Efficient Implementation of a Minimal Triangulation Algorithm

Pinar Heggernes and Yngve Villanger

Department of Informatics, University of Bergen, N-5020 Bergen, Norway
{pinar, yngvev}@ii.uib.no

Abstract. LB-triang, an algorithm for computing minimal triangulations of graphs, was presented by Berry in 1999 [1], and it gave a new characterization of minimal triangulations. The time complexity was conjectured to be $O(nm)$, but this has remained unproven until our result. In this paper we present and prove an $O(nm)$ time implementation of LB-triang, and we call the resulting algorithm LB-treedec. The data structure used to achieve this time bound is tree decomposition. We also report from practical runtime tests on randomly generated graphs which indicate that the expected behavior is even better than the proven bound.

1 Introduction

Many important graph problems are concerned with adding edges to a given arbitrary graph to obtain a chordal supergraph, and the resulting chordal graph is called a *triangulation* of the given graph. A triangulation H of G is *minimal* if no subgraph of H is a triangulation of G. The first algorithms for computing minimal triangulations appeared more than 25 years ago [12], [15] with $O(nm)$ time complexity, where n is the number of vertices and m is the number of edges of the input graph. After a time gap of 20 years, the problem began to be restudied [3], [4], [7], [14] in the following sandwich version: given an arbitrary triangulation H of G, compute a minimal triangulation M of G with $G \subseteq M \subseteq H$. The best known theoretical time complexity of computing minimal triangulations, with or without the sandwich property, has remained as $O(nm)$.

Algorithm LB-triang [1] presented a new characterization of minimal triangulations, and also solved the sandwich problem, allowing the order in which the vertices are processed as input. The time complexity of LB-triang was conjectured to be $O(nm)$, however this remained unproven since its presentation. (The straight forward implementation suggested in [1] requires $O(nm')$ time, where m' is the number of edges in the resulting triangulation).

In this paper, we prove that Algorithm LB-triang can be implemented in $O(nm)$ time using a a tree decomposition of the input graph. In the start the tree decomposition consists of only one tree node containing all the vertices of G. At each step, the tree decomposition is refined by encountering and inserting minimal separators of G into the data structure. At the end, the data structure

contains a clique tree of the computed minimal triangulation of G. We call this new $O(nm)$ time and $O(m')$ space algorithm LB-treedec. In addition to a theoretical time complexity analysis of our algorithm, we also present runtime results from a practical implementation.

This extended abstract is organized as follows. In the next section we give the necessary background. Section 3 presents Algorithm LB-treedec and contains the main results of this paper. Section 4 concludes the paper and mentions open questions and future research directions.

2 Background

A graph is denoted $G = (V, E)$, with $n = |V|$, and $m = |E|$. All graphs that we work on are connected and simple. $G(A)$ is the subgraph induced by a vertex set $A \subset V$, but we often denote it simply by A when there is no ambiguity. A *clique* is a set of vertices that are all pairwise adjacent. For all the following definitions, we will omit subscript G when it is clear from the context which graph we work on. The *neighborhood* of a vertex x in G is $N_G(x) = \{y \neq x \mid xy \in E\}$. The neighborhood of a set of vertices A is $N(A) = \cup_{x \in A} N(x) - A$, and we define $N[A] = N(A) \cup A$.

For a connected graph $G = (V, E)$ with $X \subseteq V$, $\mathcal{C}_G(X)$ denotes the set of connected components of $G(V - X)$. $S \subset V$ is called a *separator* if $|\mathcal{C}(S)| \geq 2$, an *xu-separator* if vertices x and u are in different connected components of $\mathcal{C}(S)$, a *minimal xu-separator* if S is an xu-separator and no proper subset of S is an xu-separator, and a *minimal separator* if there is some pair $\{x, u\}$ such that S is a minimal xu-separator. Equivalently, S is a minimal separator if there exist C_1 and C_2 in $\mathcal{C}(S)$ such that $N(C_1) = N(C_2) = S$. Two separators S and S' are *crossing* if there exist two components $C_1, C_2 \in \mathcal{C}(S)$, $C_1 \neq C_2$, such that $S' \cap C_1 \neq \emptyset$ and $S' \cap C_2 \neq \emptyset$.

A *chord* of a cycle is an edge connecting two non-consecutive vertices of the cycle. A graph is *chordal*, or *triangulated*, if it contains no chordless cycle of length ≥ 4. Edges that are added to an arbitrary graph G to obtain a triangulation H of G are called *fill* edges, with $m' = |E(H)|$.

Theorem 1. (Lekkerkerker and Boland [11]) *A graph G is chordal iff for every vertex x in G, every minimal separator contained in $N_G(x)$ is a clique.*

Algorithm LB-triang (Berry [1])
Input: A graph $G = (V, E)$, and an order $v_1, v_2, ..., v_n$ on the vertices of G.
Output: A minimal triangulation $H = (V, E + F)$ of G.
begin
  $H = G$; $F = \emptyset$;
  for $i = 1$ **to** n **do**
    for each connected component C in $\mathcal{C}_G(N_H[v_i])$ **do**
      Make $N_G(C)$ into a clique by adding edge set F';
      $F = F + F'$; $H = (V, E + F)$;
end

It is shown in [2] that the set of minimal separators included in the neighborhood of a vertex x is exactly $\{N(C) \mid C \in \mathcal{C}(N[x])\}$. Thus with help of Theorem 1 it can easily be shown that LB-triang produces a triangulation. The proof that the computed triangulation is minimal relies on results from [10] and [13], and involves showing that the computed minimal separators are pairwise non-crossing. It should be noted that the set of minimal separators of G processed in this way by LB-triang is exactly the set of all minimal separators of the resulting minimal triangulation H.

The interesting part for the purpose of this paper is the time complexity of LB-triang. It is mentioned in [1] that $\mathcal{C}_H(S) = \mathcal{C}_G(S)$ and $N_H(C) = N_G(C)$ for each $C \in \mathcal{C}_G(S)$; thus the component search can be done in G rather than in H, as described in Algorithm LB-triang. Despite this, proving an $O(nm)$ time complexity for LB-triang turned out to be a more difficult task than first expected. The most time consuming part is making the encountered minimal separators into cliques by adding fill edges. These fill edges are not needed in the component search, but they are needed in finding $N_H[v_j]$ at later steps j. The problem is that each minimal separator, and thus each fill edge, can be encountered and inserted several times, which gives the need to keep a sorted list of the minimal separators or the fill edges so that redundant copies can be removed. Such an approach requires $O(nm')$ time.

Our algorithm does not store the edges of each computed minimal separator, but rather stores the minimal separators as vertex sets, since these are all cliques. Computing $N_H[v_i]$ is not straight forward in this setting, and requires scanning of some of the computed minimal separators. Our practical implementation of Algorithm LB-triang relies on two important structures, called tree decompositions and clique trees.

Definition 1. *A* tree-decomposition *of a graph $G = (V, E)$ is a pair*

$$(\{X_i \mid i \in I\},\ T = (I, M))$$

where $\{X_i \mid i \in I\}$ is a collection of subsets of V, and T is a tree, such that:
- $\bigcup_{i \in I} X_i = V$,
- $(u, v) \in E \Rightarrow \exists i \in I$ *with* $u, v \in X_i$, *and*
- *for all vertices $v \in V$, $\{i \in I \mid v \in X_i\}$ induces a connected subtree of T.*

Thus each tree node corresponds to a vertex subset X_i, also called a *bag* (in which the graph vertices are placed). We will not distinguish between vertex subsets and their corresponding tree nodes. Consequently, we will refer to the tree T when we mention the corresponding tree decomposition. More about tree decompositions and their importance can be found in [6]. In our implementation, we will let each edge (X, Y) of T contain the set of vertices in $X \cap Y$. Thus we will often refer to edges of T also as vertex subsets.

For chordal graphs, tree decompositions exist where the bags are exactly the maximal cliques of the graph [8]. Such tree decompositions are called *clique trees* [5]. One important property of clique trees which is related to our implementation is the following.

Lemma 1. (Ho and Lee [9]) *Let T be a clique tree of a chordal graph G. Every edge of T is a minimal separator of G, and for every minimal separator S in G, there is an edge $(K_i, K_j) = K_i \cap K_j = S$ in T.*

A chordal graph has at most n maximal cliques and $n-1$ minimal separators, and hence the number of nodes and edges in a clique tree is $O(n)$ [9].

3 LB-treedec: an $O(nm)$ time implementation of LB-triang

At each step i, Algorithm LB-triang identifies and makes into cliques the minimal separators of G (and H) included in $N_H(v_i)$, where H is the partially filled graph so far. In our implementation these computed minimal separators are stored as vertex lists. Thus computing $N_H(v_i)$ is not straight forward as the fill edges are not actually added. However, at the beginning of step i, it is sufficient to consider $N_G(v_i)$ and the minimal separators computed so far, in order to compute $N_H(v_i)$, since an edge is a fill edge of the transitory H if and only if its endpoints belong to a previously computed separator. Our general approach will be as follows. We start with a tree decomposition T of G consisting of only one bag containing all the vertices of G. At each step, whenever we encounter a new minimal separator S, we check whether S can be inserted into T as an edge to refine the tree decomposition. We will show that if S can be inserted then there is only one bag containing the vertices belonging to S and the vertices that S minimally separates. We split this bag into two bags, insert the separator as an edge between the two new bags, correctly couple the two new bags to the rest of the tree through the neighbors of the old bag, and maintain in this way a tree decomposition of G where the bags get smaller and smaller. This is the intuition behind the implementation, and we will now give the formal details. For the following discussions, let T_x denote the subtree of T induced by all tree nodes containing graph vertex x.

3.1 Data structures and implementation details

We will first rewrite Algorithm LB-triang to operate on the tree decomposition data structure. The new algorithm that thus results is called LB-treedec, where the neighbors of a vertex are found through original edges and the already computed separators, using the tree structure to extract this information. At step i of the algorithm, let $U(i)$ be the union of all minimal separators computed at earlier steps containing vertex v_i, and let U_A be the union of all minimal separators computed so far. Every vertex in $U(i)$ is a neighbor of v_i in the transitory graph H since these minimal separators are cliques in H. Fill edges of H appear only within minimal separators, thus $U(i) \cup N_G(v_i)$ is the set of all neighbors of v_i at step i, including v_i itself if $U(i) \neq \emptyset$. In fact, it can be shown that no fill edge of H incident to v_i is created after step i, thus $N_H[v_i] = U(i) \cup N_G(v_i) \cup \{v_i\}$. Consequently, at every step i, the final adjacency set of v_i in H is computed, and can be inserted into H directly.

Algorithm LB-treedec
Input: A graph $G = (V, E)$, and an order $v_1, v_2, ..., v_n$ on the vertices of G.
Output: A minimal triangulation $H = (V, E + F)$ of G, and a clique tree T of H;
begin
$H = G$; $T = (\{V\}, \emptyset)$; $U_A = \emptyset$;
for $i = 1$ **to** n **do**
(1) Compute $U(i)$ using the separator information stored in the edges of T;
$N_H[i] = N_G(v_i) \cup U(i) \cup \{v_i\}$;
for each connected component C in $\mathcal{C}_G(N_H[i])$ **do**
$S = N_G(C)$; $U_A = U_A \cup S$;
(2) **if** there is a bag X in T containing v_i and S and a subset of C **then**
$X_1 = S \cup (X \cap C)$; $X_2 = X \cap (G - C)$;
(3) Replace X with X_1, X_2, and edge $(X_1, X_2) = S$ in T;
end

Steps of this algorithm marked as (1) - (3) need to be explained further. Step (1) is more involved, and we will describe this step last. For the time being, assume that the neighborhood of v_i in H at step i of the algorithm is correctly found.

Let $S = N_G(C)$ for a component $C \in \mathcal{C}_G(N_H[i])$ found at step i. S is a minimal separator of G and of H, separating v_i and a subset of C. If there is a tree node X containing v_i and S and any vertex of C, then this tree node can be split into two tree nodes and S can be inserted as an edge between the new tree nodes.

Lemma 2. *Let $S = N_G(C)$ for a component $C \in \mathcal{C}_G(N_H[i])$ found at step i. Then there is at most one bag X in the current tree T containing both $(\{v_i\} + S)$ and a subset of C.*

Proof. Remember first that S is a minimal separator separating v_i from C. For every vertex $u \in C$, we know that no previously computed minimal separator S' contains both u and v_i. Otherwise S and S' would be crossing separators, and this would contradict the correctness of Algorithm LB-triang.

Assume now on the contrary that there are two bags X_1 and X_2 that both contain v_i and S and at least one vertex belonging to C. Let u be a vertex of C that belongs to X_1 and x a vertex of C that belongs to X_2. If $u = x$, then we have a contradiction since every tree edge between X_1 and X_2 is a minimal separator containing u and v_i. Thus $u \in X_1 - X_2$, and $x \in X_2 - X_1$. No tree edge on the path between X_1 and X_2 in T contains both u and x, and every tree edge on this path contains S and v_i. Since u and x do not appear together in any bag on this path, there must exist at least one edge S' that does not contain any of u and x. This means that there is a minimal ux-separator S' such that $(\{v_i\} + S) \subseteq S'$. Since S' separates u and x in G, S' must contain some vertex of C. Since S' contains both v_i and a vertex of C, S and S' are crossing, leading to the desired contradiction.

Invariant 1. *At each step of the algorithm, the tree T is a tree decomposition of G and of the resulting minimal triangulation H.*

Proof. We will prove this invariant by induction. The base case is true since a tree with only one tree node containing all vertices of G is a tree decomposition of both G and H.

Let $S = N_G(C)$ for a component $C \in \mathcal{C}_G(N_H[i])$ found at step i. If there is no bag containing v_i and S and a subset of C, no changes need to be done in the tree, and we have still a tree decomposition. Otherwise, let X be the bag of T containing v_i and S and a subset of C. Assume that the invariant is true for T. We will show, by explaining how X is replaced by X_1, X_2 and the edge $(X_1, X_2) = S$, that the tree T' resulting from this operation is a tree decomposition of G and H.

After the tree node X is removed and the tree nodes X_1 and X_2 are inserted, the tree nodes that were previously incident to X must be connected to X_1 or X_2 instead. It can be easily shown that for any edge $(X, Y) = X \cap Y = S'$ previously incident to X, S' is a subset of X_1 or X_2 or both. Thus we simply connect each such edge to that of the new tree nodes that it is a subset of (arbitrary one can be chosen if it is a subset of both; a good idea is to choose the one that results in a smaller diameter of the tree). We leave it to the reader to verify that the new tree T' fulfills the requirements of a tree decomposition, both for G and H.

We will now describe how to search for the tree node X efficiently. We start from a tree node U containing a vertex u of C and do a depth first search until we find a tree node X that contains v_i. If X also contains a vertex of C, then by Lemma 2 we have found the unique tree node that we want to split. If X does not contain a vertex of C, then by the connected subtree property of a tree decomposition, we know that no tree nodes of T further away from U can contain a vertex of C, and we can conclude that the tree does not need to be updated. Every graph vertex u has a pointer to a tree node that contains u. Thus the starting point of the search is decided in constant time. We will in the next section prove that searching for v_i in a tree node at step i is an amortized constant time operation. In addition, the starting points of the searches at step i belong to disjoint components of G, and thus the searches can be done in disjoint subtrees of T by marking the already traversed paths and adding pointers as shortcuts. What now remains to be explained is how to decide whether or not X contains a vertex of C.

Let $S = N_G(C)$ for a component $C \in \mathcal{C}_G(N_H[i])$ found at step i, and let U be a tree node of T containing a vertex of C. Let $(Y, X) = S'$ be the last edge in a depth first search that starts from U in T and reaches a tree node X containing v_i. Then certainly, X contains a vertex of C iff S' contains a vertex of C. Thus, in addition to the above searches, we also need to check the last edges leading to the tree nodes containing v_i when the searches stop. These edges are edges of T with exactly one endpoint in T_{v_i}. Let us denote this set of tree edges by $Border(i)$. The sum of the sizes of these edges is $O(m)$ as will be shown during the time complexity analysis. Therefore, at each step i we can create characteristic vectors for all these edges in $O(m)$ time and space, and delete these at the end of each step. Then checking each vertex of each component C for membership in the desired $Border(i)$ is a constant time operation per check.

We will now describe how to compute the union $U(i)$. The main reason for using the tree data structure is to be able to compute $N_H(v_i)$ efficiently. Note that T_x is a connected subgraph of T for every x at each step of the algorithm, due to Invariant 1. From the construction of T, it is clear that $U(i)$ can be found by computing the union of all the edges of T_{v_i}. However, this may require $O(m')$ time at each step, which we cannot afford. We have to compute this union in a different way. The main idea is that the sum of the sizes of the edges of T_{v_i} is $O(m')$, whereas the sum of the sizes of the edges belonging to $Border(i)$ is $O(m)$. Now we show how these border edges can be used to compute $U(i)$.

Observation 1. *Let u and v be two vertices of G. A separator S containing both u and v is present as an edge in T iff T_u and T_v share an edge $\equiv S$ in T.*

Observation 1 gives an alternative definition of the desired union: $U(i) = \{u \mid T_u$ shares an edge with $T_{v_i}\}$. We define the following disjoint vertex sets for step i of the algorithm:

- $Inner(i) = \{u \neq v_i \mid$ every edge of T_u is an edge of $T_{v_i}\}$
- $InnerOuter(i) = \{u \mid T_u$ has at least one edge that is an edge of T_{v_i} and at least one edge that is not an edge of $T_{v_i}\}$
- $BorderOuter(i) = \{u \mid T_u$ has no edges in common with T_{v_i} and T_u has a node containing $v_i\}$
- $Outer(i) = \{u \mid T_u$ has no edges in common with T_{v_i} and no node of T_u contains $v_i\}$

We can see that $U(i) = Inner(i) \cup InnerOuter(i)$. The reason why we have partitioned the vertices into these subsets is that some of these subsets are less time consuming to compute than others, and we will therefore not compute $Inner(i) \cup InnerOuter(i)$ directly, but through set operations on the listed subsets. The following connection should be clear: $U_A = Inner(i) \cup InnerOuter(i) \cup BorderOuter(i) \cup Outer(i)$.

$Border(i)$ is the set of edges (X, Y) in T such that X contains v_i, and Y does not contain v_i. $Border(i)$ can be computed readily during the depth first searches of step i. The union of all vertices belonging to the tree edges (minimal separators) in $Border(i)$ gives us exactly $(InnerOuter(i) \cup BorderOuter(i))$. However, we need to separate $InnerOuter(i)$ from $BorderOuter(i)$, and also compute $Inner(i)$, since our goal is to compute $U(i) = Inner(i) \cup InnerOuter(i)$.

Let (X, Y) be an edge in $Border(i)$ where X contains v_i, and let u be a graph vertex belonging to $X \cap Y$. For each such vertex u, we need to decide whether $u \in InnerOuter(i)$ or $u \in BorderOuter(i)$. A naive and straight forward approach would be to scan every edge in T_{v_i} incident to X to decide what kind of subtree T_u is. If u does not appear on any such edge of T_{v_i} then $u \in BorderOuter(i)$. Otherwise, $u \in InnerOuter(i)$. Since we cannot afford to scan all the edges of T_{v_i}, we will avoid this by including enough information in each tree node so that scanning edges that belong to $Border(i)$ is enough. Each tree node X containing vertex u has a variable N_u which is the number of the neighboring tree nodes in

T that also contain u. (See Figure 1.) Another variable S_u is initialized to -1 when X is created. When the edge (X, Y) is being examined, for each vertex u belonging to this edge, if $S_u < i$ then S_u is updated to i, and C_u is updated to equal $N_u - 1$. If $S_u = i$ then C_u is decremented. Clearly, if C_u reaches 0 during the scanning of the edges in $Border(i)$, u belongs to $BorderOuter(i)$. Otherwise u belongs to $InnerOuter(i)$. After all vertices u belonging to the edges of $Border(i)$ are processed, we will have identified the sets $InnerOuter(i)$ and $BorderOuter(i)$.

It remains to compute $Inner(i)$. Observe that $(Inner(i) \cup Outer(i))$ is readily computed since $Inner(i) \cup Outer(i) = U_A - (InnerOuter(i) \cup BorderOuter(i))$. Since every tree node that an $Inner(i)$ vertex belongs to contains v_i, and no tree node that an $Outer(i)$ vertex belongs to contains v_i, $Inner(i)$ is easily separated from $Outer(i)$. The computation of $U(i)$ at step i of LB-treedec is summarized in the following algorithm:

$U(i) = U_A$; $Inner(i) = U(i)$;
Compute T_{v_i} and $Border(i)$ by depth first search in T;
for each edge $S = (X, Y)$ in $Border(i)$ with $v_i \in X$ **do**
  for each u in S **do**
    if $X.S_u < i$ **then**
      $X.S_u = i$; $X.C_u = X.N_u - 1$;
    else
      $X.C_u = X.C_u - 1$;
    if $X.C_u = 0$ **then** Remove u from $U(i)$;
    Remove u from $Inner(i)$;
for each u in $Inner(i)$ **do**
  Let U be any tree node containing u;
  if $v_i \notin U$ **then** Remove u from $Inner(i)$ and $U(i)$;

We use characteristic vectors to implement $U_A, U(i)$, and $Inner(i)$, so that membership can be tested and changed in constant time. Each of these vectors require $O(n)$ space. Observe that $U(i)$ and $Inner(i)$ are cleared and reused at each step of Algorithm LB-treedec.

We have thus explained the details of Algorithm LB-treedec. Note that all the minimal separators of H are inserted into T during the algorithm by the correctness of LB-triang [1]. From this and from the proof of Invariant 1 it follows that T at the end of the algorithm is actually a clique tree of H.

The data structure details of Algorithm LB-treedec are illustrated in Figure 1. Given the discussion of the implementation and of the complexity analysis, it should be clear how the shown structures work. We would like to stress that tree nodes and tree edges are *not* implemented as characteristic vectors in general. Thus each tree node and each tree edge has simply a sorted list of the vertices it contains. To start with the single tree node requires $\Theta(n)$ space. Every time a tree node is split into two tree nodes because of the insertion of a new minimal separator S, the space requirement increases by $O(|S|)$. Since every inserted

minimal separator of H has to be the neighborhood of a distinct component, the sum of the sizes of all the separators of H is $O(m')$. Thus the total space required is $O(m')$ including the mentioned characteristic vectors, assuming that $n \leq m'$.

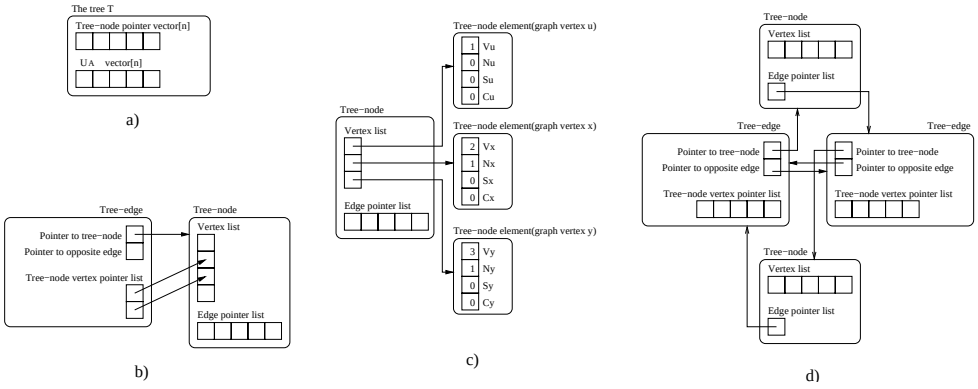

Fig. 1. a) The tree data object where each graph vertex has a pointer to a tree node containing it. b) The interaction between a tree edge and a tree node. c) The relationship between a tree node and the graph vertices contained in it. d) An edge between two tree nodes. Only vectors that end with $[n]$ have lenght exactly n. The length of each of the other lists is the number of vertices it actually contains.

3.2 Time complexity analysis

Observation 2. *The total number of tree nodes created during Algorithm LB-treedec is less than* $2n$.

Lemma 3. *Let* k *be the total number of all requests asking whether or not a given vertex belongs to a given tree node in* T *during Algorithm LB-treedec. Then the total time for all requests is* $O(n^2 + k)$.

Proof. In our data structures, each tree node X contains a sorted list of the vertices that belong to X. In addition, there is a pointer p_X in X that points to the most recently requested element in X. Each time we receive a request to check whether or not v_i belongs to X, we scan the vertices in the locations between the previous position of this pointer until we reach v_i or a vertex v_j with $j > i$. The pointer is moved during this scan. Since element v_i is never requested at other steps than i, each tree node X is scanned exactly once. Following requests involving vertex v_i at step i are handled in constant time by simply checking whether p_X is pointing to v_i or to a larger vertex. When a tree node is split into two new nodes, the pointers of the new tree nodes are moved to the beginning of the lists again. Thus in the worst case, each tree node ever

created during the algorithm is scanned once. Since at most $2n$ tree nodes are created by Observation 2, the total time is $O(n^2 + k)$.

The number of such requests during the whole algorithm in $O(n^2)$ since the depth first tree searches at each step i are done in disjoint subtrees and require $O(n)$ requests asking whether or not v_i belongs to the tree nodes traversed by the searches. $Border(i)$ is also computed during the same search. However, we need to show that the sum of the sizes of the minimal separators (or tree edges) contained in $Border(i) = O(m)$. Recall that at step i, both deciding which tree nodes that contain v_i should be split, and the computation of $U(i)$ require the scanning of $Border(i)$ tree edges.

Let $I = \{x \mid x$ belongs to a tree node of $T_{v_i}\}$. Consider the graph $G(V - I)$. The number of minimal separators belonging to $Border(i)$ is at most the number of connected components of this graph. Remember that each minimal separator S in $Border(i)$ is defined as $S = N_G(C)$ for distinct connected components C of $G(V - I)$. Thus the sum of the sizes of all involved minimal separators is at most twice the number of edges in G, $O(m)$. Consequently, at step i, the scanning of $Border(i)$ edges and the creation of the mentioned characteristic vectors can be done in $O(m)$ time. In addition, at each step i, we might have to check every vertex of every computed component C for membership in the appropriate $Border(i)$ edge. Since these components are disjoint subgraphs of G, and membership test can be done in constant time for $Border(i)$ edges, this is an $O(n)$ operation for each step i.

We have thus proven the following.

Theorem 2. *Algorithm LB-treedec requires $O(nm)$ time.*

3.3 Practical runtime tests

The runtime tests of Algorithm LB-treedec exhibit interesting properties. We have run the code for the implementation on many random graphs and have observed that LB-treedec behaves in an $O(n^2)$ fashion rather than $O(nm)$ in practice. The implementation is coded using C++ and STL, and the practical tests are run on a machine with Intel Pentium III 1GHz processor and 512 MB RAM. Our code can be obtained via anonymous ftp from `ftp.ii.uib.no` at directory `pub/pinar/LB-treedec/`.

A classical $O(nm)$ time algorithm is Lex-M [15]. We have also coded this algorithm in C++, and tested its runtime against our LB-treedec. The first chart in Figure 2 shows the results of these tests. We have tested random graphs, all with the same number of vertices, and increasing the number of edges. The x-axis shows the number of edges of each graph as a percentage of all possible edges. As we can see from the first chart in Figure 2 the runtime of LB-treedec is superior to Lex-M except for very sparse graphs. Our runtime is highest when there are few edges in the input graph. As the input graph gets denser, the runtime decreases.

We have then examined when the maximum runtime occurs, and our tests indicate that the graphs that require the maximum runtime have $O(n)$ edges.

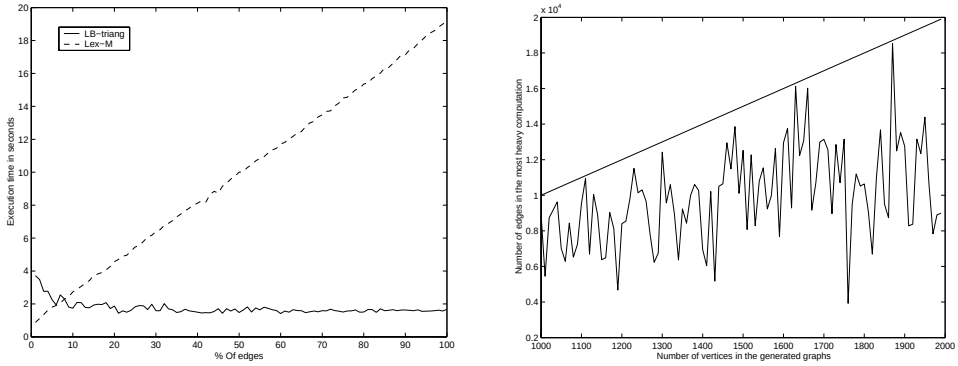

Fig. 2. The chart on the left shows the runtimes of Lex-M and LB-treedec compared on random graphs all with 1000 vertices and increasing number of edges. The chart on the right shows, for numbers $n \in [1000, 2000]$ of vertices, the numbers of edges in the graph on n vertices that requires maximum LB-treedec runtime.

This is illustrated by the second chart of Figure 2. To do this, we generated, for each number n of vertices shown on the x-axis, connected random graphs with all possible numbers of edges, and examined for each n the number of edges in the graph that required the highest runtime. As can be seen from the chart, the worst runtime occurs when the input graph is sparse. The straight line corresponds to $10n$.

4 Conclusion and further work

We have presented Algorithm LB-treedec which is an $O(nm)$ time new version of Algorithm LB-triang. LB-treedec achieves this time bound by utilizing a tree decomposition data structure and carefully placing links between elements that are related. The space complexity of LB-treedec is $O(m')$ where m' is the number of edges of the output graph H. In practical tests, Algorithm LB-treedec exhibits an interesting $O(n^2)$ runtime pattern on randomly generated graphs. In particular, it is faster than the classical Lex-M algorithm of the same theoretical time complexity. However, special graphs on which LB-Treedec requires $\Theta(nm)$ time can be generated, thus an important open problem is whether or not Algorithm LB-triang can be implemented within $O(n^2)$ theoretical time bound. It is still an open question whether or not minimal triangulations can be computed in $O(n^2)$ time in general.

The original Algorithm LB-triang is on-line, which means that the order in which the vertices are processed can be chosen during the course of the algorithm. Algorithm LB-treedec can be made on-line at the cost of more space. If the tree nodes are implemented as characteristic vectors of length n, then membership of each graph vertex in a tree node can be tested in $O(1)$ time. This on-line version requires $O(nm)$ time and $O(n^2)$ space. An interesting question is whether an on-line version can be implemented in $O(nm)$ time and $O(m')$ space.

Every tree decomposition of a graph G corresponds to a triangulation of G. Many important graph theoretical problems search for triangulations with various properties. One of the important NP-hard problems is deciding treewidth, which is equivalent to finding a triangulation where the largest clique is as small as possible. Algorithm LB-treedec gives interesting tools that might be useful for new heuristics for this kind of problems.

Acknowledgment

The authors thank Genevieve Simonet for her useful comments.

References

1. A. Berry. A wide-range efficient algorithm for minimal triangulation. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.
2. A. Berry, J-P. Bordat, and P. Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nordic Journal of Computing*, 7:164–177, 2000.
3. J. R. S. Blair, P. Heggernes, and J. A. Telle. Making an arbitrary filled graph minimal by removing fill edges. In R. Karlsson and A. Lingas, editors, *Algorithm Theory - SWAT '96*, pages 173–184. Springer Verlag, 1996. Lecture Notes in Computer Science 1097.
4. J. R. S. Blair, P. Heggernes, and J. A. Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250:125–141, 2001.
5. J. R. S. Blair and B. W. Peyton. An introduction to chordal graphs and clique trees. In J. A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 1–30. Springer Verlag, 1993. IMA Volumes in Mathematics and its Applications, Vol. 56.
6. H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.
7. E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In R. H. Möhring, editor, *Graph Theoretical Concepts in Computer Science - WG '97*, pages 132–143. Springer Verlag, 1997. Lecture Notes in Computer Science 1335.
8. F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combin. Theory Ser. B*, 16:47–56, 1974.
9. C-W. Ho and R. C. T. Lee. Counting clique trees and computing perfect elimination schemes in parallel. *Inform. Process. Lett.*, 31:61–68, 1989.
10. T. Kloks, D. Kratsch, and J. Spinrad. On treewidth and minimum fill-in of asteroidal triple-free graphs. *Theoretical Computer Science*, 175:309–335, 1997.
11. C. G. Lekkerkerker and J. C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51:45–64, 1962.
12. T. Ohtsuki. A fast algorithm for finding an optimal ordering in the vertex elimination on a graph. *SIAM J. Comput.*, 5:133–145, 1976.
13. A. Parra. *Structural and algorithmic aspects of chordal graph embeddings.* PhD thesis, Technische Universität Berlin, 1996.
14. B.W. Peyton. Minimal orderings revisited. *SIAM J. Matrix Anal. Appl.*, 23(1):271–294, 2001.
15. D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.