Two Methods for the Generation of Chordal Graphs

L. Markenzon[1], O. Vernet[1], and L. H. Araújo[2]

[1] Núcleo de Computação Eletrônica
Universidade federal do Rio de Janeiro
CP 2324, 20010-974, Rio de Janeiro, RJ, Brasil
{markenzon,vernet}@nce.ufrj.br
[2] Instituto Militar de Engenharia
Praça General Tibúrcio, 80
22290-270, Rio de Janeiro, RJ, Brasil
lharaujo@ime.eb.br

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

1 Introduction

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 a more efficient solution for many algorithmic problems [5, 9, 8, 12]. Since the generation of instances for testing these algorithms is often required, our goal in this paper is to develop procedures for automatically constructing connected chordal graphs.

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 in this paper: the first one is based on a dynamic maintainance of chordality under edge insertions developed in [2]; the second one relies on the expansion/merging of maximal cliques in the clique-tree. The complexities of both algorithms are analyzed and experimental tests show that the first method is more suitable for generating sparse chordal graphs, whereas the second one can generate denser graphs very fast.

2 Basic Notions

Let $G = (V, E)$ be a graph and $v \in V$. The set of neighbours of v is denoted as $Adj(v) = \{w \in V \mid (v, w) \in E\}$. 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 $Adj(v)$ is a clique in G, v is said to be *simplicial* in G. A *perfect elimination ordering* (*peo*) is an ordering $\sigma = [v_1, \ldots, v_n]$ of V with the property that v_i is a simplicial vertex in $G[\{v_i, v_{i+1}, \ldots, v_n\}]$, $1 \leq i \leq n$. 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.

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 [10] 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 ordering.*
- *Every minimal vertex separator induces a complete subgraph 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 of the clique-intersecton 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 [3] for more details.

3 Generation of Chordal Graphs Through Successive Edge Insertions

The first algorithm takes as input V, with $|V| = n$, and the number of desired edges m, $n - 1 \leq m \leq \frac{n(n-1)}{2}$, 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 signature (e.g. Prüfer coding) along with the corresponding reconstruction procedure ([6],[15]), the first step can be accomplished in time $O(n)$. For the second step, the dynamic maintainance 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 [1, 7, 4].

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 [11] 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 $I_{u,v} = Adj(u) \cap Adj(v)$, 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 minimun 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 $I_{u,v} = \emptyset$, then $G + (u, v)$ is not chordal.*

Proof. If $I_{u,v} = \emptyset$, 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 $I_{u,v} \neq \emptyset$, a path must be searched between u and v in $G[V - I_{u,v}]$, i.e. all vertices and edges of $G[V - I_{u,v}]$ must be traversed, in the worst case. 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 minimun-length in $G[V - I_{u,v}]$, then $\{w\} \cup I_{u,v}$ is a clique in G, $\forall w \in P_{u,v}$.*

Proof. If $I_{u,v} = \emptyset$, the result holds trivially. Otherwise, let $P_{u,v} = [u = a_1, a_2, \ldots, a_k = v]$, $k \geq 2$, and $t \in I_{u,v}$. By assumption, $a_i \notin I_{u,v}$, $1 \leq i \leq k$. So, there is a cycle in G $[t, u = a_1, a_2, \ldots, a_k = v, t]$. 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 $G[(V - I_{u,v}) \cap Adj(x)]$, for any $x \in I_{u,v}$. The implementation of *insert_query*(u, v) is shown in the following procedure.

```
procedure insert_query ($G = (V, E), u, v$);
begin
   $I_{u,v} \leftarrow Adj(u) \cap Adj(v)$;
   if $I_{u,v} = \emptyset$ then
      $G + (u, v)$ is not chordal
   else begin
      Choose $x \in I_{u,v}$;
      $Aux \leftarrow G[(V - I_{u,v}) \cap Adj(x)]$;
      Perform a breadth-first search on $Aux$, starting at $u$;
      $G + (u, v)$ is chordal $\Leftrightarrow$ $v$ has not been reached in the search
   end
end;
```

The worst-case time complexities of the operations *insert_query*(u, v) and *insert*(u, v) are given in Lemmas 2 and 3.

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

Proof. $I = Adj(u) \cap Adj(v)$ can be obtained in $O(n)$. The induced subgraph Aux does not need to be explicitly computed: the vertices belonging to $(V - I) \cap Adj(x)$ 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. As no additional data structure is maintained, just insert u in $Adj(v)$ and v in $Adj(u)$. □

Evidently the classical recognition algorithm [13] can 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 2. In particular, the answer *no* can be obtained much faster, since the test $Adj(u) \cap Adj(v) \neq \emptyset$, which can be accomplished in $O(n)$, 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, depicted in Figure 1. The following experiment is repeated 2^{15} times: starting at a randomly generated tree with $n = 100$ vertices, feasible edges are inserted until the upper limit $n(n - 1)/2 = 4950$ is reached. At each iteration, several candidate edges are attempted, until an edge is found that can be added to the graph without violating chordality. Two kinds of failures may arise:

- *Type-1 failures* (represented by white bars): a pair of vertices corresponding to an existent edge (or loop) is selected and discarded;
- *Type-2 failures* (represented by black bars): the edge is new, but its insertion violates chordality, hence it is also discarded.

As the graph grows in density, the number of failures increases. Although the number of Type-2 failures tends to decrease, Type-1 failures dominate the generation process and most of the execution time is wasted on discarding repeated edges. However, each Type-1 failure takes only $O(1)$ time to be processed.

The most important remark about this experiment is that the results obtained are quite independent from the algorithm that implements the operation *insert_query*, being actually inherent to the generation method.

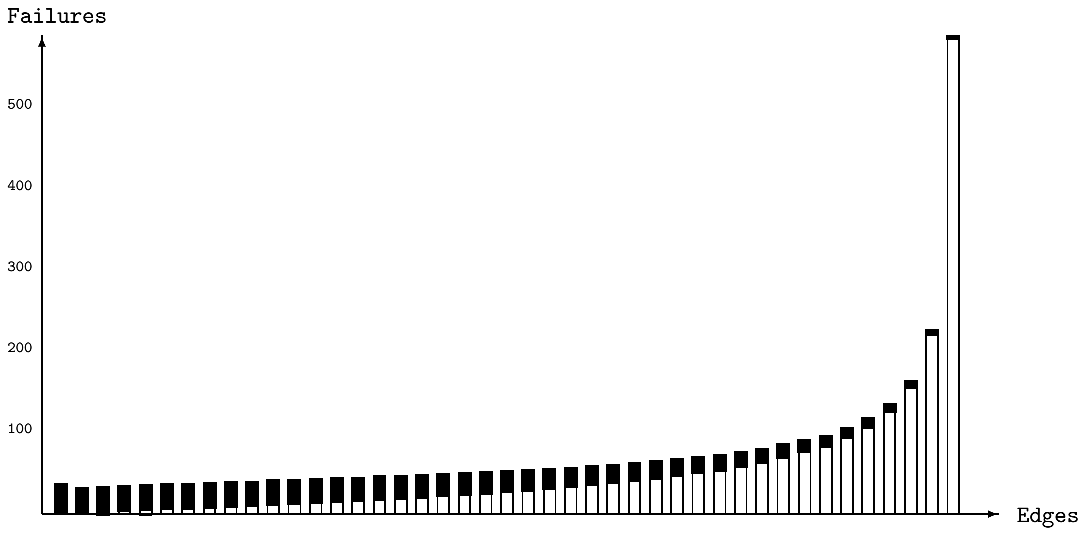

Figure 1: Type-1 and Type-2 failures

4 Generation of Chordal Graphs by Expansion and Merging of Maximal Cliques

The second generation method relies strongly on the clique-structure of a chordal graph and maintains an auxiliary data structure to allow fast identification and selection of cliques.

4.1 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*

$$\mathcal{JOIN}_{(Q,v)}(G) = (V \cup \{v\}, E \cup \{(v, x) | x \in Q\})$$

is also chordal.

Proof. Just notice that v is a simplicial vertex of $\mathcal{JOIN}_{(Q,v)}(G)$. 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 $\mathcal{JOIN}_{(Q,v)}(G)$ and, by Theorem 1, $\mathcal{JOIN}_{(Q,v)}(G)$ is chordal. □

Based upon this property, the algorithm for generating a connected chordal graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, has two steps:

- *Initialization step.* Start with the trivial graph $G = (\{v_1\}, \emptyset)$, $v_1 \in V$.
- *Clique-expansion step.* For $v \leftarrow v_2, \ldots, v_n$, perform: choose a clique $Q \subseteq V$ of G and replace G with $\mathcal{JOIN}_{(Q,v)}(G)$, linking v to all vertices of Q.

In order to compute $\mathcal{JOIN}_{(Q,v)}(G)$ 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 the *clique-tree.* Recall from the definition presented in Section 2 that each edge of the 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 so 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 $|Q|$.

It can be noticed that the exact number of edges of the graphs obtained by this method cannot be set *a priori*, since it depends on the sizes of the cliques selected during the process. In Table 1, we summarize some experimental results about the generation of large graphs. 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.

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

Table 1: Experimental Results

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 the algorithm produces mostly sparse graphs with small maximal cliques.

In order to cope with this drawback, a third step is appended to the algorithm, in which maximal cliques are merged into larger ones:

- *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.

When two cliques Q' and Q'', adjacent in the clique-tree, are collapsed, exactly

$$(|Q'| - weight(Q', Q'')) \times (|Q''| - weight(Q', Q''))$$

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. If an upper bound for the number of edges is given as input, the *Clique-merging* step can be interrupted as soon as this limit is reached.

The new results are shown in Table 2. For $n = 10000$, the generated graphs can have at most $\mathcal{M} = n(n-1)/2 = 49995000$ edges. Several upper bounds for the number of edges (M) are tested, varying in the range $n, \ldots, \mathcal{M}$. For each of these values, 2^{15} graphs are generated and the following informations are shown: the average number of edges (column 2), the standard deviation (column 3), the average size of the maximum clique (column 4) and the average number of clique merges (column 5).

$n = 10000$, $\mathcal{M} = n(n-1)/2 = 49995000$						
M	$(M/\mathcal{M})$	$\overline{m}$	$(\overline{m}/\mathcal{M})$	*sd*	*maxclique*	*merges*
10000	(< 1%)	21135.40	(< 1%)	289.86	9.08	0.00
15000	(< 1%)	21135.40	(< 1%)	289.86	9.08	0.00
20000	(< 1%)	21135.40	(< 1%)	289.86	9.08	0.00
25000	(< 1%)	25000.00	(< 1%)	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

Table 2: More Experimental Results

The results in Table 2 can be understood as follows:

- in the first section, very sparse graphs ($n \leq M < 2.2n$) 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 $M \geq 2.5n$ 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.

4.2 Implementation and Complexity

The proposed algorithm can be viewed as a 2-phase process: phase 1 is responsible for building the clique-tree; only in phase 2 the corresponding connected chordal graph is obtained.

The inputs of the algorithm are: the number of vertices n and the upper bound for the number of edges M. 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 $\mathcal{Q}_i$, $1 \leq i < n$;
- $\mathcal{L}$ 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;
- A is an auxiliary array of integers, with positions in the range $1, \ldots, n$.

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

```
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 $i$, $1 \leq i \leq \ell$;
        Choose $t$, $1 \leq t \leq \mathcal{S}_i$;
        if $t = \mathcal{S}_i$ then
        begin
            $\mathcal{Q}_i \leftarrow [v] \;||\; \mathcal{Q}_i$;     { an old clique is expanded }
            $\mathcal{S}_i \leftarrow \mathcal{S}_i + 1$
        end
        else begin
            Choose $Q \subseteq \mathcal{Q}_i$ with $|Q| = t$;
            $\ell \leftarrow \ell + 1$;
            $\mathcal{Q}_\ell \leftarrow [v] \;||\; Q$;     { a new clique is created }
            $\mathcal{S}_\ell \leftarrow t + 1$;
            $\mathcal{L} \leftarrow \mathcal{L} \;||\; [(i, \ell, t)]$
        end;
        $m \leftarrow m + t$
    end
end;
```

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 $\mathcal{Q}$;

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 $\mathcal{Q}_i$, $1 \leq i \leq \ell$:

Lemma 5. *After the execution of expand_cliques*

$$\sum_{i=1}^{\ell} |\mathcal{Q}_i| \leq n + m_1.$$

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

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

Hence

$$\sum_{i=1}^{\ell} |\mathcal{Q}_i| = \sum_{v=1}^{n} \Delta_v \leq 1 + \sum_{v=2}^{n} [t_{(v)} + 1] = n + \sum_{v=2}^{n} t_{(v)} = n + m_1. \square$$

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. $\square$

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* ([14]). 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 ← $\{\{1\},\{2\},\ldots,\{\ell\}\}$;
    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 $m + \Delta \times \delta \leq M$ 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 *FIND*s and at most one *UNION* are performed. Since there is at most $n - 2$ iterations, less than $2n$ *FIND*s and $n - 1$ *UNION*s are executed. The overall complexity is then $O(n\alpha(2n, n))$ [14]. $\square$

It is important to observe that the total number of elements, given by the sum $\sum_{i=1}^{\ell} |\mathcal{Q}_i|$, 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 *merge_cliques*, the non-empty lists in the array $\mathcal{Q}$ have repeated elements and a packing step is required:

```
procedure pack_lists;
begin
    for $i \leftarrow 1, 2, \ldots, n$ do $A_i \leftarrow 0$;
    for $i \leftarrow 1, 2, \ldots, \ell$ do
    begin
        $Q \leftarrow [\,]$;
        forall $v \in \mathcal{Q}_i$ do
            if $A_v \neq i$ then
                $Q \leftarrow Q \mid\mid [v]$, $A_v \leftarrow i$;
        $\mathcal{Q}_i \leftarrow Q$
    end
end;
```

Lemma 8. *Pack_lists performs in time $O(n + m_1)$.*

Proof. Because all lists are traversed, the complexity is $O(\sum_{i=1}^{\ell} |\mathcal{Q}_i|)$. Since the total number of elements, calculated in Lemma 5, is not affected by *merge_cliques*, the result holds. $\square$

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.

4.3 Constructing the Graph from the Clique Tree

Phase 2 is a conversion phase: the desired connected chodal 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. Procedure *make_graph_from_free* accomplishes this task.

```
procedure make_graph_from_tree;
begin
    for $i \leftarrow 1, 2, \ldots, n$ do $A_i \leftarrow 0$;
    for $i \leftarrow 1, 2, \ldots, \ell$ do
    begin
        if $\mathcal{S}_i \neq 0$ do
        begin
            $\mathcal{O} \leftarrow [v \in \mathcal{Q}_i | A_v = 1]$;
            $\mathcal{N} \leftarrow [v \in \mathcal{Q}_i | A_v = 0]$;
            make_clique ($\mathcal{N}$);
            forall $v \in \mathcal{N}$ do
            begin
                $A_v \leftarrow 1$;
                forall $w \in \mathcal{O}$ do
                    make_edge ($v$, $w$)
            end
        end
    end
end;
```

At the i-th iteration of *make_graph_from_tree*, the vertices of $\mathcal{Q}_i$ 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. The array A helps keeping track of which vertices have already been processed.

The call on *make_clique* is supposed to produce a clique with all vertices belonging to $\mathcal{N}$, issuing all needed edges among them; however, the vertices belonging to $\mathcal{O}$ need only be linked to the vertices in $\mathcal{N}$. The reason is explained in the following propositions.

Let $\mathcal{Q}_1$ be chosen as root of the clique-tree. Examining the structure of the triples added to $\mathcal{L}$ in *expand_cliques* and how the *UNION*s are performed in *merge_cliques*, we can conclude that, if $\mathcal{Q}_j$ is an ancestor of $\mathcal{Q}_i$, then $j \le i$. So, when a node is being processed in *make_graph_from_tree*, all its ancestors have already been handled, since $i \leftarrow 1, 2, \dots, \ell$.

Let $lca(\mathcal{Q}', \mathcal{Q}'')$ denote the *least common ancestor* of nodes $\mathcal{Q}'$ and $\mathcal{Q}''$ in the clique-tree.

Lemma 9. *At iteration i of make_graph_from_tree, if $v, w \in \mathcal{O} \subset \mathcal{Q}_i$, $v \neq w$, then there exists an ancestor $\mathcal{Q}_j$ of $\mathcal{Q}_i$, such that $v, w \in \mathcal{Q}_j$.*

Proof. If $v, w \in \mathcal{O}$, both vertices have already appeared in previously processed nodes. Thus there must exist $\mathcal{Q}_a$ and $\mathcal{Q}_b$ such that $v \in \mathcal{Q}_a$, $w \in \mathcal{Q}_b$ and $1 \le a, b < i$.

By the intersection property of clique-trees, since $v \in \mathcal{Q}_i \cap \mathcal{Q}_a$, v must belong to every node on the path between $\mathcal{Q}_i$ and $\mathcal{Q}_a$; in particular, $v \in lca(\mathcal{Q}_i, \mathcal{Q}_a)$. Similarly, $w \in lca(\mathcal{Q}_i, \mathcal{Q}_b)$. As $lca(\mathcal{Q}_i, \mathcal{Q}_a)$ and $lca(\mathcal{Q}_i, \mathcal{Q}_b)$ are both ancestors of $\mathcal{Q}_i$, one of the following conditions must hold:

- either $lca(\mathcal{Q}_i, \mathcal{Q}_a)$ is an ancestor of $lca(\mathcal{Q}_i, \mathcal{Q}_b)$
  in this case, $lca(\mathcal{Q}_i, \mathcal{Q}_b)$ lies on the path between $\mathcal{Q}_i$ and $lca(\mathcal{Q}_i, \mathcal{Q}_a)$, so v also belongs to $lca(\mathcal{Q}_i, \mathcal{Q}_b) = \mathcal{Q}_j$;
- or $lca(\mathcal{Q}_i, \mathcal{Q}_b)$ is an ancestor of $lca(\mathcal{Q}_i, \mathcal{Q}_a)$
  likewise w also belongs to $lca(\mathcal{Q}_i, \mathcal{Q}_a) = \mathcal{Q}_j$.

In both cases, an ancestor $\mathcal{Q}_j$ of $\mathcal{Q}_i$ was found, to which v and w belongs. □

Lemma 10. *At the end of each iteration, the edges belonging to the maximal cliques processed so far have all been issued only once.*

Proof. By induction on i, the number of the iteration:

- For $i = 1$, the result holds, since $\mathcal{Q}_1 = \mathcal{N}$.
- Suppose this is valid for $i < k$.
- Consider the k-th iteration and the partition $\mathcal{Q}_k = \mathcal{N} \cup \mathcal{O}$. Let $v, w \in \mathcal{Q}_k$, $v \neq w$:
  - if $v, w \in \mathcal{N}$, the edge (v, w) is issued by the call to *make_clique*;
  - if $v \in \mathcal{N}$ and $w \in \mathcal{O}$ (or $w \in \mathcal{N}$ and $v \in \mathcal{O}$), the edge (v, w) is issued by a call to *make_edge*;
  - if $v, w \in \mathcal{O}$, by Lemma 9, v, w also belong to an ancestor of $\mathcal{Q}_k$. Since ancestors are always processed before their descendants, by the inductive hypothesis, the edge (v, w) has already been issued. The algorithm takes no action.

  In all cases, for each vertex $v \in \mathcal{N}$, $A_v \leftarrow 1$, so that v will belong to $\mathcal{O}$ in all subsequent iterations. □

Thus *make_graph_from_tree* issues only once each edge belonging to the graph and Lemma 11 shows its time complexity.

Lemma 11. *Make_graph_from_tree performs in time $O(m)$, where m is the number of edges of the generated chordal graph.*

Proof. The total number of steps performed by all executions of *make_clique* and *make_edge* is $O(m)$, since these procedures issue exactly once each edge of the chordal graph. The complexity is then

$$O(n + \sum_{i=1}^{\ell} \mathcal{S}_i + m) = O(m),$$

since $\sum_{i=1}^{\ell} \mathcal{S}_i \le n + m_1 \le n + m$. □

Including the conversion phase, the overall complexity of the second generation method is:

$$O(n + m_1 + n\alpha(2n, n) + m) = O(m + n\alpha(2n, n)).$$

5 Conclusions

We have presented two methods for the generation of connected chordal graphs. The first one adds successively new edges to an existent chordal graph, starting at a tree to ensure conectivity. Thus pairs of vertices are randomly selected and the insertion of the corresponding edges is attempted, preserving chordality. As the graph grows denser, the probability of choosing a pair of vertices which already corresponds to an existent edge increases, so that most of the execution time is wasted on discarding repeated edges. Therefore, this first method is suitable for generating sparse graphs and has the main advantage that the final number of edges can be precisely established *a priori*.

The second method uses the clique-tree as an intermediate representation of the chordal graph. Maximal cliques are expanded and contracted, yielding a final clique-tree, that must be converted to the adjacency list graph format. Although the final number of edges cannot be set in advance, an upper limit is given as input and experimental results show that the average number of edges obtained is close enough of the desired bound. This second method turns out to be more adequate to the generation of dense graphs.

References

1. Alberts, D., Cattaneo, G., Italiano, G.F., An Empirical Study of Dynamic Graph Algorithms, ACM J. of Experimental Algorithms **6**: 1-39, 1997
2. Araújo, L.H., Algoritmos Dinâmicos para Manutenção de Grafos Cordais e Periplanares, Ph.D. Thesis, Coppe-Produção, Universidade Federal do Rio de Janeiro, RJ, Brasil, 2004
3. Blair, J. R. S., Peyton, B., An Introduction to Chordal Graphs and Clique Trees In Graph Theory and Sparse Matrix Computation, IMA **56**: 1-29, 1993
4. Berry, A., Heggernes, P., Villanger, Y., A Vertex Incremental Approach for Dynamically Maintaining Chordal Graphs, Proc. of ISAAC'03, 2003
5. Chandran, L. S., Ibarra, L., Ruskey F., Sawada, J., Generating and Characterizing the Perfect Elimination Orderings of a Chordal Graph Theoretical Computer Science, **307**: 303-317, 2003
6. Deo, N., Micikevicius, P., A New Encoding for Labeled Trees Employing a Stack and a Queue, Bulletin of the ICA, **34**:77-85, 2002
7. Eppstein, D., Galil, Z., Italiano, G. F., Sparsification - A Technique for Speeding up Dynamic Graph Algorithms, Journal of the ACM, **44**:669-696, 1997
8. Galinier, P., Habib, M., Paul, C., Chordal Graphs and Their Clique Graphs, Lecture Notes in Computer Science **1017**: 358-371, 1995
9. Gavril,F., Algorithms for Minimum Coloring, Minimum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph, SIAM J. of Comput., **1**: 180-187, 1972
10. Golumbic, M.C., Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980
11. Ibarra, L., Fully Dynamic Algorithms for Chordal Graphs and Split Graphs, Technical Report DCS-262-IR, University of Victoria, 2000
12. Kumar, P. S., Madhavan, C. E. V., Clique Tree Generalization and New Subclasses of Chordal Graphs Discrete Applied Mathematics, **117**: 109-131, 2002
13. Rose, D.J., Tarjan, R.E., Lueker, G., Algorithmic Aspects of Vertex Elimination on Graphs, SIAM J.Comput., **5**: 266-283, 1976
14. Tarjan, R.E., Data Structures and Network Algorithms, SIAM, Philadelphia, 1983
15. Tinhofer, G., Generating Graphs Uniformly at Random, Computing Supp., **7**: 235-255, 1990