A Simple Cubic Algorithm for Computing Minimum Height Elimination Trees for Interval Graphs

Bengt Aspvall, Pinar Heggernes, Jan Arne Telle

Department of Informatics, University of Bergen
N–5020 Bergen, Norway
email: Bengt.Aspvall@ii.uib.no, Pinar.Heggernes@ii.uib.no, Jan.Arne.Telle@ii.uib.no

Abstract

In parallel sparse Cholesky factorization, finding elimination orderings that produce elimination trees of low height is important. The problem of finding the minimum elimination tree height for a graph is $\mathcal{NP}$-hard for general graphs. This problem is equivalent to the problem of finding the vertex ranking number of a graph. The problem can be solved in polynomial time for special classes of graphs. Aspvall and Heggernes presented a polynomial time algorithm for solving the problem for interval graphs, in 1993. This algorithm is $O(n^4)$. Later, when Deogun, Kloks, Kratsch and Müller solved the vertex ranking problem for permutation graphs in time $O(n^6)$, they mentioned that the problem for interval graphs could be solved in time $O(n^3)$. In this paper we present a simple $O(n^3)$ algorithm for finding minimum elimination tree height for interval graphs. In addition, our algorithm gathers enough information so that an actual elimination tree of minimum height can be found within the same time bound.

1 Introduction

Elimination trees play an important role in parallel sparse Cholesky factorization. The elimination tree height is a bound on the amount of parallelism allowed by the sparse system and its associated graph. Different elimination orderings of a graph may result in elimination trees of varying height. It is desirable to find orderings that result in low elimination trees.

The problem of minimizing the elimination tree height for general graphs is $\mathcal{NP}$-hard. However, there exist classes of graphs for which the problem can be solved in polynomial time. It is therefore important to identify these classes, and to find efficient algorithms for them. Two of the present authors, Aspvall and Heggernes, showed in [1] that the problem can be solved for interval graphs in polynomial time. The first version of their paper was published as a technical report in March 1993. Before this result, little was known about polynomial time solvable classes of graphs apart from trees. The algorithm for interval graphs in [1] is $O(n^4)$, but the main purpose of the paper was to show that the problem for interval graphs was indeed in $\mathcal{P}$. Therefore, concentrating on the simplicity and clarity of the algorithm was important, whereas efficiency was of secondary interest. Now in this paper, we revisit the algorithms from [1], and with some new definitions, we are able to present an $O(n^3)$ algorithm that solves the problem for interval graphs.

The problem of finding the height of a minimum elimination tree for a graph is equivalent to the problem of finding the vertex ranking number of a graph. An optimal vertex ranking does not by itself provide enough information to compute an elimination tree of minimum height. However, given also the input graph, a minimum height elimination tree can be constructed rather efficiently. On the other hand, an optimal vertex ranking can readily be found directly from an elimination tree of minimum height. We refer the reader to [4] for details.

In [4], Deogun, Kloks, Kratsch and Müller show that the vertex ranking problem can be solved for permutation graphs in time $O(n^6)$. They also mention that the vertex ranking problem for interval graphs possesses a certain structure that allows it to be solved in time $O(n^3)$. We will here give a simple cubic algorithm for finding minimum height elimination trees for interval graphs, along with some implementation details, and complete implementation should be straightforward. Furthermore, we show that after an $O(n^2)$ preprocessing, updating of each of the minimal separators can be done in constant time whenever needed. This opens for a possible algorithm of lower time complexity if the number of subproblems that can be treated in constant time is lower than $\Theta(n^3)$.

The background for this paper will be [1]. We assume that the reader is familiar with the results of [1], and we will frequently refer to these results. Definitions will be repeated only when we feel that they are of additional interest.

This paper is organized as follows. Some of the definitions from [1] are repeated in Section 2. In Section 3, we introduce the notion of *elimination forests*, which is then utilized to develop a simple algorithm for finding minimum elimination tree height in time $O(n^3)$. Some more implementation details of this algorithm are explained in Section 4. We explain the preprocessing on the maximal cliques and the minimal separators, and how they can be updated later on. In Section 5, we discuss whether it might be possible to develop an algorithm that is significantly better than $O(n^3)$. Finally, some concluding remarks are given in Section 6.

2 Definitions

The reader is assumed to be familiar with standard graph notation. Background on symbolic factorization, elimination trees, chordal graphs, and clique trees may be found in [1]. A few of these definitions are repeated in this section.

Let G be a chordal graph and let $\mathcal{V}_G = \{C_1, C_2, ..., C_m\}$ be the set of maximal cliques in G. The following characterization of clique trees from [2] will be used later in the paper.

Theorem 2.1 *A tree $\mathcal{T} = (\mathcal{V}_G, \mathcal{E})$ is a clique tree of G if and only if for every pair of distinct cliques C_i and $C_j \in \mathcal{V}_G$, the intersection $C_i \cap C_j$ is contained in every maximal clique on the path connecting C_i and C_j in $\mathcal{T}$.*

In [1], the authors worked on two classes of graphs. One of these was interval graphs, and the other one was a more restricted class called Ω, which is a subset of interval graphs. We repeat the definitions of these two classes, along with some of their properties.

Definition: A graph G belongs to Ω if G is chordal and has at most two maximal cliques that contain simplicial vertices.

Definition: A graph $G = (V, E)$ is an *interval graph* if there is a mapping I of the vertices in G into sets of consecutive integers such that for each pair of vertices $v, w \in V$ the following is true: $(v, w) \in E \Leftrightarrow I(v) \cap I(w) \neq \emptyset$.

Definition: A *trail* is a tree $T = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$, and $E = \{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)\}$.

Theorem 2.2 *Let G be a graph in Ω. Then G has a unique clique tree $\mathcal{T}$ that is a trail.*

Theorem 2.3 *Let G be a graph in Ω. Then every minimal v-w separator in G divides G into exactly two components.*

The following theorem, here stated in terms of clique trees, was first proved by Gilmore and Hoffman [5].

Theorem 2.4 *A graph $G = (V, E)$ is an interval graph if and only if there exists a clique tree $\mathcal{T}$ for G, where $\mathcal{T}$ is a trail.*

We would like to stress the difference between the clique trees for the graphs in Ω and for interval graphs. As we see in Theorems 2.2 and 2.4, a graph in Ω has a *unique* clique tree that is a trail, whereas an interval graph may have several clique trees where *at least one* is a trail. Booth and Lueker show in [3] that the interval representation of an interval graph can be constructed in linear time. A linear time algorithm for finding a clique tree of a chordal graph can be found in [6]. Using these results, a trail clique tree of an interval graph can be found within the time bounds of the algorithms presented in this paper.

3 A cubic algorithm

In [1], two polynomial time algorithms were presented. Algorithm *DynHeight* is an $O(n^3)$ algorithm that computes the minimum elimination tree height for an input graph $G \in \Omega$. Algorithm *Interval* is an $O(n^4)$ algorithm that computes the minimum elimination tree height for an interval graph. In this section we examine why *Interval* is slower than *DynHeight* by a factor of n, and how we can eliminate this difference in complexity. In fact, Algorithm *DynHeight* actually computes the minimum height also for interval graphs. However, to find an actual elimination tree of minimum height, some changes are necessary.

Let G be a graph in Ω with maximal cliques $C_1, C_2, ..., C_m$, numbered as they appear in the clique tree of G, and minimal separators $S_1, S_2, ..., S_{m-1}$, where $S_i = C_i \cap C_{i+1}$. Algorithm *DynHeight* keeps a two dimensional array $Opt[1..m, 1..m]$. The entry $Opt[i, j]$ holds the minimum elimination tree height for the subgraph $H_{i,j}$ induced by the vertices in $(C_i \cup ... \cup C_j) - (S_i \cup S_{j-1})$. An additional array $Root[1..m, 1..m]$ is kept to store an actual elimination tree of minimum height. The entry $Root[i, j]$ holds a minimal separator S'_k of $H_{i,j}$, where S'_k is the top separator in a minimum height elimination tree for $H_{i,j}$. In Ω, each subgraph $H_{i,j}$ is connected and thus has a minimum height elimination tree. Furthermore, when S'_k is removed from $H_{i,j}$, the remaining graph has at most two nonempty connected components:

$H_{i,k}$ and $H_{k+1,j}$. Thus for each subproblem $H_{i,j}$, $Opt[i,j]$ and $Root[i,j]$ can be computed in a straightforward way. The situation is slightly different for interval graphs.

Algorithm *Interval* keeps the same arrays *Opt* and *Root* as *DynHeight*. For an interval graph G with maximal cliques $C_1, C_2, ..., C_m$, numbered as they appear in a clique tree of G that is a trail, the subgraph analogous to $H_{i,j}$ is called $I_{i,j}$. A subgraph $I_{i,j}$ may be disconnected. In this case, $Root[i,j]$ is left empty by Algorithm *Interval*, and $Opt[i,j]$ is unaltered and holds a large number as initiated at the beginning of the algorithm. If $I_{i,j}$ is connected then the removal of a minimal separator S'_k can divide the graph into several connected components. Algorithm *Interval* runs through all the connected components of $I_{i,j} - S'_k$ to look up from the table the minimum elimination tree height for each component. This introduces an extra loop in the algorithm increasing the time complexity to $O(n^4)$. We will now introduce *elimination forests* and *empty separators* in order to dismiss this last loop. We hope that these definitions will be useful in other contexts as well.

Definition: An *elimination forest* for a disconnected graph G is a forest where each connected tree is an elimination tree for a connected component of G. The *height* of an elimination forest is the height of a highest connected elimination tree in the forest. An elimination forest of *minimum height* consists of elimination trees of minimum height for each of the connected components of G.

We will change Algorithm *Interval* to reduce the time complexity. We can rather say that we will extend Algorithm *DynHeight* slightly so that it can handle interval graphs correctly without increasing the time complexity. We still have to consider whether the subgraph $I_{i,j}$ is connected or not. If $I_{i,j}$ is not connected, then at least one of the sets S'_k is empty. In this case we want to find an elimination forest of minimum height for $I_{i,j}$. An elimination forest F with connected components $T_1, ..., T_p$, can be regarded as having a dummy top separator S which is empty, and where $T_1, ..., T_p$ are the children of S. Thus we can treat an empty set S'_k as a separator of size 0; an *empty separator*, and try it as the top separator for $I_{i,j}$.

When trying S'_k, which might be empty, as the top separator, the minimum elimination tree heights of $I_{i,k}$ and $I_{k+1,j}$ are found in $Opt[i,k]$ and $Opt[k+1,j]$, which are already computed. If one or both of these subgraphs are disconnected then the corresponding values in the table *Opt* are the heights of elimination forests of minimum height.

```
Algorithm NewInterval;

begin

    for $i = 1$ to $m$ do                (* Initialization*)
        for $j = 1$ to $m$ do
            $Opt[i, j] = \infty$;
            $Root[i, j] = 0$;
        end-for;

    for $i = m$ downto 1 do              (* Main loop*)
        $Opt[i, i] = |Simp(C_i)| - 1$;
        for $j = i + 1$ to $m$ do
            for $k = i$ to $j - 1$ do
                $|S'_k| = |S_k - (S_i \cup S_{j-1})|$;
                $height = |S'_k| + max(Opt[i, k], Opt[k + 1, j])$;
                if $height \leq Opt[i, j]$ then
                    $Opt[i, j] = height$;
                    $Root[i, j] = k$;
                end-if;
                if $|S'_k| = 0$ then
                    $k = j$; (*$I_{i,j}$ is disconnected - jump out of the loop *)
            end-for;
    end-for;

end;
```

Figure 1: Algorithm *NewInterval.*

If $I_{i,j}$ is disconnected, then choosing an empty separator S'_k as the top separator must yield the minimum height. This will also give a correct elimination forest with S'_k as the dummy top separator. Choosing a separator in a component of $I_{i,j}$ that has the highest minimum height elimination tree, as the top separator for $I_{i,j}$, may also give the minimum height for $I_{i,j}$. But this will not result in a valid elimination tree, since $I_{i,j}$ must have an elimination forest which is disconnected. Therefore, when we find out that $I_{i,j}$ is disconnected, we choose an empty separator S'_k as the top separator. Algorithm *NewInterval* is given in Figure 1.

Algorithm *NewInterval* contains also some slight changes other than the ones mentioned above. We find some of these changes necessary because they will make it easier to explain the implementation details in Section 4. The set $Simp(C_i)$ in the algorithm is the set of simplicial vertices in C_i. Setting $Opt[i,i] = |Simp(C_i)| - 1$ is necessary in order to compute the correct height. We remark that Algorithms *DynHeight* and *Interval* in [1] should be changed so that the empty case is treated as the general case, when $j = i$.

Algorithm *NewInterval* computes the minimum elimination tree height of an interval graph. It may be confusing that we choose empty sets to be separators, but this is just symbolic. If the top separator is empty, then we have an elimination forest. Otherwise the graph must be connected and has an elimination tree. The elimination tree defined by the resulting array *Root* at the end of the algorithm is a binary separator tree where some of the separators are empty sets. To find the separators of the actual elimination tree, we must recursively do the following starting from the top separator. If a separator S in the elimination tree is empty then we set $parent(leftchild(S)) = parent(S)$ and $parent(rightchild(S)) = parent(S)$. Since G is assumed connected, we have always a top separator that is nonempty, and thus we have a connected elimination tree for G. Therefore, we have a valid base case, and for each new separator S to consider as we go down the elimination tree, we know that none of the ancestors of S are empty.

In Algorithms *DynHeight* and *Interval*, the table *Root* actually contains the vertices to be placed in the elimination tree, and the tree can be built in linear time in a top-down fashion. How to actually build the elimination tree from the table *Root* is straightforward and was not explained further in [1]. Now in Algorithm *NewInterval*, we only compute the sizes of the separators and not which vertices they contain. Therefore, the construction is a little more complicated. Again we build the tree in a top-down fashion. The vertices of G are marked when they are placed in the elimination tree, and each new separator to be placed in the tree must check which of its vertices are already in the elimination tree. Thus the construction of the elimination tree can be done in time $O(n^2)$.

4 Set update

In this section we look at how we can compute $|S'_k| = |S_k - (S_i \cup S_{j-1})|$ efficiently. For each pair $(i,j), i < j$, $|S'_k|$ is computed for $i \leq k \leq j-1$ by Algorithm *NewInterval*. If the actual sets S'_k were to be computed, this would take $O(n^4)$ time. But the algorithm only needs the sizes of the minimal separators in order to be able to compute the minimum elimination tree height. We will now see that after a preprocessing of $O(n^2)$ on all the pairs of separators (S_i, S_j), $i < j$, computing a value $|S_k - (S_i \cup S_j)|$ can be done in constant time when k is between i and j.

Lemma 4.1 *Let G be an interval graph, and let $S_1, S_2, ..., S_{m-1}$ be the minimal separators of G numbered as they appear in a clique tree that is a trail. For $1 \leq i \leq k \leq j < m$, $|S_k - (S_i \cup S_j)| = |S_k| - |S_i \cap S_k| - |S_k \cap S_j| + |S_i \cap S_j|$.*

Proof: From set theory we have the following result for general sets: $|A - (B \cup C)| = |A| - |A \cap B| - |A \cap C| + |A \cap B \cap C|$. From Theorem 2.1, we can conclude that $(S_i \cap S_j) - S_k = \emptyset$. Thus, $S_i \cap S_k \cap S_j = S_i \cap S_j$. From this the lemma follows with $A = S_k$, $B = S_i$, and $C = S_j$. □

As we can see from Lemma 4.1, $|S_k - (S_i \cup S_{j-1})|$, for any triple (i, k, j), can be computed in constant time if $|S_k|$, $|S_i \cap S_k|$, $|S_k \cap S_{j-1}|$ and $|S_i \cap S_{j-1}|$ are already computed. We will show that in time $O(n^2)$, $|S_i \cap S_j|$ can be computed for all pairs (i, j), $1 \leq i < j < m$.

In order to explain how to find the sizes of the intersections between all pairs of minimal separators in G, we need the following definitions. Let $\mathcal{T}$ be a clique tree of G that is a trail. To each vertex v in G, we associate a number $right(v)$ which is the number of the rightmost maximal clique in $\mathcal{T}$ that contains v. That is, $right(v) = max\{i \mid v \in C_i\}$. Let the vertices in each minimal separator S_i be sorted by their $right$ values. The vertices are renamed so that $S_i = \{v_1^i, v_2^i, ..., v_{n_i}^i\}$, where $right(v_k^i) \leq right(v_{k+1}^i)$, $1 \leq k < n_i$.

For each S_i, $i = 1, 2, ..., m-2$, we want to find $|S_i \cap S_j|$ for $i < j < m$. For a vertex v_k^i in S_i, we know by Theorem 2.1 that if $right(v_k^i) > j$ then $v_k^i \in S_j$, since $S_j = C_j \cap C_{j+1}$. Let k be the smallest index for which $right(v_k^i) > j$. Then we have $right(v_{k-1}^i) \leq j$, and thus $v_{k-1}^i \notin S_j$. In this case, by the same theorem, we can conclude that $v_1^i, ..., v_{k-1}^i \notin S_j$, and $v_k^i, ..., v_{n_i}^i \in S_j$. The preprocessing algorithm is given in Figure 2.

In order to consider the time complexity of Algorithm *Preprocess*, notice that k is initiated outside the j-loop, and the number of steps in this loop is $O(n_i + (m - 1 - i))$. The total number of steps for Algorithm *Preprocess* is then $O(\sum_{i=1}^{m-2}(n_i + (m - 1 - i)) = O(n^2)$, since $n_i \leq n$ and $m \leq n$, where n is the number of vertices in G. Regarding Algorithm *NewInterval*, the sizes of all the minimal separators and of $Simp(C_i)$ for all the maximal cliques can easily be computed within this time bound. Algorithm *NewInterval* is clearly $O(n^3)$.

```
**Algorithm** *Preprocess;*

**begin**

    **for** $i = 1$ **to** $m - 2$ **do**
        $k = 1$;
        **for** $j = i + 1$ **to** $m - 1$ **do**
            **while** $k \leq n_i$ **and** $right(v_k^i) \leq j$ **do**
                $k = k + 1$;
            $|S_i \cap S_j| = n_i - k + 1$;
        **end-for;**
    **end-for;**

**end;**
```

Figure 2: Algorithm *Preprocess.*

5 Can we do better?

As we have seen, the overhead work in Algorithm *NewInterval* is done in time $O(n^2)$. What increases the time complexity is the number of minimal separators that we have to run through for each subproblem $I_{i,j}$.

In dynamic programming algorithms, a linear factor can often be removed from the time bound if the problem possesses some monotonicity property. When computing $Opt[i, j]$ and $Root[i, j]$, Algorithm *NewInterval* runs through every k between i and j. Assume that the following monotonicity property holds: There exists an integer k, such that $Root[i, j - 1] \leq k \leq Root[i + 1, j]$, and setting $Root[i, j] = k$ gives the minimum height for $I_{i,j}$. In this case, the total time is reduced to $O(n^2)$ when the innermost loop only runs for $k = Root[i, j - 1]$ to $Root[i + 1, j]$.

Our algorithm finds an arbitrary minimum height elimination tree or forest for each subproblem. With this approach, the monotonicity property mentioned above does not hold for interval graphs or for the class Ω. The lack of monotonicity in the sizes of the separators S'_k, $i \leq k < j$, seems to be an obstacle. We still suspect and hope that at least one of these classes possesses a similar property. For example, can one, by restricting the attention to minimum height elimination trees with some additional requirements, obtain a desirable monotonicity property? We have not been able to prove such a result so far.

6 Conclusion

In this paper we have given a simple $O(n^3)$ algorithm for finding minimum height elimination trees for interval graphs. An interesting open problem is whether it is possible to develop an algorithm that is significantly better than $O(n^3)$. Our algorithm uses the method of dynamic programming, and there exist techniques for increasing the efficiency of dynamic programming algorithms. We have given a discussion on how one might possibly reduce the time complexity of this particular algorithm. Unfortunately, we have to leave as an open problem to show some appropriate monotonicity property for interval graphs.

References

[1] B. Aspvall and P. Heggernes, *Finding minimum height elimination trees for interval graphs in polynomial time*, BIT, 34 (1994), pp. 484 – 509.

[2] J. R. S. Blair and B. W. Peyton, *On finding minimum-diameter clique trees*, Nordic Journal of Computing, 1 (1994), pp. 173 –201.

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

[4] J. S. Deogun, T. Kloks, D. Kratsch, and H. Müller, *On vertex ranking for permutation and other graphs*, Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science, 775 (1994), pp. 747 – 758.

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

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