Ann Oper Res (2008) 157: 47–60
DOI 10.1007/s10479-007-0190-4

Two methods for the generation of chordal graphs

**Lilian Markenzon · Oswaldo Vernet ·
Luiz Henrique Araujo**

Published online: 6 July 2007
© Springer Science+Business Media, LLC 2007

Abstract In this paper two methods for automatic generation of connected chordal graphs are proposed: the first one is based on new results concerning the dynamic maintenance of chordality under edge insertions; the second is based on expansion/merging of maximal cliques. Theoretical and experimental results are presented. In both methods, chordality is preserved along the whole generation process.

Keywords Chordal graphs · Dynamic algorithms · Clique-tree · Graph generation

In the solution of algorithmic problems, graphs can play different roles, being the very input for an algorithm or simply an auxiliary data structure handled by it. In the first case, the generation of suitable instances (i.e. input graphs satisfying given constraints) can be so complex that it constitutes a further problem, sometimes as hard to solve as the original one.

Chordal graphs are a broadly studied class, as their peculiar clique-based structure allows the solution of many algorithmic problems. See, for instance, Chandran et al. (2003), Gavril (1972), and Kumar and Madhavan (2002). Since the generation of instances for testing these algorithms is often required, our goal in this paper is to develop procedures for automatically constructing large connected chordal graphs, with more than 10,000 vertices.

Rather than generating connected graphs at random and testing whether they are chordal or not, we focus here on generation procedures in which graphs are constructed while chordality is maintained during the whole generation process. Two methods are presented.

The *incremental method* is based on new results about the dynamic maintenance of chordality under edge insertions developed by Araujo (2004). It allows the generation of

L. Markenzon's research is partially supported by grant 301068/2003-8, CNPq, Brazil.

L. Markenzon (✉) · O. Vernet
Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil
e-mail: markenzon@nce.ufrj.br

L.H. Araujo
Instituto Militar de Engenharia, Rio de Janeiro, Brazil

a chordal graph with given number of vertices and exact number of edges, without using any supporting data structures. The accuracy in the final number of edges is reached by monotonously increasing it by 1, which turns out to be a very costly process. As it will be shown in the experimental results, this method is more suitable for generating sparse graphs, whose number of edges is a small multiple of the number of vertices, like those which model most of real-life problems.

The *clique-based method* takes as input the number of vertices and an upper bound for the number of edges, which may be not reached precisely. The algorithm works much faster than the previous one and relies on the expansion/merging of maximal cliques in a clique-tree, which is used as a supporting data structure. The output is the final clique-tree, what can be very useful for some applications which need the chordal graph represented this way (see Blair and Peyton 1993). If necessary, a linear-time post-processing phase can easily convert the clique-tree into the traditional representation of the chordal graph through adjacency lists, as shown in Markenzon et al. (2004). A perfect elimination ordering (*peo*) for the generated graph is also issued by the algorithm, which may be equally helpful in some practical applications.

The complexities of both algorithms are analyzed and experimental tests, presented in Sect. 4, show that the first method is more suitable for generating sparse chordal graphs, whereas the second one can generate denser graphs very fast. Both are able to produce very large graphs and some of our test were performed on graphs with up to 10,000 vertices.

1 Basic notions

Let $G = (V, E)$ be a graph and $v \in V$. The set of neighbours of v is denoted as $\mathrm{Adj}(v) = \{w \in V | (v, w) \in E\}$. For any $S \subseteq V$, let $G[S]$ be the subgraph of G induced by S. S is a *clique* when $G[S]$ is a complete graph. If $\mathrm{Adj}(v)$ is a clique in G, v is said to be *simplicial* in G. A *perfect elimination ordering* (*peo*) is an ordering $\sigma = [v_1, \dots, v_n]$ of V with the property that v_i is a simplicial vertex in G$[\{v_i, \dots, v_n\}]$, $1 \le i \le n$. A proper subset $S \subset V$ is a *vertex separator* for non-adjacent vertices u and v (a *u–v separator*) if the removal of S from the graph separates u and v into distinct connected components. If no proper subset of S is an u–v separator then S is a *minimal u–v separator*. When the pair of vertices remains unspecified, we refer to S as a *minimal vertex separator*. A graph G is *chordal* when every cycle of length 4 or more has a chord (i.e. an edge joining two non-consecutive vertices of the cycle). Golumbic (1980) presents the following characterization:

Theorem 1 *Let G be a graph. The following statements are equivalent*:

- *G is a chordal graph.*
- *G has a perfect elimination ordering. Moreover, any simplicial vertex can start a perfect elimination ordering.*
- *Every minimal vertex separator induces a complete subgraph of G.*

In a connected graph $G = (V, E)$, a *spanning subgraph* $H = (V', E')$ of G is a subgraph of G such that $V' = V$. If H is a tree, it is called a *spanning tree* of G.

Given a connected chordal graph $G = (V, E)$ the clique-intersection graph of G is the connected weighted graph whose vertices are the maximal cliques of G and whose edges connect vertices corresponding to non-disjoint cliques. Each edge is assigned an integer weight, given by the cardinality of the intersection between the maximal cliques represented by its endpoints.

Every maximum-weight spanning tree (i.e. a spanning tree such that the sum of the weights of its edges is maximum) of the clique-intersection graph of G is called a *clique-tree* of G. It can be proved that each edge (Q', Q'') of a clique-tree corresponds to a minimal vertex separator for the vertices belonging to $Q' - Q''$ and $Q'' - Q'$. Besides, if Q_1 and Q_2 are maximal cliques of G, the intersection $Q_1 \cap Q_2$ is a subset of any maximal clique of G lying on the path between Q_1 and Q_2 in any clique-tree of G. See Blair and Peyton (1993) for more details.

2 Generation through successive edge insertions: an incremental method

The first algorithm takes as input V, with $|V| = n$, and the number of desired edges $m, n-1 \leq m \leq n(n-1)/2$, building up a connected labeled chordal graph $G = (V, E)$, with $|E| = m$, in two steps:

- Generates a labeled tree with vertices in V;
- Fills in this seed tree with the $m - n + 1$ missing edges. At each iteration, a pair of non-adjacent vertices $u, v \in V$ is selected and the edge (u, v) is added as long as the resulting graph is also chordal.

The first step aims at guaranteeing connectivity, since trees are chordal graphs. Using an appropriate tree codification (e.g. Prüfer coding) along with the corresponding reconstruction procedure (Deo and Micikevicius 2002; Tinhofer 1990), the first step can be accomplished in time $O(n)$. For the second step, the dynamic maintenance of chordality under edge insertions must be studied, avoiding to test for this property over the whole graph at every edge insertion.

In the last years there has been considerable research interest in dynamic algorithms. An algorithm for a problem is said to be *dynamic* if it is able to update a current solution while the structure of the problem undergoes changes, rather than computing an entirely new solution from scratch. Hence a standard model for dynamic graph problems involves a sequence of intermixed updates and queries: an update inserts or deletes an edge or isolated vertex and a query asks for certain information about a graph property (Alberts et al. 1997; Eppstein et al. 1997; Berry et al. 2003).

We are interested only in maintaining chordality under edge insertions, providing efficient algorithms for the following operations:

- *insert_query* (u, v): checks whether the insertion of edge (u, v) preserves chordality.
- *insert* (u, v): inserts the edge (u, v).

Ibarra (2000) solves this problem using a clique-tree of the graph as an auxiliary data structure. We propose instead a quite different approach, in that no additional data structure is needed. The central result is given in Theorem 2.

In a graph $G = (V, E)$, let $I_{u,v} = \text{Adj}(u) \cap \text{Adj}(v)$, for $u, v \in V$.

Theorem 2 *Let $G = (V, E)$ be a connected chordal graph and $u, v \in V$, $(u, v) \notin E$. The augmented graph $G + (u, v)$ is chordal if and only if $G[V - I_{u,v}]$ is not connected.*

Proof Let $S = V - I_{u,v}$ and $G' = G + (u, v)$.

$\Rightarrow$ If $G[S]$ is connected, there must be at least one path between u and v in $G[S]$. Let $P_{u,v}$ be such a path with minimum length. As no vertex belonging to $I_{u,v}$ lies on $P_{u,v}$, this path has length greater than 2. Hence $G'[S] = G[S] + (u, v)$ has a cycle with length greater than 3, composed of $P_{u,v}$ and the new edge (u, v). Since $P_{u,v}$ has minimum length, this cycle has no chords and $G'[S]$ is not chordal. So G' is not chordal either.

$\Leftarrow$ If $G[S]$ is not connected, then $I_{u,v}$ is a u–v separator in G, since G is connected by assumption. Moreover, $I_{u,v}$ is a minimal u–v separator. Thus, by Theorem 1, $I_{u,v}$ is a clique in G and $I_{u,v} \cup \{u, v\}$ is a clique in G'. The vertices u and v belong to distinct connected components of $G[S]$: $G_u = (V_u, E_u)$ and $G_v = (V_v, E_v)$, both chordal. For every $x \in V_u$ and $y \in V_v$, $I_{u,v}$ is a minimal x–y separator in G, but not in G'. However, the only minimal x–y separators in G that may have been modified under the addition of (u, v) are subsets of $I_{u,v} \cup \{u, v\}$, which are cliques in G'. So, every minimal vertex separator of G' is a clique and, by Theorem 1, G' is chordal. □

Corollary 1 *Let $G = (V, E)$ be a connected chordal graph and $u, v \in V$, $(u, v) \notin E$. If $I_{u,v} = \varnothing$, then $G + (u, v)$ is not chordal.*

Proof If $I_{u,v} = \varnothing$, then $G[V - I_{u,v}] = G[V] = G$. Since G is connected, by Theorem 2, $G + (u, v)$ is not chordal. □

Theorem 2 and Corollary 1 give the answer to *insert_query* (u, v): if $I_{u,v} \neq \varnothing$, a path must be searched between u and v in $G[V - I_{u,v}]$, i.e., in the worst case, all vertices and edges of $G[V - I_{u,v}]$ must be traversed. If $I_{u,v}$ has few vertices, this search may cover almost every vertex and edge of G. Lemma 1 shows however that this search can be constrained.

Lemma 1 *Let $G = (V, E)$ be a connected chordal graph. If there is a non-empty path $P_{u,v}$ between u and v with minimum length in $G[V - I_{u,v}]$, then $\{w\} \cup I_{u,v}$ is a clique in G, for all $w \in P_{u,v}$.*

Proof If $I_{u,v} = \varnothing$, the result holds trivially. Otherwise, let $P_{u,v} = [u = a_1, a_2, \dots, a_k = v]$, $k > 3$, and $t \in I_{u,v}$. By assumption, $a_i \notin I_{u,v}$, $1 \le i \le k$. So, there is a cycle in $G[t, u = a_1, a_2, \dots, a_k = v, t]$. Since $P_{u,v}$ has minimum length and G is chordal, all chords within this cycle must have t as an endpoint. Hence every vertex on $P_{u,v}$ is adjacent to every vertex in $I_{u,v}$. But $I_{u,v}$ is a subset of a minimal u–v separator and, by Theorem 1, $I_{u,v}$ is a clique. □

Lemma 1 reduces the set of candidates to the path $P_{u,v}$. Actually the search for such a path can be restricted to $G[(V - I_{u,v}) \cap \mathrm{Adj}(x)] = G[\mathrm{Adj}(x) - I_{u,v}]$, for any $x \in I_{u,v}$. The implementation of *insert_query* (u, v) is shown in the following procedure. The worst-case time complexities of the operations *insert_query* (u, v) and *insert* (u, v) are given in Lemmas 2 and 3.

```
procedure insert_query ($G = (V, E), u, v$);
begin
    $I_{u,v} \leftarrow \mathrm{Adj}(u) \cap \mathrm{Adj}(v)$;
    if $I_{u,v} = \varnothing$ then
        $G + (u, v)$ is not chordal
    else begin
        Choose $x \in I_{u,v}$;
        Aux $\leftarrow G[\mathrm{Adj}(x) - I_{u,v}]$;
```

```
            Perform a breadth-first search on Aux, starting at $u$;
            $G + (u, v)$ is chordal $\Leftrightarrow$ $v$ has not been reached during the search
        **end**
    **end**;
```

Lemma 2 *In the worst case, insert_query* (u, v) *performs in time* $O(m)$.

Proof $I_{u,v}$ can be obtained in $O(n)$. The induced subgraph Aux does not need to be explicitly computed: the vertices belonging to $\mathrm{Adj}(x) - I_{u,v}$ can be marked as the only ones to be visited during the search. This can be also done in $O(n)$. In the worst case, the breadth-first search performs in time $O(m)$. □

Lemma 3 *Operation insert* (u, v) *has complexity of* $O(1)$.

Proof Since no additional data structure is maintained, just insert u in $\mathrm{Adj}(v)$ and v in $\mathrm{Adj}(u)$. □

Evidently the classical recognition algorithm of Rose et al. (1976) could be used to solve this problem: for each new candidate edge (u, v), chordality must be tested for the augmented graph $G + (u, v)$. This algorithm consists of two steps: a lexicographic breadth-first search, which produces a vertex ordering σ, and the verification if σ is a *peo*. For the answer *yes*, $2m$ iterations are performed in each step. For the answer *no*, the first step is entirely performed, whereas the second one may not run to completion. Instead, our algorithm performs a single step, in which at most $2m$ edges are traversed. In average, much fewer edges are considered in the search, given the result of Lemma 1. In particular, the answer *no* can be obtained much faster, since the test $\mathrm{Adj}(u) \cap \mathrm{Adj}(v) \neq \emptyset$, which can be accomplished in $O(n)$ time, may readily fail.

Although the worst-case complexities obtained in Lemmas 2 and 3 does not represent a theoretical improvement over the classical recognition algorithm, it is not difficult to conclude that our algorithm performs much faster on average.

We stress here an interesting result concerning the generation of connected chordal graphs through this method. When trying to add a new edge joining a pair of randomly selected vertices, two kinds of failures may arise:

- *Type-1 failures*: the pair of vertices corresponds to an existent edge (or loop) and must be discarded.
- *Type-2 failures*: the edge is new, but its insertion violates chordality, hence it is also discarded.

As the graph grows denser, the probability of selecting already existing edges increases, so that the number of *Type-1 failures* tends to dominate the generation process, whereas the number of *Type-2 failures* decreases. Although more frequent, *Type-1 failures* consume much less time than *Type-2 failures* do in the overall process.

To avoid the selection of useless edges, an auxiliary data structure can be introduced: a sequential list of initial size $n(n-1)/2 - (n-1)$, storing candidate edges—exactly those belonging to the complement of the graph being generated. At each step, one edge of this list is chosen as a candidate, rather than randomly selecting a pair of vertices. If the insertion of the selected edge in the graph being generated preserves chordality, it is removed from the list of candidates; otherwise, the list remains unaffected.

Both versions of the algorithm (with and without the auxiliary data structure) perform practically in the same amount of time. This is due to the time spent in the initialization and maintenance of the auxiliary list, which is $O(n^2)$. So the time saved by avoiding *Type-1 failures* is wasted in handling the auxiliary structure. Moreover an extra memory storage is required, which is also $O(n^2)$. We conclude that the pure version is more advantageous.

3 Generation by expansion and merging of maximal cliques

The second generation method relies strongly on the clique-structure of a chordal graph. The user must provide as input the number of vertices and an upper bound for the number of edges. The output issued by the algorithm is a clique-tree of the graph. This feature is very interesting due to applications of chordal graphs and clique trees in, for instance, sparse matrix computations (see Blair and Peyton 1993 for details). As a further step, we show how the clique-tree can be converted to adjacency lists in linear time. The algorithm provides also a perfect elimination ordering of the vertices.

The maintenance of the clique-tree as a supporting data structure allows fast identification and selection of cliques. The efficiency of the algorithm qualifies it for the generation of very large graphs (up to 100,000 vertices).

3.1 The algorithm

A careful look at the *perfect elimination ordering* (*peo*) suggests a secure way of adding a new vertex to an existent chordal graph: if the incoming vertex is joined to any clique of the graph, the resulting graph is guaranteed to be chordal.

Lemma 4 *Let $G = (V, E)$ be a chordal graph, $v \notin V$ a new vertex and $Q \subseteq V$ a clique of G. The graph JOIN $(G, Q, v) = (V \cup \{v\}, E \cup \{(v, x) | x \in Q\})$ is also chordal.*

Proof Just notice that v is a simplicial vertex of JOIN (G, Q, v). So, if $[v_1, v_2, \ldots, v_n]$ is a *peo* of G, then $[v, v_1, v_2, \ldots, v_n]$ is a *peo* of JOIN (G, Q, v) and, by Theorem 1, JOIN (G, Q, v) is chordal. □

The algorithm for generating a connected chordal graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, has three steps:

- *Initialization step*. Start with the trivial graph $G = (\{v_1\}, \varnothing)$, $v_1 \in V$.
- *Clique-expansion step*. For $v \leftarrow v_2, \ldots, v_n$, perform: choose a clique $Q \subseteq V$ of G and replace G with the graph JOIN (G, Q, v), linking v to all vertices of Q.
- *Clique-merging step*. Perform many times: choose two maximal cliques Q' and Q'' of G, represented by adjacent nodes in the clique-tree, and merge them into a single one.

The first step sets up a trivial graph. During the second step, vertices are added in such a way that, after $n - 1$ iterations, the reverse of the incoming order is a *peo*. Experimental results show that, at the end of the second step, the resulting graph is likely to have only a few edges. Thus the third step increases the number of edges, allowing to reach the upper bound specified as input by the user, what can originate a sparse or a dense graph. It is important to observe that the third step does not affect the *peo* obtained beforehand. Following, the last two steps are described in details.

Table 1 Experimental results

n	$\overline{m}$	$\overline{m}/n$	sd	min	max
10000	21135.40	**2.114**	289.86	20093	22640
20000	42312.48	**2.116**	419.12	40746	44401
30000	63492.96	**2.116**	525.14	61494	65878
40000	84678.21	**2.117**	613.27	82460	87900
50000	105871.98	**2.117**	690.48	103434	109132
60000	127051.38	**2.118**	756.71	124217	131327
70000	148242.81	**2.118**	824.94	144836	152174
80000	169431.58	**2.118**	891.14	166192	174837
90000	190610.54	**2.118**	941.51	187188	196546
100000	211808.79	**2.118**	1005.11	207799	217068

In order to compute JOIN (G, Q, v) efficiently, a fast way to choose a clique Q of G at random is required. Since any clique Q is a subset of a maximal clique, the idea is to keep track of all maximal cliques of G, using a *clique-tree*. Recall from the definition presented in Sect. 2 that each edge of a clique-tree has as weight the cardinality of the intersection between the cliques given by its endpoints. This auxiliary structure is maintained along the generation process according to the following guidelines:

- Initially the clique-tree is trivial, consisting of a single node: $\{v_1\}$.
- At each iteration of the *Clique-expansion step*, a node Q' of the clique-tree is selected. This node is a maximal clique in G and a subset $Q \subseteq Q'$ is chosen.
  - If $Q = Q'$, a new maximal clique $\{v\} \cup Q$ replaces Q' in the graph being generated. So the node Q' is simply replaced with $\{v\} \cup Q$ in the clique-tree.
  - If $Q \subset Q'$, a new maximal clique $\{v\} \cup Q$ is created and Q is a minimal separator between v and the vertices in $Q' - Q$. Thus a new node $\{v\} \cup Q$ is added to the clique-tree, joined to Q' by an edge with weight $|Q|$.

In Table 1, we summarize some experimental results about the generation of large graphs without the *Clique-merging step*. Each row shows the results of 2^{15} executions of the algorithm for the number of vertices specified in the first column (n); $\overline{m}$ is the average number of edges, sd is the standard deviation and min, max are respectively the minimum and the maximum number of edges obtained.

The results suggest that the average number of edges of the generated graphs tends to grow up almost linearly with the number of vertices, within a factor of 2.11. Hence, this two-step algorithm produces mostly sparse graphs with small maximal cliques, due to the following facts:

- The initial clique has size 1.
- Each time a clique $Q \subseteq Q'$ is chosen, exactly $|Q|$ edges are added to the graph being generated.
- A chosen maximal clique Q' has only $1/|Q'|$ probability to grow (when $Q = Q'$).
- Although small cliques have higher probability of growing, the total number of edges undergoes a significant increase only when Q' has maximum size.

In order to allow the generation of denser graphs, the *Clique-merging step* repeatedly merges two maximal cliques into a single one. When two cliques Q' and Q'', adjacent in

Table 2 More experimental results

$n = 10,000$, $U = n(n-1)/2 = 49,995,000$						
M	M/U	$\overline{m}$	$\overline{m}/U$	sd	maxclique	merges
10000	(1.0 n)	21135.40	(2.1 n)	289.86	9.08	0.00
15000	(1.5 n)	21135.40	(2.1 n)	289.86	9.08	0.00
20000	(2.0 n)	21135.40	(2.1 n)	289.86	9.08	0.00
25000	(2.5 n)	25000.00	(2.5 n)	0.00	22.98	816.70
214980	(<1%)	214980.00	(<1%)	0.00	409.80	3745.42
404960	(<1%)	404960.00	(<1%)	0.00	634.57	4153.20
499950	(1.00%)	499950.00	(1.00%)	0.00	727.90	4277.76
4999500	(10.00%)	4999500.00	(10.00%)	0.04	2847.09	5485.50
9999000	(20.00%)	9998999.75	(20.00%)	8.76	4178.11	5820.81
14998500	(30.00%)	14998021.40	(30.00%)	22553.66	5213.73	6013.69
19998000	(40.00%)	19988378.12	(39.98%)	131036.68	6100.69	6145.99
24997500	(50.00%)	24965199.52	(49.94%)	317725.92	6875.00	6248.87
29997000	(60.00%)	29720361.41	(59.45%)	983019.15	7559.18	6328.46
34996500	(70.00%)	34344306.45	(68.70%)	2066276.17	8170.49	6389.55
39996000	(80.00%)	39073234.46	(78.15%)	2875688.94	8760.86	6438.77
44995500	(90.00%)	43950092.65	(87.91%)	3279525.34	9330.25	6483.56
49995000	(100.00%)	49995000.00	(100.00%)	0.00	10000.00	6534.63

the clique-tree, are collapsed, exactly $(|Q'| - \text{weight}(Q', Q'')) \times (|Q''| - \text{weight}(Q', Q''))$ edges with endpoints in $Q' - Q''$ and $Q'' - Q'$ are added to G, since weight(Q', Q'') is the number of vertices that Q' and Q'' have in common. In the clique-tree, the edge (Q', Q'') disappears and the weights of the remaining ones are unaffected.

As the generated graph is connected, no more than $n-1$ maximal cliques can be obtained by the *Clique-expansion step*. Hence, at most $n-2$ merge operations can be performed during the *Clique-merging step*. Since an upper bound for the number of edges is given as input, the *Clique-merging step* must be interrupted as soon as this limit is reached.

The new results are shown in Table 2. For $n = 10{,}000$, the generated graphs can have at most $U = n(n-1)/2 = 49{,}995{,}000$ edges. Several upper bounds for the number of edges (M) are tested, varying in the range $n, \ldots, U$. For each of these values, 2^{15} graphs are generated and the following informations are shown: the average number of edges (column 3), the standard deviation (column 5), the average size of the maximum clique (column 6) and the average number of clique merges (column 7).

The results in Table 2 can be understood as follows:

- In the first section, very sparse graphs ($n \le M \le 2n$) are generated and $\overline{m} > M$; the upper limit is violated during the *Clique-expansion step* and no merge operation is performed.
- In the second section, sparse graphs with $M \ge 2.5n$ edges are produced and $\overline{m} = M$ (the standard deviation is 0.0).
- In the third section, as the density of the graphs become higher, a slight difference can be noticed between the values of $\overline{m}$ and M.

3.2 Implementation and complexity

The inputs of the algorithm are: the number of vertices n and the upper bound for the number of edges M. Without loss of generality, let us assume that the generated graph will have $V = \{1, \ldots, n\}$ as its vertex set. The following data structures are used:

- Each node of the clique-tree is represented by a linked list of vertices and the array of those lists is called $\mathcal{Q}$, with positions in the range $1, \ldots, n-1$.
- $\mathcal{S}$ is an array of integers, such that $\mathcal{S}_i$ is the cardinality of $\mathcal{Q}_i$, $1 \le i < n$.
- $\mathcal{L}$ is the list of weighted edges of the clique-tree; each element of $\mathcal{L}$ is a triple containing the endpoints of an edge (actually the positions of the endpoints in the array $\mathcal{Q}$) and its weight.

The implementation of the initial and the *Clique-expansion steps* is given in the following procedure. The operator ‖ means list concatenation.

At each iteration, the current vertex v must be joined to an existent clique. The value of i is the position in the array $\mathcal{Q}$ of the maximal clique from which this clique is chosen and t stores the cardinality of the clique. Hence the value of t is exactly the number of edges being added to the chordal graph being generated. After the execution of *expand_cliques*, the following can be claimed:

- The variable ℓ holds the number of maximal cliques generated ($\ell < n$, since the graph is connected).
- The array $\mathcal{Q}$ stores the contents of the ℓ maximal cliques generated.
- The list of triples $\mathcal{L}$ contains the edges of the clique-tree along with their weights; each edge is given by a pair of integers (i, j), which are the positions of its endpoints in the array $\mathcal{Q}$.

```
procedure expand_cliques;
begin
   $\mathcal{Q}_1 \leftarrow [1]; \mathcal{S}_1 \leftarrow 1; \mathcal{L} \leftarrow []; m \leftarrow 0; \ell \leftarrow 1;$
   for $v \leftarrow 2, 3, \ldots, n$ do
   begin
      Choose $i, 1 \le i \le \ell$;
      Choose $t, 1 \le t \le \mathcal{S}_i$;
      if $t = \mathcal{S}_i$ then
      begin
         $\mathcal{Q}_i \leftarrow [v] \parallel \mathcal{Q}_i$;   {an old clique is expanded}
         $\mathcal{S}_i \leftarrow \mathcal{S}_i + 1$
      end
      else begin
         Choose $Q \subseteq \mathcal{Q}_i$ with $|Q| = t$;
         $\ell \leftarrow \ell + 1$;
         $\mathcal{Q}_\ell \leftarrow [v] \parallel Q$;   {a new clique is created}
         $\mathcal{S}_\ell \leftarrow t + 1$;
         $\mathcal{L} \leftarrow \mathcal{L} \parallel [(i, \ell, t)]$
      end;
      $m \leftarrow m + t$
   end
end;
```

Let m_1 denote the number of edges of the chordal graph whose clique-tree is generated by *expand_cliques* (i.e. the value of m after the execution of *expand_cliques*). Lemma 5 shows an upper bound for the number of elements (vertices) belonging to the lists $\mathcal{Q}_i$, $1 \le i \le \ell$:

Lemma 5 *After the execution of expand_cliques*, $|\mathcal{Q}_1| + \cdots + |\mathcal{Q}_\ell| \le n + m_1$.

Proof Let us call Δ_v the number of vertices added to some list at the v-th iteration, $1 \le v \le n$.

- Initially, vertex 1 is added to $\mathcal{Q}_1 : \Delta_1 = 1$.
- At the subsequent iterations ($v \leftarrow 2, 3, \dots, n$), either v is added to $\mathcal{Q}_i$, $1 \le i < v$, or a new maximal clique is created and $\mathcal{Q}_\ell$ is built up with $t_v + 1$ vertices, where t_v means the value of t at iteration v; in both cases, $\Delta_v \le t_v + 1, 2 \le v \le n$.

Hence $|\mathcal{Q}_1| + \cdots + |\mathcal{Q}_\ell| = \Delta_1 + \cdots + \Delta_n \le 1 + (t_2 + 1) + \cdots + (t_n + 1) = n + t_2 + \cdots + t_n = n + m_1$. □

Lemma 6 *At each iteration, if the choice "$Q \subseteq \mathcal{Q}_i$ with $|Q| = t$" is performed in $O(t)$, then expand_cliques performs in the worst case in time $O(n + m_1)$.*

Proof To select a subclique $Q \subseteq \mathcal{Q}_i$ with t elements in time $O(t)$, just pick up the first t elements of $\mathcal{Q}_i$. Since the two other choices at the iteration can be performed in $O(1)$, the result follows directly from Lemma 5. □

At each iteration of the *Clique-merging step*, a pair of adjacent nodes of the clique-tree is selected to be coalesced. If explicitly performed, this operation involves modifications on the clique-tree: a new node replaces the original ones in the array $\mathcal{Q}$ and the list $\mathcal{L}$ must be consistently altered. To avoid this overhead, a disjoint set union structure is used, along with the well known operations UNION and FIND (Tarjan 1983). The initial collection consists of unitary sets containing the positions of the nodes in the array $\mathcal{Q}$. Since each node of the clique-tree is represented by a list of vertices of the original graph, when coalescing two nodes, it is enough to concatenate both lists, yielding a new one with repeated elements. So, at the end of the generation process, the remaining lists need to be packed. In order to assure that the final number of edges does not exceed the given upper bound M, we must determine the exact number of edges added at each iteration, as shown beforehand. The *Clique-merging step* is implemented by the following algorithm (notice that $\mathcal{S}_i$ always contains the true cardinality of $\mathcal{Q}_i$, considering repeated elements only once):

```
procedure merge_cliques;
begin
   Initial collection of disjoint sets ← $\{\{1\}, \{2\}, \dots, \{\ell\}\}$;
   while $\mathcal{L} \neq []$ and $m < M$ do
   begin
      Choose $(a, b, \omega) \in \mathcal{L}$ and remove $(a, b, \omega)$ from $\mathcal{L}$;
      $i \leftarrow$ FIND $(a)$;      $\delta \leftarrow \mathcal{S}_i - \omega$;
      $j \leftarrow$ FIND $(b)$;      $\Delta \leftarrow \mathcal{S}_j - \omega$;
      if $m + \Delta \times \delta \le M$ then
      begin
         UNION $(i, j, i)$;
         $\mathcal{Q}_i \leftarrow \mathcal{Q}_i || \mathcal{Q}_j$;      $\mathcal{S}_i \leftarrow \Delta + \delta + \omega$;
```

```
            $\mathcal{Q}_j \leftarrow []$;      $\mathcal{S}_j \leftarrow 0$;
            $m \leftarrow m + \Delta \times \delta$
        end
    end
end;
```

Lemma 7 gives the worst-case time complexity of *merge_cliques*. Notice that this complexity depends only on n.

Lemma 7 *In the worst case, merge_cliques performs in time* $O(n\alpha(2n, n))$.

Proof Considering $\mathcal{L}$ as a sequential list (with at most n positions), the choice and deletion from $\mathcal{L}$ can both be performed in $O(1)$. Since $\ell < n$, in the worst case $\ell = n - 1$ and $|\mathcal{L}| = n - 2$. So the initialization of the collection can be performed in $O(n)$ and, at each iteration, two FINDs and at most one UNION are performed. Since there is at most $n - 2$ iterations, less than $2n$ FINDs and $n - 1$ UNIONs are executed. The overall complexity is then $O(n\alpha(2n, n))$, where $\alpha(p, q)$ is a functional related to the inverse of Ackermann's function (Tarjan 1983). □

It is important to observe that the total number of elements, given by $|\mathcal{Q}_1| + \cdots + |\mathcal{Q}_\ell|$, remains unaffected after the execution of *merge_cliques*: when a list $\mathcal{Q}_j$ is concatenated into $\mathcal{Q}_i$ no element is lost and $\mathcal{Q}_j$ becomes empty.

After the execution of procedure *merge_cliques*, the non-empty lists in the array $\mathcal{Q}$ have repeated elements and a packing step is required. All lists are traversed once, so the complexity is $O(|\mathcal{Q}_1| + \cdots + |\mathcal{Q}_\ell|)$. Since the total number of elements, determined in Lemma 5, is not affected by procedure *merge_cliques*, this step can be trivially performed in $O(n + m_1)$.

Computing the overall complexity of the generation so far, we obtain $O(n + m_1 + n\alpha(2n, n))$. However, the desired chordal graph is represented through its clique-tree, instead of the usual adjacency lists. In the next subsection, a linear-time conversion phase is examined.

3.3 Constructing the graph from the clique-tree

Phase 2 is a conversion phase: the desired connected chordal graph must be obtained from the so far generated clique-tree. Since the maximal cliques may have edges in common, the main concern here is to avoid the generation of the same edge more than once. Blair and Peyton (1993) provide us with the basic concepts needed to perform this task efficiently.

A total ordering of the cliques of the *clique-intersection graph*, say $Q_1, Q_2, \ldots, Q_\ell$, has the *running intersection property* (RIP) if, for each clique $Q_j, 2 \le j \le \ell$, there exists a clique $Q_i, 2 \le i \le j - 1$, such that $Q_j \cap (Q_1 \cup Q_2 \cup \ldots Q_{j-1}) \subset Q_i$.

For any RIP ordering of the cliques, a tree T_{rip} on the set of maximal cliques of G can be constructed by making each clique Q_i adjacent to a *parent* clique Q_j identified by the expression above. Observe that any RIP ordering numbers each parent before any of its children. Moreover, also by Blair and Peyton (1993), the set of trees T_{rip} equals the set of clique-trees of G.

In order to build the adjacency lists of the vertices of the graph, we must inspect all the cliques of the clique-tree. Initially, all adjacency lists are empty. When inspecting the vertices of the i-th clique $\mathcal{Q}_i$, they are partitioned into two sublists: $\mathcal{O}$ stores the vertices that have already appeared in a clique at a previous iteration; $\mathcal{N}$ holds the new vertices, which have never appeared before in a clique.

We have to produce a clique with all the vertices belonging to $\mathcal{N}$, issuing all needed edges among them. However, depending on the order in which the cliques are inspected, the vertices belonging to $\mathcal{O}$ need only be linked to the vertices in $\mathcal{N}$; we assume that their edges exist already. In other words, if $v, w \in \mathcal{O}$ then they have appeared before in the same clique. The running intersection property makes us sure of that.

It can be observed that, by construction, the sequence of cliques $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_\ell$ obtained by the generation algorithm establishes a RIP ordering. Hence the cliques must be processed in this order.

Thus the conversion phase issues only once each edge belonging to the graph. So, the construction of the adjacency lists of the graph from the clique-tree performs in time $O(n + |\mathcal{Q}_1| + \cdots + |\mathcal{Q}_\ell| + m) = O(m)$, where m is the number of edges of the generated chordal graph. Including the conversion phase, the overall complexity of the second generation method is: $O(n + m_1 + n\alpha(2n, n) + m) = O(m + n\alpha(2n, n))$.

4 A Comparison between both methods

In Table 3, the execution times of the algorithms presented in the previous sections are compared through the generation of connected chordal graphs with $n = 1000$ vertices and increasing edge density. 1000 graphs are generated for each value of the edge density and the average times of both algorithms are measured. In order to allow a fair comparison, the execution times of the clique-based method include the post-processing phase, in which the clique-tree is converted to the usual adjacency list representation. The last column shows,

Table 3 A time-comparison between both methods

m		Incremental Method			Clique-based Method			
		Execution Time (s)			Execution Time (s)			$\overline{m}$
		average	min	max	average	min	max	
n	1000	**0.000176**	0.000000	0.010000	**0.000283**	≈0	0.010000	2096.67
$2n$	2000	**0.071880**	0.059999	0.080000	**0.000290**	≈0	0.010000	2099.71
$3n$	3000	**0.165250**	0.109999	0.180000	**0.000370**	≈0	0.010000	3000.00
$4n$	4000	**0.280150**	0.259999	0.310000	**0.000390**	≈0	0.010000	4000.00
1%	4995	**0.416140**	0.379999	0.450000	**0.000400**	≈0	0.010000	4995.00
2%	9990	**1.529200**	1.479999	1.600000	**0.000440**	≈0	0.010000	9989.99
3%	14985	**3.767000**	3.530000	4.070000	**0.000480**	≈0	0.010000	14984.99
4%	19980	**8.451000**	7.660000	9.239999	**0.000520**	≈0	0.010000	19979.99
5%	24975	**17.447000**	15.880000	18.810000	**0.000540**	≈0	0.010000	24974.98
6%	29970	**33.978000**	28.270000	40.630000	**0.000580**	≈0	0.010000	29969.97
7%	34965	**55.755000**	47.570000	69.950000	**0.000610**	≈0	0.010000	34964.92
8%	39960	**93.826000**	83.179999	107.390000	**0.000640**	≈0	0.010000	39959.91
9%	44955	**149.811000**	127.590000	200.900000	**0.000670**	≈0	0.010000	44954.87
10%	49950	**235.744000**	197.820000	303.690000	**0.000700**	≈0	0.010000	49949.78
20%	99900	–	–	–	**0.000970**	≈0	0.010000	99897.31
30%	149850	–	–	–	**0.001290**	≈0	0.010000	149819.81
40%	199800	–	–	–	**0.001550**	≈0	0.010000	199407.14

for this method, the average number of edges obtained. Tests were performed on a 2.8 Ghz Pentium IV processor, running the Linux Operating System.

It can be observed that the clique-based method systematically outdoes the incremental method in execution time, being able to generate chordal graphs with different edge densities very fast. As the density increases, the execution time of the incremental method grows drastically, and the tests were aborted as soon as 10% was reached. However, when generating very sparse graphs ($m \leq 2n$), the clique-based method produces, in average, more edges than required; in this case, the incremental method must be preferred, since it performs reasonably fast and a higher accuracy in the final number of edges is often a major concern.

5 Conclusions

We have presented two methods for the generation of connected chordal graphs. The incremental method successively new edges to an existent chordal graph, starting at a tree to ensure connectivity. Experimental results show that this method is suitable for generating sparse graphs and, although a great amount of time is spent in the generation, it has the following advantages: the final number of edges can be precisely established a priori and no auxiliary data structures are necessary.

The clique-based method employs a clique-tree as a supporting data structure to represent the chordal graph being generated. Maximal cliques are expanded and contracted, yielding a final clique-tree that may be converted to the adjacency list graph format if necessary. Although the final number of edges cannot be set in advance, an upper limit is given as input and the experimental results show that the average number of edges obtained is close enough to the desired bound. Whenever accuracy in the number of edges is not the main concern, the clique-based method must be preferred, since it systematically outdoes the incremental one in execution time. Moreover, it also produces a perfect elimination ordering (*peo*) of the vertices along with the generated graph, what can be useful in many practical applications.

References

Alberts, D., Cattaneo, G., & Italiano, G. F. (1997). An empirical study of dynamic graph algorithms. *ACM Journal of Experimental Algorithms*, *6*, 1–39.

Araujo, L. H. (2004). *Algoritmos dinâmicos para manutenção de grafos cordais e periplanares*. Ph.D. Thesis, Coppe-Produção, Universidade Federal do Rio de Janeiro, RJ, Brasil.

Blair, J. R. S., & Peyton, B. (1993). An introduction to chordal graphs and clique trees. In J. A. George, J. R. Gilbert & J. W. Liu (Eds.), *IMA volumes in mathematics and its applications: Vol. 56. Graph theory and sparse matrix computation* (pp. 1–29). Berlin: Springer.

Berry, A., Heggernes, P., & Villanger, Y. (2003). A vertex incremental approach for dynamically maintaining chordal graphs. In *Proceedings of the 14th international symposium on algorithms and computation (ISAAC 2003)*, Springer LNCS 2906, pp. 47–57.

Chandran, L. S., Ibarra, L., Ruskey, F., & Sawada, J. (2003). Generating and characterizing the perfect elimination orderings of a chordal graph. *Theoretical Computer Science*, *307*, 303–317.

Deo, N., & Micikevicius, P. (2002). A new encoding for labeled trees employing a stack and a queue. *Bulletin of the Institute of Combinatorics and Its Applications*, *34*, 77–85.

Eppstein, D., Galil, Z., & Italiano, G. F. (1997). Sparsification—a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, *44*, 669–696.

Gavril, F. (1972). Algorithms for minimum coloring, minimum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, *1*, 180–187.

Golumbic, M. C. (1980). *Algorithmic graph theory and perfect graphs*. New York: Academic.

Ibarra, L. (2000). *Fully dynamic algorithms for chordal graphs and split graphs*. Technical Report DCS-262-IR, University of Victoria, http://facweb.cs.depaul.edu/ibarra/research.htm.

Kumar, P. S., & Madhavan, C. E. V. (2002). Clique tree generalization and new subclasses of chordal graphs. *Discrete Applied Mathematics*, *117*, 109–131.

Markenzon, L., Vernet, O., & Araujo, L. H. (2004). *Two methods for the generation of chordal graphs*. Technical Report NCE-13/04, Universidade Federal do Rio de Janeiro.

Rose, D. J., Tarjan, R. E., & Lueker, G. (1976). Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, *5*, 266–283.

Tarjan, R. E. (1983). *Data structures and network algorithms*. Philadelphia: SIAM.

Tinhofer, G. (1990). Generating graphs uniformly at random. *Computing Supplement*, *7*, 235–255.