Available online at www.sciencedirect.com
SCIENCE DIRECT®
Discrete Mathematics 306 (2006) 351–358
DISCRETE MATHEMATICS
www.elsevier.com/locate/disc

A linear time algorithm to list the minimal separators of chordal graphs☆

L. Sunil Chandran[1], Fabrizio Grandoni

Max-Planck Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany

Received 10 November 2003; received in revised form 16 December 2004; accepted 8 November 2005
Available online 27 January 2006

Abstract

Kumar and Madhavan [Minimal vertex separators of chordal graphs, Discrete Appl. Math. 89 (1998) 155–168] gave a linear time algorithm to list all the minimal separators of a chordal graph. In this paper we give another linear time algorithm for the same purpose. While the algorithm of Kumar and Madhavan requires that a specific type of PEO, namely the MCS PEO is computed first, our algorithm works with any PEO. This is interesting when we consider the fact that there are other popular methods such as Lex BFS to compute a PEO for a given chordal graph.

Keywords: Minimal separator; Chordal graph; Perfect elimination ordering

1. Introduction

Let C be a cycle in a graph G. A chord of C is an edge of G joining two vertices of C which are not consecutive. A graph G is called a chordal (or triangulated) graph iff every cycle in G, of length 4 or more has a chord. Chordal graphs arise in many applications (see [7,13,17]). Chordal graphs constitute one of the most important subclasses of perfect graphs [7].

In a connected graph G, a *separator* S is a subset of vertices whose removal separates G into at least two connected components. S is called a $(a-b)$ *separator* iff it disconnects vertices a and b. A $(a-b)$ separator is said to be a *minimal separator* iff it does not contain any other $(a-b)$ separator.

The problem of listing all minimal separators is one of the fundamental enumeration problems in graph theory, which has great practical importance in reliability analysis for networks and operations research for scheduling problems [8,6,1].

The problem of listing all minimal separators of an undirected graph is considered by various authors [6,11,15]. A $O(n^6 R_\Sigma)$ algorithm is given in [11], to list all *minimal* separators, where R_Σ is the total number of minimal separators in the graph. This is improved in [15] to $O(n^3 R_\Sigma^+ + n^4 R_\Sigma)$, where $R_\Sigma^+ \leqslant (n(n-1)/2-m)R_\Sigma$. ($n$ and m represent the

☆ A preliminary version of this paper appeared in the Proceedings of the Seventh International Computing and Combinatorics Conference 2001, Lecture Notes in Computer Science, vol. 2108, Springer, Berlin.

E-mail addresses: sunil@mpi-sb.mpg.de (L.S. Chandran), grandoni@mpi-sb.mpg.de (F. Grandoni).

[1] Part of this work was done when the author was at the Indian Institute of Science, Bangalore and was partially supported by the INFOSYS fellowship.

0012-365X/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.disc.2005.12.010

number of vertices and number of edges respectively.) The current best-time algorithm for this problem is by Berry et al. [2]: they present an algorithm which computes the set of minimal separators of a graph in $O(n^3 R_\Sigma)$ time. Algorithms to list the minimal separators for some subclasses of perfect graphs (e.g. permutation graphs) are given in [9,10].

Kumar and Madhavan [12] presented a linear time ($O(m+n)$) algorithm that lists all minimal separators of a chordal graph. Their algorithm first computes a specific kind of perfect elimination ordering (namely the ordering given by the maximum cardinality search (MCS) algorithm of Yannakakis and Tarjan [18]) and then makes use of certain properties of this particular PEO to list all the minimal separators. But there exists many other ways to generate PEOs. For example, the Lexico Graphic Breadth First Search algorithm of Rose et al. [14] can output a PEO which is different from what is generated by MCS. In fact even the Lex BFS and MCS together also cannot exhaust all the possible PEOs. Shier [16] gives a characterization of all the possible PEOs in a chordal graph. Chandran et al. [4] give a fast algorithm for generating all the PEO in a given chordal graphs in constant amortized time.

In this paper we give a different linear time algorithm for listing all the minimal separators of a chordal graph. The advantage of this algorithm over the algorithm of Kumar and Madhavan is that, it does not depend on the particular type of PEO used. For example, there may be an application using the Lex BFS PEO and then at some point if it wants to list the minimal separators, it is a waste of effort to recompute a MCS PEO just for this purpose.

Our algorithm is based on the same structural characterization of minimal separators of chordal graphs as that of [12]. But the algorithms are different. (In fact when we wrote the preliminary version of the paper, we were not aware of Kumar and Madhavan's work, and thus a different proof of this structural characterization (Theorem 1) also appears in the preliminary version.)

2. Preliminaries

Let $G = (V, E)$ be a simple, connected, undirected graph. $|V|$ and $|E|$ will be denoted by n and m, respectively. $N(v)$ will denote the set of neighbours of v, that is $N(v) = \{u \in V : (u, v) \in E\}$. For $A \subseteq V$, we use $N(A)$ to denote the set $\bigcup_{v \in A} N(v) - A$. The subgraph of G induced by the nodes in A will be denoted by $G[A]$.

A bijection $f : V \to \{1, 2, \ldots, n\}$ is called an ordering of the vertices of G. Then $f(v)$ is referred to as the number associated with the vertex v, or simply the *number of* v with respect to the ordering f. Given an ordering f of a graph G, we define the following terms.

Definition 1. Let $A \subseteq V$. The *highest*(A) is defined to be the vertex with the highest number in A. Similarly *lowest*(A) is the vertex in A with the lowest number.

Definition 2. A path $P = (w_1, w_2, \ldots, w_k)$ in G is called an *increasing path*, iff $f(w_1) < f(w_2) < \cdots < f(w_k)$. It is called a *decreasing path* iff $f(w_1) > f(w_2) > \cdots > f(w_k)$. A single node can be considered as either increasing or decreasing.

Definition 3. A vertex $u \in N(v)$ is called a *higher neighbour* of v iff $f(u) > f(v)$. The set of higher neighbours of v will be denoted by $N_h(v)$ i.e.,

$$N_h(v) = \{u \in N(v) : f(u) > f(v)\}.$$

Similarly, the set of *lower neighbours* of v is denoted by $N_l(v)$.

$$N_l(v) = \{u \in N(v) : f(u) < f(v)\}$$

$d_h(v) = |N_h(v)|$ and $d_l(v) = |N_l(v)|$.

Definition 4. An ordering f of G is called a *perfect elimination ordering* (PEO) iff for each $v \in V$, $G(\{v\} \cup N_h(v))$ is a complete subgraph (clique) of G.

A graph G is chordal if and only if there exists a PEO for G [7]. Note that there can be more than one PEO for a given chordal graph. The observations and the algorithm presented in this paper are valid with respect to any PEO.

Therefore, we assume that a PEO is given on G and we just use PEO (v) to denote the number of v with respect to this PEO.

A chordless path from u to v is defined to be a path from u to v in G such that no two non-consecutive nodes of the path are adjacent. The reader can easily verify that if there is a path between u and v then there is a chordless path also. For example, a shortest path between u and v has to be a chordless path.

Lemma 1. *Let $P = (w_1, w_2, \ldots, w_k)$ be a chordless path in a chordal graph G and let $w_i = highest(P)$. Then $(w_1, w_2, \ldots, w_i)$ is an increasing path while $(w_i, w_{i+1}, \ldots, w_k)$ is a decreasing path.*

Corollary 1. *Let $P = (w_1, w_2, \ldots, w_k)$ be a chordless path and $w_k = highest(P)$. Then P is an increasing path.*

Corollary 2. *Let $P = (w_1, w_2, \ldots, w_k)$ be a chordless path and* $\mathrm{PEO}(w_1) < \mathrm{PEO}(w_k)$. *Then* $\mathrm{PEO}(w_2) > \mathrm{PEO}(w_1)$.

Lemma 2. *Let $P = (w_1, w_2, \ldots, w_k)$ be an increasing path in a chordal graph. Let* $\mathrm{PEO}(u) > \mathrm{PEO}(w_k)$ *where $u \in N(w_1)$. Then $u \in N_h(w_k)$.*

Proof. We prove the lemma by induction on $|P|$, the number of nodes in P. For $|P| = 1$, the lemma is trivial. Assume that for all increasing paths with $|P| = k-1$, where $k > 1$, the lemma is true. Let $P = (w_1, w_2, \ldots, w_k)$ be an increasing path with $|P| = k$. Note that we have $\mathrm{PEO}(u) > \mathrm{PEO}(w_k) > \mathrm{PEO}(w_{k-1})$ since P is an increasing path. Applying the induction assumption on the $(k-1)$-node path $(w_1, w_2, \ldots, w_{k-1})$, we get $u \in N_h(w_{k-1})$. Now $\{w_k, u\} \subseteq N_h(w_{k-1})$ and by definition of a PEO, $N_h(w_{k-1})$ is a clique, and $(u, w_k) \in E$. Remembering $\mathrm{PEO}(u) > \mathrm{PEO}(w_k)$, we conclude $u \in N_h(w_k)$. □

Lemma 3. *Let $A \subset V$ such that $G[A]$ is connected. Let $x = highest(A)$ and $z = lowest(N(A))$. Then if* $\mathrm{PEO}(x) < \mathrm{PEO}(z)$, $N(A) = N_h(x)$.

Proof. First, note that since $x = highest(A)$, $N_h(x) \cap A = \phi$ and therefore, $N_h(x) \subseteq N(A)$. Now we will prove $N(A) \subseteq N_h(x)$, from which we can conclude $N(A) = N_h(x)$. Let $y \in N(A)$. Then there is a $w \in A$ such that $y \in N(w)$. Also $\mathrm{PEO}(y) \geqslant \mathrm{PEO}(z) > \mathrm{PEO}(x)$. Consider a chordless path $P = (w, \ldots, x)$, which is completely in $G(A)$. Such a path exists, since $G(A)$ is connected. Also since $x = highest(P)$, by Corollary 1, P is an increasing path. Then by Lemma 2, $y \in N_h(x)$. Thus, we conclude that $N(A) \subseteq N_h(x)$. It follows that $N(A) = N_h(x)$. □

Kumar and Madhavan [12] proves the following characterisation of the minimal separators of chordal graphs. (It is presented slightly differently in [12].) A different proof of this characterisation appears in the preliminary version of the present paper also [3].

Theorem 1 (*Characterisation of minimal separators*). *S is a minimal separator of a chordal graph G if and only if there exist two vertices $a, b \in V$, such that* $\mathrm{PEO}(a) < \mathrm{PEO}(b)$, *and $S = N_h(a) \subseteq N(b)$.*

3. The algorithm

Our algorithm examines $N_h(u)$ for each vertex u, and decides whether it is a minimal separator or not. We consider a node w to be a *witness* for $N_h(u)$, if the existence of w proves that $N_h(u)$ is a minimal separator. For example, if we can find a node w with $\mathrm{PEO}(w) > \mathrm{PEO}(u)$ and $N(w) \supseteq N_h(u)$, then by Theorem 1, w is a witness for $N_h(u)$. Thus, the issue in designing the algorithm is to efficiently identify a witness, if one exists.

In our algorithm, we make use of only two types of witnesses. These types are defined in terms of a special node in $N_h(u)$, namely $z = lowest(N_h(u))$.

Definition 5. First type witness: a node w is defined to be a first type witness for $N_h(u)$, iff $w \in N_h(z) - N_h(u)$.

Lemma 4 (*First type witnesses are indeed witnesses*). *If there exists a node $w \in N_h(z) - N_h(u)$, then $N_h(u)$ is a minimal separator.*

Proof. Clearly, since $w \in N_h(z)$, $\mathrm{PEO}(w) > \mathrm{PEO}(u)$. Now note that since $N_h(u)$ is a clique, and $z = \mathit{lowest}(N_h(u))$, we have $N_h(u) - \{z\} \subseteq N_h(z)$. Also $w \in N_h(z)$. Since by definition of a PEO, $\{z\} \cup N_h(z)$ forms a clique, it follows that $N_h(u) \subseteq N(w)$. Then by Theorem 1, w is a witness for $N_h(u)$. □

Lemma 5. *$N_h(u)$ has a first type witness if and only if $d_h(u) \leqslant d_h(z)$.*

Proof. Since $N_h(u) - \{z\} \subseteq N_h(z)$, if $d_h(u) \leqslant d_h(z)$ i.e., if $d_h(u) - 1 < d_h(z)$, clearly there exists a node $w \in N_h(z) - N_h(u)$. On the other hand, if there exists a node $w \in N_h(z) - N_h(u)$ then clearly $d_h(z) > |N_h(u) - \{z\}| = d_h(u) - 1$ and thus $d_h(u) \leqslant d_h(z)$. □

Definition 6. Second type witness: a node $w \neq u$ is defined to be a second type witness for $N_h(u)$ iff $w \in N_l(z)$, and $N_h(w) = N_h(u)$.

Lemma 6 (*Second type witnesses are indeed witnesses*). *If there exists $w \neq u$, $w \in N_l(z)$, such that $N_h(w) = N_h(u)$, then $N_h(u)$ is a minimal separator.*

Proof. If $\mathrm{PEO}(w) > \mathrm{PEO}(u)$, clearly since $N_h(u) = N_h(w) \subseteq N(w)$, by Theorem 1, $N_h(u)$ is a minimal separator and w is a witness for $N_h(u)$. If $\mathrm{PEO}(w) < \mathrm{PEO}(u)$, we just have to interchange the roles of u and w, and we again get $N_h(w) = N_h(u)$ is a minimal separator. □

The first and second type witnesses certainly do not exhaust the set of all possible witnesses. But it turns out that these two types are sufficient for our algorithm. This is because of the following lemma, which assures that, if first type witnesses are not available, then a second type witness is guaranteed to exist, provided the $N_h(u)$ in question is a minimal separator.

Lemma 7. *Let $N_h(u)$ be a minimal separator. If $N_h(z) - N_h(u) = \emptyset$, then there exists a node $w \neq u$, such that $w \in N_l(z)$, and $N_h(w) = N_h(u)$.*

Proof. Since $N_h(u)$ is a minimal separator, there exists a node $v \in V$, such that $N_h(u)$ minimally separates u from v. (If $N_h(u)$ is a minimal $(v_1 - v_2)$ separator, at least one of them, either v_1 or v_2, should be in a different connected component than that of u, when $N_h(u)$ is removed from G. Then clearly $N_h(u)$ *minimally* separates u from that node.) Let A be the connected component of $G[V - N_h(u)]$ which contains v. Clearly, $u \notin A$. First, note that by minimality of the separator $N_h(u)$, $N_h(u) = N(A)$. Let $x = \mathit{highest}(A)$. We claim that $\mathrm{PEO}(x) < \mathrm{PEO}(z)$. Otherwise, if $\mathrm{PEO}(x) > \mathrm{PEO}(z)$, consider a chordless path $P = (z, w_1, \ldots, x)$ in $G(A \cup \{z\})$, which is guaranteed to exist since $G(A \cup \{z\})$ is connected. Then clearly $w_1 \in N_h(z)$, by Corollary 2. Also $w_1 \notin N_h(u)$.Thus $w_1 \in N_h(z) - N_h(u)$, contradicting the assumption that $N_h(z) - N_h(u) = \emptyset$. We infer that $\mathrm{PEO}(x) < \mathrm{PEO}(z)$. We conclude from Lemma 3, that $N_h(x) = N(A) = N_h(u)$. Also clearly $x \in N_l(z)$ and the lemma follows. □

Finally, the reason why we consider these special kind of witnesses, namely the second type witnesses, is exactly that, they are easy to identify. The following lemma explains this.

Lemma 8. *Suppose that $N_h(z) - N_h(u) = \emptyset$. A node $w \neq u$ is a second type witness if and only if $d_h(w) = d_h(u)$ and $\mathit{lowest}(N_h(w)) = z$.*

Proof. If w is a second type witness i.e., if $N_h(w) = N_h(u)$, clearly we have $d_h(u) = d_h(w)$ and $\mathit{lowest}(N_h(w)) = z$. Now suppose that there is a node w with $d_h(u) = d_h(w)$ and $\mathit{lowest}(N_h(w)) = z$. We first show that $N_h(w) \subseteq N_h(u)$. Let $y \in N_h(w)$. If $y = z$ then y is in $N_h(u)$. Otherwise, noting that by the assumption $z = \mathit{lowest}(N_h(w))$, $z \in N_h(w)$ also, we get $(z, y) \in E$, since $N_h(w)$ should form a clique. But $z = \mathit{lowest}(N_h(w))$, therefore $\mathrm{PEO}(y) > \mathrm{PEO}(z)$, i.e., $y \in N_h(z)$. Now if $y \notin N_h(u)$, then $y \in N_h(z) - N_h(u)$, which contradicts the assumption that $N_h(z) - N_h(u) = \emptyset$. Thus $y \in N_h(u)$. It follows that $N_h(w) \subseteq N_h(u)$. But since $d_h(u) = d_h(w)$, it has to be the case that $N_h(u) = N_h(w)$. That is, w is a second type witness of $N_h(u)$. □

3.1. Algorithm to list minimal separators

1. Find a PEO of G.
2. (*Preprocessing*:) For each $u \in V$, find $d_h(u)$. If $d_h(u) \neq 0$ then $z_u = lowest(N_h(u))$. Prepare the list $N_h(u)$, and store these informations with the node, for future use.
3. (*Initialisations*:)
   for $i = 1$ to n do:
   (a) $A[i] = NULL$
   (b) $X[i] = False$
4. for each $u \in V$ do:
   If $d_h(u) \neq 0$ then
   If $d_h(u) \leqslant d_h(z_u)$ then {First type witness found}
   Output $(N_h(u))$
   else
   add u to $A[z_u]$.
5. For each $z \in V$ do:
   (a) For each $u \in A[z]$ do:
   if $X[d_h(u)]$ then {Second type witness found}
   output $N_h(u)$
   else
   $X[d_h(u)] = True$
   (b) For each $u \in A[z]$ do $X[d_h(u)] = False$

Theorem 2. *The above algorithm outputs exactly the set of all minimal separators.*

Proof. First, note that the algorithm outputs $N_h(u)$ only if a witness is found: if it is output at step 4, it is because a first type witness is found (by Lemma 5). If it is output at step 5(a), it is because a second type witness is found. (Note that at this step, $X[d_h(u)]$ is set to True only if another node $w \in A[z]$ was already encountered, such that $d_h(w) = d_h(u)$. Moreover, u is added to $A[z]$ only if there is no first type witness for $N_h(u)$ i.e., only if $N_h(z_u) - N_h(u) = \emptyset$. We also have $z_w = z = z_u$, since both w and u belong to $A[z]$. Thus, w is a second type witness for $N_h(u)$ by Lemma 8.)

Now every minimal separator is $N_h(u)$ for some $u \in V$, by Theorem 1, and has either a first type witness, in which case it is output at step 4, or by Lemma 7 a second type witness w such that $w \neq u$, $z_w = z_u$ and $d_h(w) = d_h(u)$. If $N_h(u)$ is not output at step 4, $d_h(u) > d_h(z)$ and thus $d_h(w) > d_h(z)$ also. It follows that both u and w are added to $A[z_u]$. Then, $N_h(u)$ is output at step 5(a) when processing u if u is processed after w and when processing w otherwise. Therefore, every minimal separator will be output. □

Theorem 3. *The above algorithm runs in* $O(m + n)$ *time.*

Proof. Steps 1–4 involve traversing the neighbours of each vertex at most a constant number of times. Thus, since $\sum_{u \in V} d(u) = 2m$, the overall time requirement of these steps is $O(m+n)$. Step 5 runs in time $O\left(\sum_{z \in V} \sum_{u \in A[z]} |N(u)|\right)$ time. Note that each vertex u belongs to at most one list $A[z]$, since for each u there is at most one z_u. Thus, $\sum_{z \in V} \sum_{u \in A[z]} |N(u)| \leqslant \sum_{u \in V} |N(u)| \leqslant 2m$. Thus the algorithm has overall time complexity $O(m + n)$. □

4. Discussion on removing the duplicates

The Definition of the problem REMOVE–DUPLICATES: The algorithm as described in the previous section has the following drawback: suppose that a minimal separator M is such that $M = N_h(u_1) = N_h(u_2) = N_h(u_3) = \cdots = N_h(u_k)$ for k different vertices. Then, our algorithm may output M, k times.

Let S be the multiset of minimal separators output by the algorithm. Our intention is to design a linear time procedure REMOVE–DUPLICATES(S), which takes the multiset S as input and outputs a set S' in which each minimal separator appears once and only once.

We would like to mention that just like our algorithm, the algorithm of Kumar and Madhavan [12] also suffers from the above-mentioned problem: the same minimal separator may be output more than once. In their paper they have suggested that the minimal separators can be stored in a balanced binary search tree, so that the duplicates can be removed efficiently. But this method takes overall $O(\omega(G)|V|\log|\mathscr{B}|+|E|)$ time where $\omega(G)$ denotes the maximum clique size and $|\mathscr{B}|$ denotes the number of minimal separators (see [12, p. 167]) and therefore it is not consistent with the linear time complexity of their listing algorithm. The solution we suggest below is different: in fact the method to remove duplicates described here is independent of the specific listing algorithm used (except for an inconsequential assumption on the format in which the vertices of the minimal separator appear in the corresponding list). Thus, the drawback of the algorithm of [12] also can be corrected, if it is used in conjunction with the REMOVE–DUPLICATES procedure developed in this section.

The format of the input for REMOVE–DUPLICATES: We can assume that the multiset S is stored as an array of lists, where each list represents a minimal separator. Note that such an array of lists takes only $O(m+n)$ space. This is because, for each minimal separator S and any node u, S is output at most once as $N_h(u)$, and $\sum_{u\in V}|N_h(u)|=m$.

We assume that the vertices of a minimal separator appear in its list, in increasing order of their identifying numbers. To ensure that the minimal separators are output in this way, we may have to do the following transformation on the adjacency list representing the graph, before the listing algorithm of Section 3.1 starts executing: rearrange the adjacency list such that for each vertex u, its neighbours appear in the increasing order of their identifiers in the corresponding list. We leave it to the reader to convince himself/herself that this can be achieved in $O(m+n)$ time. Once this is done, it is easy to ensure that each list in the array of lists storing the sets $N_h(u)$, prepared at step 2 (of the algorithm of Section 3.1), satisfies the same property.

Reducing REMOVE–DUPLICATES(S) to an easier problem Remove–Duplicates(S_i): Note that each list representing a minimal separator can be thought of as a string, consisting of characters from the alphabet $\{1,2,\ldots,n\}$ (i.e., the set of identifiers of vertices). Note that in our case, each of these strings is formed of distinct characters, and by the assumption on the input format described in the previous paragraph, the characters of the string appear in increasing order. Thus, a given minimal separator M corresponds to the unique string obtained by arranging the vertices of M in the increasing order of their identifiers. Thus, the question of removing the duplicates of a certain minimal separator reduces to the problem of removing the duplicates of the corresponding string.

Let S_i be the multiset of strings with exactly i characters. First, note that if two strings A and B are duplicates of each other, then both have the same number of characters. Thus, the procedure REMOVE–DUPLICATES(S) can be implemented as follows.

REMOVE–DUPLICATES(S)
for $i=1$ to n do: Remove–Duplicates(S_i)

We leave it to the reader to verify that partitioning the strings of S into the subsets S_i can be done in $O(m+n)$ time. Let n_i be the number of strings in S_i (with repetitions counted). Also let $m_i=i\cdot n_i$ be the total number of characters (with all the repetitions counted) in the strings of S_i. If we can implement Remove–Duplicates(S_i) in $O(m_i+n_i)$ time, REMOVE–DUPLICATES(S) can be achieved in $O(m+n)$ time, since $\sum_i m_i\leqslant m$ and $\sum_i n_i\leqslant n$.

An intuitive approach: How do we remove duplicates from the multiset S_i? The most intuitive approach would be to sort the strings of S_i in lexicographic order: then the copies of the same string appear contiguously and to identify the duplicates, one just has to compare each string with the next one in the sequence. Clearly, the latter step can be achieved in $O(m_i+n_i)$ time. What about the first step?

Let us consider the following Radix sort algorithm, to sort the strings in S_i. (See p. 178, Chapter 9, Section 9.3 of [5], to see a discussion on the RADIX–SORT algorithm.)

RADIX–SORT(S_i)
for $j=i$ down to 1 do:
use a stable sort to sort the strings of S_i based on the jth character.

A stable sort of an array of strings based on the jth character is a sort of this array based on the jth character such that strings with the same jth character appear in the output array in the same order as they do in the input array. Such

a sort can be achieved, if we use the sorting procedure named COUNTING–SORT (given in Chapter 9, Section 9.2 of [5]) to sort the array of characters formed by the jth characters of the strings, and then order the strings accordingly. Given an array A of α characters, and if each character is guaranteed to be in the range 1 to k, then COUNTING–SORT procedure sorts the characters in $O(\alpha + k)$ time. Thus, the overall time complexity of RADIX–SORT(S_i) would be $O(i(\alpha + k))$. In our case, $\alpha = n_i$ and $k = n$, and thus the radix sort works in $O(m_i + i \cdot n)$ time. Unfortunately, the second term $i \cdot n$ is too big, since i can be $\Omega(n)$.

But notice that for our purpose, namely to remove the duplicates, it is not necessary to get a sorted list of strings: rather we need only to somehow ensure that the copies of the same string appear contiguously in the output list of strings. Taking advantage of this relaxed requirement, we can solve the above-mentioned problem by mapping the alphabet to a smaller one, as explained below.

Mapping the alphabet to a smaller one: Note that in the radix sort algorithm the COUNTING–SORT procedure is invoked once for each character position. Let us call each invocation of COUNTING–SORT a "phase" of the RADIX–SORT. Though the actual alphabet size is n, it is clear that the number of characters occurring during a given phase of the RADIX–SORT is at most n_i, since there are only n_i strings in S_i. So we can define a partial function f_j from $\{1, 2, 3, \ldots, n\}$ to $\{1, 2, \ldots, n_i\}$ which maps the relevant characters of the bigger alphabet to the characters of a smaller alphabet. The partial function f_j which corresponds to the jth phase of RADIX–SORT can be implemented using an array of n elements. This array is initialised with 0 at the beginning of the REMOVE–DUPLICATES(S) in $O(n)$ time. The relevant elements of this array are set (to a character from the smaller alphabet) at the beginning of each phase and reset at the end of that phase. Clearly, the setting and resetting can be done in $O(n_i)$. (See the pseudocode at the end of this section for the details of implementing f_j using an array.) Now this modified RADIX–SORT(S_i) can run in $O(m_i + i \cdot n_i) = O(m_i)$ time.

The output sequence of strings—unsorted, but good enough: During the jth phase of the RADIX–SORT(S_i), each time we encounter the character x, we use the character $f_j(x)$ instead. But note that this mapping may not maintain the relative values of the characters: that is, it is not guaranteed that $f_j(x_1) > f_j(x_2)$ whenever $x_1 > x_2$. Moreover, the relative values of the characters may change from phase to phase: i.e., it is possible to have $f_{j_1}(x_1) > f_{j_1}(x_2)$ and $f_{j_2}(x_1) < f_{j_2}(x_2)$ for $j_1 \neq j_2$. Thus, if we run the modified RADIX–SORT on a multiset of strings, the list of strings output is not guaranteed to be sorted. But it is easy to verify that the following property is guaranteed. A string $(a_1, a_2, \ldots, a_i)$ appears earlier in the output list than a different string $(b_1, b_2, \ldots, b_i)$ if and only if $f_\ell(a_\ell) < f_\ell(b_\ell)$ where ℓ is the first character position where the two strings differ. From this property, it is easy to infer that all the copies of the same string appear contiguously in the sequence of strings output by the algorithm.

Thus, the Remove–Duplicates(S_i) procedure described above removes all the duplicates in S_i in $O(m_i)$ time. It follows that REMOVE–DUPLICATES(S) runs in $O(m + n)$ time.

Finally, we summarise REMOVE–DUPLICATES(S) below, leaving out the trivial details:

REMOVE–DUPLICATES(S)

1. (Initialise the array F:) for $j = 1$ to n do: $F[j] = 0$.
2. Form the sets S_i for each $1 \leqslant i \leqslant n$.
3. for $i = 1$ to n do: {Remove–Duplicates(S_i) for each i:}
   (a) for $j = i$ down to 1 do: {modified RADIX–SORT for a given S_i}
      i. (Defining f_j:) count = 0
         For each string $s \in S_i$ do:
            If $F[s[j]] = 0$ then
               count = count + 1
               F[s[j]] = count
      ii. Use COUNTING–SORT (see Chapter 9 of [5]) to sort the strings of S_i based on F[jth character].
      iii. (Clearing f_j:) For each string s of S_i do: $F[s[j]] = 0$.
   (b) for $j = 1$ to $n_i - 1$ do: If the jth string is the same as the $(j + 1)$th string then remove the jth string.

Acknowledgements

The authors thank both the anonymous referees for pointing out a mistake in the complexity analysis in the earlier version, for pointing out that a duplication removing procedure, which is consistent with the linear time complexity

of the algorithm is required and for suggesting non-trivial simplifications for the definitions of the witnesses. The first author recalls that he had tried to get some simplifications for the definition of the witnesses, but at that time, had failed to notice the subtilities now pointed out by the referee. Even after the first revision, one of the referees suggested non-trivial simplifications for the algorithm, making it look much more elegant. We thank the meticulous refereeing.

References

[1] H. Ariyoshi, Cut-set graph and systematic generation of separating sets, IEEE Trans. Circuit Theory 19 (1972) 233–240.
[2] A. Berry, J.P. Bodat, O. Cogis, Generating all the minimal separators of a graph, Internat. J. Foundations Comput. Sci. 11 (2000) 397–404.
[3] L.S. Chandran, A linear time algorithm for enumerating all the minimum and minimal separators of a chordal graph, in: Proceedings of the Seventh International Computing and Combinatorics Conference, Lecture Notes in Computer Science, vol. 2108, Springer, Berlin, 2001, pp. 308–317.
[4] L.S. Chandran, L. Ibarra, F. Ruskey, J. Sawada, Generating and characterizing the perfect elimination orderings of a chordal graph, Theoret. Comput. Sci. 307 (2003) 303–317.
[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT press, Cambridge, MA, London, England, 1990.
[6] L.A. Goldberg, Efficient Algorithms for Listing Combinatorial Structures, Cambridge University Press, Cambridge, 1993.
[7] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.
[8] A. Kanevsky, Finding all minimum size separating vertex sets in graphs, Networks 23 (1993) 533–541.
[9] T. Kloks, Treewidth: Computations and Approximations, Lecture Notes in Computer Science, vol. 842, Springer, Berlin, 1994.
[10] T. Kloks, H. Bodlaender, D. Kratsch, Treewidth and pathwidth of permutation graphs, SIAM J. Discrete Math. 8 (1995) 606–616.
[11] T. Kloks, D. Kratsch, Listing all minimal separators of a graph, SIAM J. Comput. 27 (1998) 605–613.
[12] P.S. Kumar, C.E.V. Madhavan, Minimal vertex separators of chordal graphs, Discrete Appl. Math. 89 (1998) 155–168.
[13] R.H. Mohring, Graph problems related to gate matrix layout and PLA folding, in: Computational Graph Theory, Springer, Wein, New York, 1990, pp. 17–52.
[14] D.J. Rose, R.E. Tarjan, G.S. Leuker, Algorithmic aspects of vertex elimination in graphs, SIAM J. Comput. 5 (1976) 266–283.
[15] H. Shen, W. Liang, Efficient enumeration of all minimal separators in a graph, Theoret. Comput. Sci. 180 (1997) 169–180.
[16] D.R. Shier, Some aspects of perfect elimination orderings in chordal graphs, Discrete Appl. Math. 7 (1984) 325–331.
[17] M. Yannakakis, Computing the minimum Fill–in is NP–complete, SIAM J. Algebra Discrete Math. 2 (1981) 77–79.
[18] M. Yannakakis, R.E. Tarjan, Simple linear time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs, SIAM J. Comput. 13 (1984) 566–579.