*, Tech. Report CS-88-13, Pennsylvania State University, 1988.

[22] D. J. Rose, *Triangulated graphs and the elimination process*, SIAM J. Matrix Anal. Appl., 32 (1970), pp. 597–609.

[23] D. J. Rose, R. E. Tarjan, and G. S. Leuker, *Algorithmic aspects and vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[24] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.

[25] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.

[26] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Discrete Meth., 2 (1981), pp. 77–79.