Reducing the fill-in size for an elimination tree

Pinar Heggernes
Department of Informatics
University of Bergen

April,1992

Abstract

In sparse Cholesky factorization, finding elimination orderings that produce small fill-in is very important and various heuristics have been proposed. For parallel computations, good orderings should produce elimination trees of low height. Finding optimal fill-in orderings and finding optimal height orderings are NP-hard problems. A class of Nested Dissection orderings with minimal separators, has been shown to contain optimal height orderings. We show that for such orderings, the problem of computing minimum fill-in decomposes into independent subproblems for each separator. These subproblems are shown to be NP-hard. However, for small separators, they can be solved optimally in reasonable time. We also introduce a new heuristic for these subproblems.

1 Introduction

In parallel Cholesky factorization, it is important to find orderings that result in both low fill-in and low elimination trees. (For detailed information about elimination trees, see Liu [2]). The usual approach to achieve this, has been to first find orderings that produce low fill-in, and then to reduce the height of the resulting elimination tree while preserving the low fill-in ordering. We suspect however, that doing things in the reverse order might result in better orderings. There are examples which show that orderings for low fill-in may result in elimination trees that are much higher than optimal (Heggernes [1]).

We will in this paper, look at how we can further reduce the fill-in size when a low elimination tree ordering is already given. We show how to reduce the fill-in size for a given elimination tree without changing its structure. Manne shows in [3] that there always exists a Nested Dissection ordering with minimal separators of a graph G that results in an elimination tree of minimum height. This type of Nested Dissection does not have any requirements on the size of the remaining components. Throughout this paper we will be using Nested

Dissection orderings that choose minimal separators in each step, and there are no requirements on the size of the remaining components. In the next section, we show that for such orderings, reducing the fill-in becomes a local problem for each minimal separator.

We now give a brief discussion on why we think it might be better to start with low elimination trees and try to reduce the fill-in size, instead of starting with low fill-in and trying to reduce the elimination tree height. If two vertices v_i and v_j of a graph G are numbered respectively i and j where $i > j$, then v_i might be an ancestor of v_j in the resulting elimination tree although $l_{ij} = 0$. This case is shown in Figure 1 a). Thus, low fill-in does not imply a low elimination tree. But if $i > j$ and v_i is not an ancestor of v_j in the elimination tree, as shown in Figure 1 b), then l_{ij} must be zero. Reducing the elimination tree height might therefore also result in low fill-in.

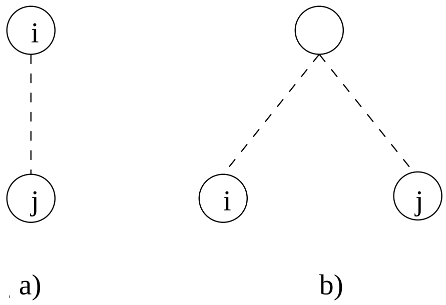

Figure 1: a) l_{ij} might or might not be 0. b) l_{ij} must be 0.

2 A local problem

We first give a formal description of the problem mentioned in the previous section. We will call this problem Γ.

Γ: Given a graph G, a Nested Dissection ordering of G with minimal separators, and the resulting elimination tree T, find an ordering that gives minimum fill-in while preserving the structure of T.

In this section we show that in order to solve Γ, we need to change the relative ordering of the vertices in each separator, and that this local reordering for each separator can be done independently of the other separators in the graph.

Definition: Let G be a graph ordered by a Nested Dissection ordering α with minimal separators, and let T be the resulting elimination tree. We define a *separator tree* corresponding to T, to be a tree whose vertices represent the separators of G chosen by α, and whose leaves represent the components of G that cannot be separated further.

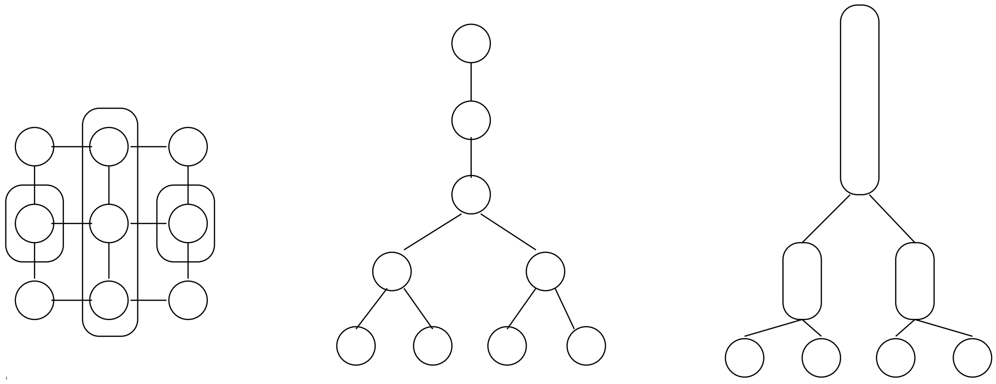

Figure 2: The elimination tree and the separator tree for a graph.

An example of a separator tree is given in Figure 2. The vertices of a separator tree are called *s-nodes*. Every s-node consists of one or more vertices of G, and corresponds to a separator. The s-nodes in the separator tree are related to each other in the same way as the separators are related in T; i.e. the root s-node corresponds to the separator in G that is eliminated last. The leaf s-nodes correspond to the components of G that cannot be separated further and that are eliminated first.

Lemma 1 *Each separator in G chosen by a Nested Dissection ordering with minimal separators is a clique in the filled graph G^*.*

Proof: Lemma 1 follows if we show that the s-nodes in a separator tree ST are cliques in G^*. It is easy to see that the leaf s-nodes are cliques since they cannot be separated further by removal of any vertices. We have to show that all the other s-nodes are also cliques. Let S be any s-node in ST that is not a leaf. Let x be the vertex in S which is eliminated first and let y be a child of x in T, where T is an elimination tree that ST corresponds to. (See Figure 3). Since S is a minimal separator, there must be an edge in G from each vertex in S to some vertex in $T(y)$. Thus $S \subseteq adj_G(T(y))$. By the "Path Theorem" ([4]), y has edges in G^* to all the vertices of $adj_G(T(y))$. Therefore when y is eliminated, $adj_G(T(y))$ becomes a clique in G^*. Thus S must also be a clique in G^*. ■

Lemma 2 *Changing the relative ordering of the vertices in an s-node does not result in any changes in the elimination tree structure.*

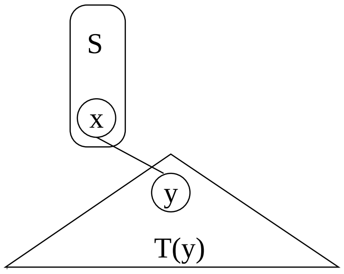

Figure 3: Picturizing the proof of Lemma 4.1.

Proof: Since we choose the separators before we order the vertices in them, a local reordering of vertices in a separator does not affect the choice of separators. It follows from Lemma 1 that the vertices in an s-node will induce a path in the elimination tree regardless of their relative ordering, and thus the elimination tree will remain the same. ∎

Lemma 2 shows that we can reorder the vertices in an s-node without changing the elimination tree structure. In order to find a good reordering of the vertices in an s-node, we have to know which portion of the fill-in can be reduced by such a reordering. We divide the fill-in that is caused by the elimination of vertices in S, into four disjoint groups:

1. Fill-in that occurs within S
2. Fill-in that occurs within another s-node
3. Fill-in that occurs between other s-nodes
4. Fill-in that occurs between S and other s-nodes

Some of this fill-in occurs regardless of the local ordering of the vertices in an s-node. The following lemmas show where this kind of fill-in occurs. Lemmas 3 and 4 are stated without proofs, since they both follow directly from Lemma 1.

Lemma 3 *Changing the relative ordering of the vertices in an s-node S does not result in any changes in the size of fill-in that occurs within S.*

Lemma 4 *Changing the relative ordering of the vertices in an s-node does not result in any changes in the size of fill-in that occurs within another s-node.*

Lemma 5 *Changing the relative ordering of the vertices in an s-node S does not result in any changes in the size of fill-in that occurs between other s-nodes.*

Proof: We need only to consider those s-nodes that lie on the path between S and the root s-node. Whatever lies below S is eliminated before S, and the s-nodes that lie elsewhere cannot be affected by the elimination of S. Let S_1, S_2 and S be s-nodes as shown in Figure 4. Let x_i be a vertex in S_1 and x_j be a vertex in S_2, where x_i and x_j are numbered respectively i and j, where $(i > j)$. By the "Path Theorem, we know that $l_{ij} \neq 0$ if and only if there exists a path $x_i, x_{p_1}, ..., x_{p_t}, x_j$ in G such that all subscripts in $\{p_1, ..., p_t\}$ are less than j. If no such path exists then a reordering of the vertices in S cannot create such a path, since all the vertices in S are numbered lower than j. By the same argument, if such a path does exist, then it will exist also after a reordering of the vertices in S. ∎

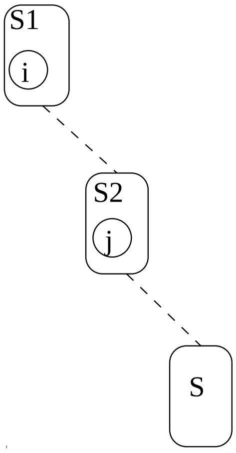

Figure 4: The s-nodes in the proof of Lemma 4.5.

The only vertices whose elimination may result in fill-in edges from S to higher numbered vertices, are the vertices in S and the descendants of S. It should be clear from the previous lemmas that, by reordering S, we can only reduce the number of fill-in edges between S and its higher numbered neighbors, that are caused by the elimination of vertices within S. Thus we need only to consider the fill-in edges that belong to Group 4.

Lemmas 1 - 5 and the discussion above suggest a bottom-up elimination tree reordering that, beginning with leaf s-nodes, reorders one s-node at a time. However, as we will see in the rest of this section, each s-node can be reordered independently of the other s-nodes. Thus the s-nodes can be reordered in an arbitrary order, or in parallel.

Definition: Let *G=(V,E)* be a graph, S a set of vertices from V, and let α

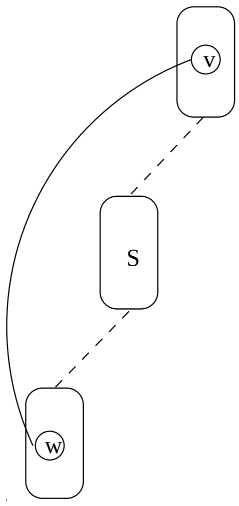

Figure 5: Picturizing the proof of Lemma 4.6.

be an ordering of the vertices in V. We define $madj_G(S)$ to be the set $\{x \notin S \mid (x, y) \in E,\ y \in S,\ x$ is higher numbered than all the vertices in $S\}$. We will call the set $(madj_{G^*}(S) - madj_G(S))$ the *fill-in neighbors* of S.

We will see in the next lemma that we do not have to consider the fill-in neighbors of S when we reorder S, since these have edges in G^* to all the vertices in S regardless of the relative ordering of the vertices in S.

Lemma 6 *Let G be a graph ordered by a Nested Dissection ordering with minimal separators, and let T and ST be the resulting elimination, and separator trees. If there is a fill-in edge in G^* from an s-node S to a higher numbered vertex v in T that is caused by the elimination of a descendant of S, then every vertex in S has an edge in G^* to v.*

Proof: Let S be any s-node in ST that is not a leaf or the root. Since there is a fill-in edge from S to v, by Theorem 3.5 in [2], there must be a vertex w as shown in Figure 5 that has an edge to v in G. By the same theorem, every vertex in S must also have edges to v. ∎

We see by Lemma 6 that when we reorder S, it is enough to consider the edges in G, rather than G^*. The fill-in neighbors of S caused by the elimination of the descendants of S, have edges to all the vertices in S. Therefore, they cannot cause any new fill-in when we reorder S. Hence we need only to consider edges between S and $madj_G(S)$. But some of the vertices in $madj_G(S)$ might also be fill-in neighbors of S; i.e. they might have fill-in edges to all the vertices in

S, caused by the elimination of the descendants of S. Hence we are allowed to exclude such vertices from the neighborhood of S which we will work with, since they cannot introduce any more fill-in to S.

The following algorithm *Reorder* describes how a graph G can be reordered to reduce fill-in, given a separator tree for G. It also gives a description of the local reordering problem for each s-node. We will call this problem Δ.

Algorithm *Reorder* (*ST:* Separator tree; *G:* Graph);

begin
  mark the root s-node in *ST;*
  while there are unmarked s-nodes in ST **do**
    pick an unmarked s-node S in *ST;*
    $adj'(S) = madj_G(S) - \cup\{madj_G(C) | C$ is a descendant of $S\}$;
    Δ: Find a reordering of S that gives minimum fill-in
      between S and $adj'(S)$;
    mark *S;*
  end-while;
end;

Theorem 1 *The algorithm Reorder solves the problem* Γ.

Proof: The proof follows from Lemmas 1 - 6 and the discussion above. ■

We have shown that the problem Γ decomposes into smaller subproblems Δ, which can be solved independently for each s-node. For parallel algorithms, this means that the s-nodes can be reordered in parallel. This will reduce the time required to reorder a graph. In the next section, we show that Δ, though more restricted and structured than the general minimum fill-in problem, is still NP-hard.

Now we want to show that it is worth trying to reduce the fill-in size in an elimination tree, and that in some cases considerable amount of fill-in may be reduced. Let C and S be s-nodes as shown in Figure 6 a), where C is the parent of S. Let f be the number of edges in the filled graph between the vertices in $S \cup C$. Then we have

$$f \geq \frac{|C|(|C|-1)}{2} + \frac{|S|(|S|-1)}{2} + |C|,$$

$$f \leq \frac{|C|(|C|-1)}{2} + \frac{|S|(|S|-1)}{2} + |C||S|.$$

Thus the maximum number of edges that might be possible to reduce is $|C|(|S|-1)$. Since for most values of C and S we have that

$$|C|(|S|-1) \leq \frac{|C|(|C|-1)}{2} + \frac{|S|(|S|-1)}{2} + |C|$$

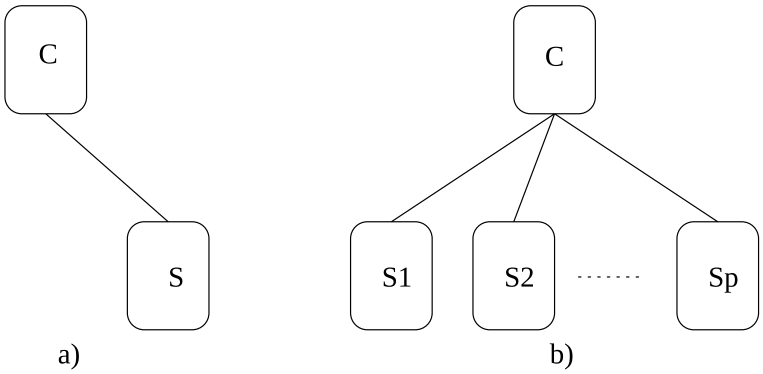

Figure 6: Illustrating the discussion on the number of fill-in edges.

there are not as many edges that can be reduced as the ones that must occur since C and S are cliques. But if C has several children as shown in Figure 6 b), then if all p children are of equal size, we have

$$f \geq \frac{|C|(|C|-1)}{2} + p\frac{|S|(|S|-1)}{2} + p|C|,$$

$$f \leq \frac{|C|(|C|-1)}{2} + p\frac{|S|(|S|-1)}{2} + p|C||S|.$$

If p is large enough, and $|C|$ is larger enough than $|S|$, then we can have

$$p|C|(|S|-1) \geq \frac{|C|(|C|-1)}{2} + p\frac{|S|(|S|-1)}{2} + p|C|.$$

Thus in some cases we can potentially have at least as many unnecessary fill-in edges as the ones that are bound to occur, and it is possible to reduce the number of such fill-in edges.

3 Still NP-hard

In this section we are going to show that the problem Δ, described in the previous section, is NP-hard. Let $adj'(S)$ be as described by the algorithm in the previous section. We can view S and its neighborhood as a separate graph with the vertex set consisting of S and $adj'(S)$. We do not have to consider fill-in between two vertices that are both in S or the fill-in between two vertices that are both in $adj'(S)$. The only edges we are interested in, are the ones between S and $adj'(S)$. Therefore, we can work on a graph that has only the vertices in S and $adj'(S)$, and only the edges that are between S and $adj'(S)$. This observation leads us to the following definition.

Definition: Let S be an s-node. We define G_S to be the bipartite graph $(S, adj'(S), E)$, where E contains only those edges between the vertices in S and their neighbors in $adj'(S)$. (See Figure 7 b).

In order to show that Δ is NP-hard, we need some results from Yannakakis who shows in [5] that the general minimum fill-in problem is NP-hard. Lemmas 7 and 8 are from his article and are quoted without proofs.

Definition: A bipartite graph $G = (P, Q, E)$ is a *chain graph* if the neighborhoods of the vertices in P form a chain; i.e., there is a bijection $\pi : \{1, ..., |P|\} \leftrightarrow P$ such that $adj_G(\pi(1)) \supseteq adj_G(\pi(2)) \supseteq ... \supseteq adj_G(\pi(|P|))$. Then the neighborhoods of the vertices in Q form also a chain.

Definition: Let *G=(P,Q,E)* be a bipartite graph. We define *C(G)* to be the graph (V, E'), where $V = P \cup Q$ and $E' = E \cup \{(u,v) | u, v \in P\} \cup \{(u,v) | u, v \in Q\}$. Thus P and Q are cliques in *C(G)*.

Lemma 7 *Let G be a bipartite graph. C(G) is chordal if and only if G is a chain graph.*

Lemma 8 *It is NP-hard to find the minimum number of edges whose addition to a bipartite graph G=(P,Q,E) gives a chain graph.*

These two lemmas show that a restriction of the general minimum fill-in problem is NP-hard, implying that the general problem is also NP-hard. We will see in the next lemma, that this restricted problem is indeed equivalent to Δ.

Lemma 9 *$C(G_S)$ is chordal if and only if we can eliminate the vertices in S without introducing any fill-in between S and $adj'(S)$.*

Proof: *(if)* This is easy to see. We can first eliminate the vertices of S in $C(G_S)$ without introducing any fill-in between S and $adj'(S)$. This will not introduce any fill-in in $C(G_S)$ since S and $adj'(S)$ are cliques. Then we can eliminate the vertices of $adj'(S)$ and since they induce a clique in $C(G_S)$, this can be done without introducing any fill-in in $C(G_S)$. Thus $C(G_S)$ has a perfect elimination ordering and must be chordal.
(only if) Since $C(G_S)$ is chordal, it has a perfect elimination ordering. We need to show that there exists a perfect elimination ordering of $C(G_S)$ which eliminates the vertices of S before the vertices of $adj'(S)$. We know by Lemma 7 that G_S is a chain graph. Let $s = |S|$ and let α be an ordering of S such that $adj_{G_S}(\alpha(1)) \supseteq ... \supseteq adj_{G_S}(\alpha(s))$. The neighborhood of $\alpha(s)$ in $C(G_S)$ is $adj_{C(G_S)}(\alpha(s)) = adj_{G_S}(\alpha(s)) \cup (S - \alpha(s))$. This is a clique in $C(G_S)$, since S and $adj'(S)$ are cliques, and each vertex in S has edges to all the vertices of $adj'(S)$ that are neighbors of $\alpha(s)$. Thus $\alpha(s)$ is simplicial in $C(G_S)$ and can be eliminated first without introducing fill-in. When $\alpha(s)$ is eliminated, we can find a new simplicial vertex in the same way, since the chain property is

hereditary. Thus we can eliminate the vertices in S first without introducing any fill-in between S and $adj'(S)$. ■

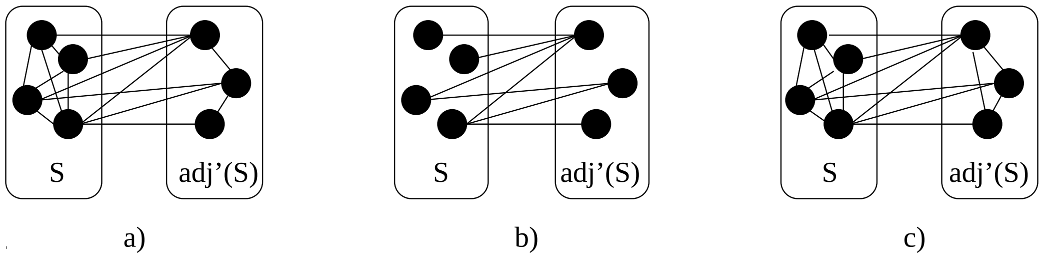

Figure 7: a) An s-node S and $adj'(S)$. b) G_S. c) $C(G_S)$.

Theorem 2 *Δ is NP-hard.*

Proof: The proof follows from Lemmas 7 - 9. ■

4 Heuristics

In this section we discuss how we can solve the problem Δ described in the previous section. Since the problem is NP-hard, it is not likely that one will find a polynomial time algorithm that solves it optimally.

Already existing heuristics like Minimum Degree or Nested Dissection may be used in order to solve Δ. However, there are no guarantees for how well these will perform when used for solving Δ. It is easy to find examples which show that these heuristics may result in more fill-in than optimal. On the other hand, since all the s-nodes can be reordered simultaneously, it is possible to find an optimal solution in reasonable time by simply trying all possibilities, if the s-nodes are small enough. Also, both Minimum Degree and Nested Dissection try to reduce the number of fill-in edges globally. Although we can run these heuristics locally on a single s-node, they consider all four groups of fill-in, whereas only fill-in edges that belong to Group 4 need to be considered.

We now present a new heuristic for reordering the vertices in an s-node to reduce fill-in. The algorithm is called *MinimumFillFirst*, and is locally sensitive only to Group 4 of fill-in. The idea behind this heuristic is to find how much fill-in each vertex in S will produce between S and $adj'(S)$ when eliminated first, and to eliminate the one that produces the least number of such fill-in edges first. The following lemma shows the size of the fill-in produced by the vertex that is eliminated first.

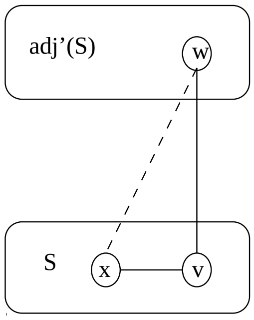

Figure 8: The proof of lemma 4.10.

Lemma 10 *Let $G = (V,E)$ be a graph ordered by a Nested Dissection ordering with minimal separators. Let S be an s-node, v be a vertex in S, and let G_S be as described in the previous section. If v is eliminated first of all vertices in S then the number of fill-in edges between S and $adj'(S)$ in G^* produced by the elimination of v is:*

$$\sum_{w \in adj_{G_S}(v)} |\{x : x \in S \wedge (x,w) \notin E\}|.$$

Proof: Since S is a clique in G^*, v has edges to all the other vertices in S. Therefore, for each vertex $x \in S$, if $(x,\ w) \notin E$ for a neighbor w of v in G_S ($w \in adj'(S)$ and $w \in adj_G(v)$), then there will be a fill-in edge $(x,w) \in E^*$. This situation is illustrated in Figure 8. ∎

We can now formally describe our heuristic by the following algorithm. First the number of fill-in edges for each vertex in S is found (*FindSum*). Then in each iteration a vertex with the least sum is chosen and numbered. This vertex is taken away and the sums of the other vertices are updated (*Update*).

Algorithm *MinimumFillFirst* (*S:* S-node; *adj'(S):* Set of vertices);

begin
    FindSum;
    while there is more than one vertex in S **do**
        find the vertex $v \in S$ with the least sum;
        number *v;*
        Update(v);
    end-while;
end;

Next we give the algorithm for finding the sums. This algorithm uses Lemma 10 to compute the number of fill-in edges each vertex produces when eliminated first.

Algorithm *FindSum;*

begin
    for each vertex $v \in S$ **do**
        v.sum $= 0$;
        for each vertex $w \in adj'(S)$ **do**
            if $(v, w) \in E$ **then**
                for each vertex $x \in S$ **do**
                    if $x \neq v$ and $(w, x) \notin E$ **then**
                        v.sum $=$ v.sum $+ 1$;
    end-for;
end;

The time complexity of this algorithm is $O(s^2h)$, where $s = |S|$ and h is the height of the elimination tree. This is because $|adj\prime(S)| \leq h$ since we only look at the neighbors of S on the path from S to the root.

We will now look at the algorithm *Update* which has a vertex v as input parameter. The vertex v is the vertex that is currently being numbered. We look at each neighbor w of v ($w \in adj'(S)$), and decide if the elimination of v results in any fill-in between w and vertices in S.

For each vertex $x \in S$, where $x \neq v$, there is either an edge $(w, x) \in E$ or there will be a fill-in edge $(w, x) \in E^*$ when v is eliminated. If $(w, x) \in E$ then the elimination of x would have produced as many fill-in edges from w to vertices in S as the elimination of v produces. After v is eliminated, the elimination of x will not result in these fill-in edges any more since they already exist. Therefore, the number of such fill-in edges must be subtracted from x.sum. We update the sums for all the vertices x in this way. We must also update x.sum for vertices x that have neighbors which are not neighbors of v. After v is eliminated, there will not be any fill-in edges between v and these neighbors of x.

Note that if a vertex x does not have edges to any of v's neighbors then x will have edges to these when v is eliminated. This does not change x.sum or the sum of any other vertex, since every vertex in S has edges to every neighbor in $adj'(S)$ of v after v's elimination. Thus there will not be any more fill-in between these neighbors and vertices in S.

Algorithm *Update* (*v:* Vertex);

begin

```
    for each vertex w ∈ adj'(S) do
        if (v, w) ∈ E then
            fill = 0;
            unmark all vertices in S;
            for each vertex x ∈ S do
                if (w, x) ∉ E then
                    fill = fill + 1
                else
                    mark x;
            for each marked vertex x do
                x.sum = x.sum - fill;
            adj'(S) = adj'(S) - w;
        end-if;
    for each vertex x ∈ S, where (x ≠ v) do
        for each vertex w ∈ adj'(S) do
            if (w, x) ∈ E and (v, w) ∉ E then
                x.sum = x.sum - 1;
    S = S - v;
end;
```

It is easy to see that the time complexity of the algorithm *Update* is $O(sh)$. This gives a total time complexity of $O(s^2 h)$ for the algorithm *MinimumFillFirst* since the loop is executed s times.

We will now discuss the time complexity of reordering the whole graph $G = (V, E)$. This complexity depends on whether the reordering is done sequentially or in parallel. The time complexity of finding $adj'(S)$ for each s-node must also be considered. We now give a recursive algorithm *FindAdj* that finds $adj'(S)$ for all the s-nodes in the separator tree, when called with the root s-node as parameter.

```
Algorithm FindAdj (S: S-node);

begin
    adj'(S) = {};
    if S is a leaf s-node then
        for each vertex v ∈ S do
            for each vertex x on the path to the root do
                if (x, v) ∈ E then
                    mark x by S;
                    adj'(S) = adj'(S) ∪ {x};
                end-if;
    else
        for each child S_1 of S do
            FindAdj(S_1);
        for each vertex v ∈ S do
```

```
        for each vertex x on the path to the root do
            if the last s-node that marked x is a child of S then
                mark x by S
            else
                if (x,v) ∈ E then
                    adj'(S) = adj'(S) ∪ {x};
                    mark x by S;
                end-if;
    end-if;
end;
```

The time complexity of the algorithm *FindAdj* is $\sum_{S_i} |S_i|h = O(nh)$, where $n = |V|$. If the algorithm is implemented in parallel, it is possible to achieve a time complexity of $O(h^2)$ by using one processor per s-node. We now change the algorithm *Reorder* so that it uses all the algorithms that we have given.

Algorithm *Reorder* (*ST:* Separator tree; *G:* Graph);

```
begin
    mark the root s-node in ST;
    FindAdj (the root s-node);
    while there are unmarked s-nodes in ST do
        pick an unmarked s-node S in ST;
        MinimumFillFirst(S,adj'(S));
        mark S;
    end-while;
end;
```

If the algorithm *Reorder* is implemented sequentially, then the time complexity is

$$O(nh) + \sum_{S_i} O(s_i^2 h)$$

where $s_i = |S_i|$ for all s-nodes S_i in ST. Since $s_i \leq h$ and $\sum_{S_i} s_i = n$, we have

$$\sum_{S_i} O(s_i^2 h) = \sum_{S_i} O(s_i h^2) = O(h^2) \sum_{S_i} O(s_i) = O(h^2 n).$$

The time complexity of the algorithm *Reorder* is then $O(hn)+O(h^2 n) = O(h^2 n)$. Note that this is a pessimistic worst case time complexity analysis, and we may expect the algorithm to perform better in practice. It will indeed be difficult to perform this bad, since all the s-nodes cannot be as large as $O(h)$ and simultaneously have $O(h)$ neighbors higher up in the elimination tree.

Note that the time complexity analysis that we have given for the algorithms in this chapter, assumes a space complexity of $O(n^2)$. We must have that much space in order to be able to check if an edge $(v, w) \in E$ in $O(1)$ time. If n is very large, it might be difficult to store $O(n^2)$ entries. It is possible to use other data structures for storing. As an example, each vertex can have a balanced binary tree as its list of neighbors. This way, testing if $(v, w) \in E$ takes $O(\log d)$ time, where d is the largest degree in G. Note also that we have concentrated on making the algorithms as simple and easy to understand as possible. By coding the algorithms in a different way, it might be possible to reduce the time complexity further.

We have shown in Section 2 that the s-nodes can be reordered in parallel. In a parallel algorithm the loop is executed in parallel. Thus the s-nodes are being reordered simultaneously. The complexity of doing this will be bounded by the complexity of reordering the largest s-node in the separator tree. If the s-nodes are small, we can solve the problem of reordering all the s-nodes optimally in reasonable time though this will mean trying every permutation of the vertices in every s-node.

References

[1] P. Heggernes, *Nested dissection may produce very high elimination trees.* An unpublished manuscript, 1992.

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

[3] F. Manne, *Reducing the height of an elimination tree through local reorderings*, Tech. Rep. 51, University of Bergen, Norway, 1991.

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

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