Linear-time generation of random chordal graphs*

Oylum Şeker[1], Pinar Heggernes[2], Tınaz Ekim[1], and Z. Caner Taşkın[1]

[1] Department of Industrial Engineering, Boğaziçi University, Istanbul, Turkey.
{oylum.seker,tinaz.ekim,caner.taskin}@boun.edu.tr
[2] Department of Informatics, University of Bergen, Norway.
pinar.heggernes@uib.no

Abstract. Chordal graphs form one of the most well studied graph classes. Several graph problems that are NP-hard in general become solvable in polynomial time on chordal graphs, whereas many others remain NP-hard. For a large group of problems among the latter, approximation algorithms, parameterized algorithms, and algorithms with moderately exponential or sub-exponential running time have been designed. Chordal graphs have also gained increasing interest during the recent years in the area of enumeration algorithms. Being able to test these algorithms on instances of chordal graphs is crucial for understanding the concepts of tractability of hard problems on graph classes. Unfortunately, only few published papers give algorithms for generating chordal graphs. Even in these papers, only very few methods aim for generating a large variety of chordal graphs. Surprisingly, none of these methods is based on the "intersection of subtrees of a tree" characterization of chordal graphs. In this paper, we give an algorithm for generating chordal graphs, based on the characterization that a graph is chordal if and only if it is the intersection graph of subtrees of a tree. The complexity of our algorithm is linear in the size of the produced graph. We give test results to show the variety of chordal graphs that are produced, and we compare these results to existing results.

1 Introduction

Algorithms particularly tailored to exploit properties of various graph classes have formed an increasingly important area of graph algorithms during the last five decades. With the introduction of relatively new theories for coping with NP-hard problems, like parameterized algorithms, algorithmic research on graph classes has become even more popular recently, and the number of results in this area appearing at international conferences and journals is now higher than ever. One of the most studied graph classes in this context is the class of chordal graphs, i.e., graphs that contain no induced cycle of length 4 or more. Chordal graphs arise in practical applications from a wide variety of unrelated fields, like

* This work is supported by the Research Council of Norway, Bogazici University Research Fund (grant 11765), and Turkish Academy of Sciences GEBIP award.

sparse matrix computations, database management, perfect phylogeny, VLSI, computer vision, knowledge based systems, and Bayesian networks [6, 13, 21, 24, 26]. This graph class that first appeared in the literature as early as 1958 [14], has steadily increased its popularity, and there are now more than 20 thousand references on chordal graphs according to Google Scholar.

With a large number of existing algorithms specially tailored for chordal graphs, it is interesting to note that not much has been done to test these algorithms in practice. Very few such tests are available as published articles [2, 18, 22]. In particular, there seems to be no efficient and all-purpose chordal graph generator available. Most of the work in this direction involves generating chordal graphs tailored to test a particular algorithm or result [2, 22]. This is a clear shortcoming for the field, and it was even mentioned as an important open task at a Dagstuhl Seminar [16]. Until some years ago, most of the algorithms tailored for chordal graphs had polynomial running time, and testing was perhaps not crucial. Now, however, many parameterized and exponential-time algorithms exist for chordal graphs, for problems that remain hard on this graph class, see e.g., [4, 12, 19, 20]. The proven running times of such algorithms might often be too high compared to the practical running time. Just to give some examples from the field of enumeration, there are now several algorithms and upper bounds on the maximum number of various objects in chordal graphs [1, 11, 12]. However, the lower bound examples at hand usually do not match these upper bounds. Tests on random chordal graphs is a good way of getting better insight about whether the known upper bounds are too high or tight.

In this paper we present an algorithm for generating random chordal graphs. The algorithm is based on the characterization that a graph is chordal if and only if it is the intersection graph of subtrees of a tree. Surprisingly, this characterization does not seem to have been exploited for chordal graph generation earlier. The running time of our algorithm is linear in the size of the generated graph, and it generates a large variety of chordal graphs, where the variety is measured using the characteristics of maximal cliques as already used in [22]. After proving the correctness and the time complexity, we give extensive tests to demonstrate the kind of chordal graphs that our algorithm generates. We compare our tests with existing test results, and we implement one of the earlier proposed methods and include this in our tests. According to these tests, our algorithm outperforms previous algorithms both with respect to complexity and with respect to the richness of the family of the generated chordal graphs. Observe that graph isomorphism is as hard on chordal graphs as on general graphs [17], which adds to the difficulty of producing chordal graphs uniformly random. Still our algorithm is able to generate every chordal graph with positive probability.

2 Background, terminology and existing algorithms

In this section we give the necessary background on chordal graphs, as well as a short review of the existing algorithms for chordal graph generation. We work with simple and undirected graphs, and we use standard graph terminology. We let n denote the number of vertices and m denote the number of edges of a graph. A *maximal clique* is an inclusion-wise maximal set of vertices that are pairwise adjacent. An ordering $(v_1, v_2, \ldots, v_n)$ of the vertices of a graph is a *perfect elimination order (peo)* if the set of higher numbered neighbors of each vertex forms a clique.

Let $F = \{S_1, S_2, \ldots, S_n\}$ be a family of sets from the same universe. A graph G is called an *intersection graph of F* if there is a bijection between the set of vertices $\{v_1, v_2, \ldots, v_n\}$ of G and the sets in F such that v_i and v_j are adjacent if and only if $S_i \cap S_j \neq \emptyset$, for $1 \leq i, j \leq n$. In the special case where there is a tree T such that each set in F corresponds to the vertex set of a subtree of T, then G is called the *intersection graph of subtrees of a tree*. In this case, we call T a *host tree* for G.

A tree T with a bijection between its vertex set and the set of maximal cliques of a graph G, is called a *clique tree* of G if, for every vertex v of G, the set of vertices of T that correspond to the cliques containing v induce a connected subtree of T.

A graph is *chordal* if it contains no induced cycle of length 4 or more. A chordal graph on n vertices has at most n maximal cliques [7]. Chordal graphs have many different characterizations. For our purposes, the following will be sufficient.

Theorem 1 ([5, 8–10]). *Let G be a graph. The following are equivalent.*

- *G is chordal.*
- *G has a perfect elimination order.*
- *G is the intersection graph of subtrees of a tree.*
- *G has a clique tree.*

Especially the last two points of Theorem 1 are crucial for our algorithm and its implementation. To make sure that there is no confusion between the vertices of G and the vertices of a host tree or a clique tree, we will from now on refer to vertices of a tree as *nodes*.

Rose, Tarjan, and Lueker [25] gave an algorithm called Maximal Cardinality Search (MSC) that creates a perfect elimination order of a chordal graph in time $O(n + m)$. Blair and Peyton [3] gave a modification of MCS to list all the maximal cliques of a chordal graph in time $O(n + m)$. Implicit in their proofs is the following well-known fact that is not often highlighted on its own.

Lemma 1 ([3, 25]). *The sum of the sizes of the maximal cliques of a chordal graph is $O(n + m)$.*

Next, we briefly mention the algorithms for generating chordal graphs from the works of Andreou, Papadopoulou, Spirakis, Theodorides, and Xeros [2]; Pemmaraju, Penumatcha, and Raman [22]; and Markenzon, Vernet, and Araujo [18]. Some of these algorithms create very limited chordal graphs, which is either mentioned by the authors or clear from the algorithm. Thus, in the following we only mention the algorithms that are general enough to be interesting in our context.

It should also be noted that the purpose of Andreou et al. [2] is not to obtain general chordal graphs, but rather chordal graphs with a known bound on some parameter. One of the algorithms that they propose starts from an arbitrary graph and adds edges to obtain a chordal graph. How the edges are added is not given in detail, but note that there are many algorithms for generating a chordal graph from a given graph by adding a minimal set of edges and their running time is usually $O(nm)$, far from linear [15]. Andreou et al. [2] do not report on the quality of chordal graphs obtained by this method.

We highlight below the algorithms that are the most promising with respect to generating random chordal graphs. In addition to these, there is an $O(n^2)$-time algorithm by Markenzon et al. [18] that generates a random tree and adds edges to this tree until a chordal graph with desired edge density is obtained. However, no test results about the quality of the generated graphs is given.

Alg 1 [2]. The algorithm constructs a chordal graph by using a peo. At every iteration, a new vertex is added and made adjacent to a random selection of already existing vertices. Then necessary edges are added to turn the neighborhood of the new vertex into a clique. No test results are given in the paper about the quality of the chordal graphs this algorithm produces. As we found the algorithm interesting, we have implemented it, and we compare the resulting graphs to those generated by our algorithm in Section 4.

Alg 2 [18, 22]. The algorithm starts from a single vertex. At each subsequent step, a clique C in the existing graph is chosen at random, and a new vertex is added adjacent to exactly the vertices of C. The inverse of the order in which the vertices are added is a peo of the final graph. It is observed by the authors of both papers that this procedure results in chordal graphs with approximately $2n$ edges experimentally. They propose the following changes:

Alg 2a [18] modifies the above generated graph by randomly choosing maximal cliques that are adjacent according to the clique tree and merging these until desired edge density is obtained. Some test results about the graphs generated by Alg 2a are provided in [18]. Although these tests are not as comprehensive as the tests we give on our algorithm in Section 4, we compare our results to those of [18] as best we can. The running time of Alg 2a is $O(m + n\alpha(2n, n))$.

Alg 2b [22] is a modification of Alg 2 in a different way: instead of randomly choosing a clique, a maximum clique is chosen and a random subset of it is made adjacent to the new vertex. Although test results for Alg 2b are provided in [22], the authors acknowledge that the produced graphs are still very particular with very few large maximal cliques and many very small maximal cliques. For this reason, we do not include Alg 2b in our comparisons.

3 Generating chordal graphs using subtrees of a tree

We find it surprising that the intersection graph of subtrees of a tree characterization of chordal graphs has not been used for generation. One reason could be that this characterization does not give a direct way to decide the number of edges. However, as we will see, edge density can be regulated by adjusting the sizes of the generated subtrees. We are now ready to present our main algorithm for generating chordal graphs on n vertices:

```
Algorithm ChordalGen
Input: Two integers $n$ and $k$
Output: A chordal graph $G$ on $n$ vertices and $m$ edges
Generate a tree $T$ on $n$ nodes uniformly at random
Create $n$ random subtrees of $T : \{T_1, \dots, T_n\}$ of average size $k$
Output as $G$ the intersection graph of the trees $\{T_1, \dots, T_n\}$
```

By Theorem 1 the graph generated by Algorithm ChordalGen is chordal. We want to show that this high level definition of the algorithm is general and can create any chordal graph. The proof of the following lemma is already implicit in the proofs of the relevant parts of Theorem 1. We give it here, as it will also be of help in the explanation of the running time of our algorithm.

Lemma 2. *Let G be a chordal graph on n vertices and m edges. There is an execution of Algorithm ChordalGen that generates G.*

Proof. First of all we want to show that there is a host tree T on exactly n nodes, and a set of n subtrees of T, such that G is the intersection graph of these subtrees. Let T' be a clique tree of G. Let us call the vertices of G: $v_1, v_2, \dots, v_n$. Define subtree T'_i to be the subtree of T' that corresponds to the nodes (maximal cliques) that contain vertex v_i, for $1 \leq i \leq n$. By the definition of a clique tree, T' has at most n nodes and each T'_i is a connected subgraph of T'. If T' has less than n nodes, we can add new nodes adjacent to arbitrary nodes of T' until we get a new tree T with exactly n nodes. The subtrees stay the same. As two vertices are adjacent in G if and only if they appear together in a clique, G is the intersection graph of subtrees $T'_1, \dots, T'_n$ of T. Finally, we simply let k be the average size of the subtrees T_i. $\square$

The most interesting part of the algorithm is the generation of the subtrees of T. For this, we propose an algorithm called SubtreeGen as follows.

```
Algorithm SubtreeGen
Input: A tree $T$ on $n$ nodes and an integer $k$
Output: A set of $n$ subtrees of $T$ of average size $k$
for $i = 1$ to $n$ do
  Select a random node $x$ of $T$ and set $T_i = \{x\}$
  Select a random integer $k_i \leq n$ between 1 and $2k - 1$
  for $j = 1$ to $k_i - 1$ do
    Select a random node $y$ of $T_i$ that has neighbors in $T$ outside of $T_i$
    Select a random neighbor $z$ of $y$ outside of $T_i$ and add $z$ to $T_i$
Output $\{T_1, T_2, \dots, T_n\}$
```

Lemma 3. *The running time of Algorithm SubtreeGen is $O(n + \sum_{1=i}^{n} |T_i|)$.*

Proof. Observe first that each subtree T_i is simply a list of nodes of T. We show that after an initial $O(n)$ preprocessing time, each subtree T_i can be generated in time $O(|T_i|)$. For this, we need to be able to add a new node to T_i in constant time, at each of the $k_i - 1$ steps.

As selecting random elements in constant time is easier when accessing the elements of an array directly by indices, we start with copying the nodes of T into an array A of size n, and copying the adjacency list of each node x into an array A_x of size $deg(x)$. This can clearly be done in total time $O(n)$ since T is a tree.

In general, selecting an unselected element of a set at random can be done easily in constant time if the set is represented with an array. Let us say we have an array S of t elements. We keep a separation index s that separates the selected elements from the not selected ones. At the beginning s is 1. At each step, we generate a random integer r between s and t. $S[r]$ is our randomly selected element. Then we swap the elements $S[s]$ and $S[r]$ and increase s by 1.

We can use this method both for selecting a node y of T_i that still has neighbors outside and for selecting a neighbor z of y that has not yet been selected. For the latter, whenever we select a neighbor z of y, we move z to the first part of the array A_x using swap. When the separation index reaches the degree of y then we know that y should not be selected to grow the subtree T_i at later steps. Representing T_i with an array of size k_i, we can use the same trick to move y to a part of the array that we will not select from. Also, when z is added, we can check whether it is a leaf in T in constant time, and immediately move it to the irrelevant part of the array for T_i if so, since z can then not be used for growing T_i at later steps. It is sufficient to check that z is a leaf of T, because otherwise it must have neighbors outside of T_i, since T is a tree and we cannot have cycles. When the generation of T_i is finished, the separation indices of each of its nodes should be reset before we start generating T_{i+1}. The adjacency arrays need not be reorganized, as we will anyway be selecting neighbors at random.

Note that we do not need this trick to select an initial node x of each subtree T_i, because we should indeed be able to select the same node several times (and grow another subtree from it perhaps in a different way).

With the described method, each step of Algorithm SubtreeGen takes $O(1)$ time, in addition to initial $O(n)$ time to copy the information into appropriate arrays. Thus the total running time is $O(n + \sum_{1=i}^{n} |T_i|)$. □

We can now prove the total running time for chordal graph generation.

Theorem 2. *Algorithm ChordalGen generates a chordal graph with n vertices and m edges in time $O(n + m)$.*

Proof. Rodionov and Choo [23] prove that the following procedure which runs in $O(n)$ time generates a tree T on n nodes uniformly at random: start with a tree T that contains only one node. Then repeat $n - 1$ times the following: pick a random node x of T and add a new node adjacent to it.

We use Algorithm SubtreeGen to generate n subtrees of T. By Lemma 3, this adds to our running time an order of the sum of the sizes of the generated subtrees.

To each subtree T_i, we associate a vertex v_i of G. In addition to storing the node lists T_i, we also store in the nodes of T information about which subtrees contain that node. More precisely, at node x of T, we store the following list: $\{v_j \mid T_j \text{ contains } x\}$. Observe that this is equivalent to each node of T representing a clique by storing the list of graph vertices that are contained in this clique. By Lemma 2, T then contains the information that corresponds to a clique-tree of G. By Lemma 1 the sum of the sizes of the lists contained at the nodes is $O(n+m)$. As we only used the lists T_i to generate this information, the sum of the sizes of the subtrees is also $O(n+m)$. By methods described by Blair and Peyton [3] it is possible to turn T into a proper clique tree for G in time $O(n+m)$. Thus, in total $O(n+m)$ time we both have a representation of our output graph G and a list of maximal cliques of it. It could, however, be desirable to output an adjacency list representation for G. Markenzon et al. [18], using the methods of Blair and Peyton [3], explain how this can be done in $O(n+m)$ time. $\square$

As argued in the proof of Theorem 2, the sum of the sizes of the generated subtrees is $O(n+m)$. In our test results, we give both k and the resulting number of edges, m, to give an indication of how k affects the density of the generated graph. It is also possible to supply Algorithm SubtreeGen with a vector of n subtree sizes $\{k_1, k_2, \ldots, k_n\}$ to generate subtrees of exactly desired size. This does not change the running time of the algorithm. Within the same running time, even more user control is possible, like limiting the maximum degree of each subtree, if so desired, for instance to generate intersection graphs of paths in a tree. In fact, a completely different method for subtree generation can be plugged in instead of SubtreeGen in Algorithm ChordalGen. This gives the possibility of fine-tuning the generation towards designated purposes. In the concluding section, we mention a few other ideas for subtree generation.

4 Experimental results

In this section, we give extensive test results to show what kind of chordal graphs are generated by Algorithm ChordalGen. In Table 1 we show how the selection of parameter k affects the number of resulting edges and connected components. We also present the number of maximal cliques, and the minimum, maximum, and mean size for the maximal cliques, along with their average standard deviation. For each parameter pair n and average subtree size k, we performed ten independent runs and report the average values across those then runs. For each n, we tuned the average subtree sizes in order to approximately achieve some selected average edge density values of 0.01, 0.1, 0.5, and 0.8, where edge density is defined as $\frac{m}{n(n-1)/2}$.

We want to compare our results to the results showing the kind of chordal graphs that are generated by Alg 2a [18]. Note, however that, the results given

Table 1: Experimental results of Algorithm ChordalGen

n	avg subtree size (k)	density	# edges	# conn. comp.s	# maximal cliques	min clique size	max clique size	mean clique size	sd of clique sizes
1000	4.0	0.011	5646.6	16.5	355.7	1.0	21.4	6.16	3.41
1000	17.0	0.101	50374.8	1.0	169.6	5.2	134.1	30.62	20.26
1000	70.0	0.505	252237.8	1.0	77.6	29.6	474.0	140.24	92.23
1000	162.5	0.801	399906.4	1.0	49.1	74.8	726.2	313.27	163.59
2500	7.0	0.011	35289.6	3.0	680.5	1.1	54.0	11.58	6.96
2500	32.0	0.103	322433.4	1.0	299.0	10.5	344.8	62.07	44.47
2500	135.0	0.503	1572067.0	1.0	134.5	50.0	1196.3	291.23	206.63
2500	318.0	0.803	2509818.4	1.0	88.4	115.0	1866.1	639.35	385.94
5000	10.5	0.010	130255.1	1.2	1092.9	1.8	97.1	18.15	11.47
5000	50.5	0.098	1229487.3	1.0	476.4	15.3	650.9	100.99	77.59
5000	225.0	0.509	6361645.4	1.0	199.5	76.9	2531.3	504.05	381.98
5000	549.0	0.809	10114806.0	1.0	122.0	163.6	3695.8	1217.44	756.58
10000	16.0	0.010	506598.1	1.0	1745.7	3.4	203.5	29.05	19.98
10000	85.0	0.107	5338077.0	1.0	706.5	25.0	1366.8	181.86	148.12
10000	377.0	0.497	24832462.0	1.0	312.6	103.0	4871.6	861.59	681.79
10000	926.0	0.802	40101492.0	1.0	191.7	236.6	7294.4	2109.65	1394.89

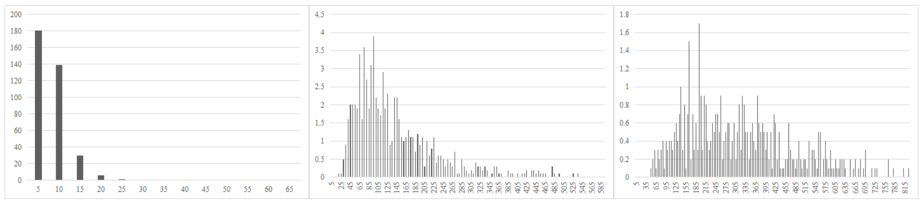
by [18] only contain graphs on 10000 vertices, with varying number of edges. Most metrics presented in [18] are about the number of edges. When it comes to the maximal cliques, they present only the average maximum clique size over the generated graphs for each edge density. Comparing these to our numbers we see that graphs corresponding to edge densities 0.01, 0.1, 0.5, and 0.8 of Alg 2a have average maximum clique sizes 727, 2847, 6875, and 8760, respectively. As can be seen from Table 1, these numbers are quite higher than the corresponding numbers for the graphs generated by Algorithm ChordalGen. In fact, studying the numbers more carefully, we can conclude that the maximum clique of a graph generated by Alg 2a contains almost all the edges of the graph. In the case of density 0.01, such a clique contains more than half of the edges, whereas in the case of higher densities, the largest clique contains more than 80, 94, and 95 percent of the edges, respectively. Thus there does not seem to be an even distribution of the sizes of maximal cliques of graphs generated by Alg 2a.

As we mentioned in Section 2, we also implemented Alg 1 [2]. In Table 2 we give results analogous to Table 1 for 1000, 2500, and 5000 vertices. In order to obtain results for Table 2 comparable to those given in Table 1, we wanted to have approximately the same edge density values. For this purpose, when determining in Alg 2 the number of neighbors of a vertex at each step, we multiplied the total number of candidate vertices with a coefficient between 0 and 1, which we call *upper bound coefficient*. A running time analysis for this algorithm has not been given [2]. With our implementation, this algorithm turned out to be too slow to allow testing graphs on 10000 vertices in reasonable time. However, already from the obtained numbers, we can reach a conclusion for Alg 1 similar to that on Alg 2a. Observe that the maximum clique sizes obtained for 5000 vertices by Alg 1, are comparable to the maximum clique sizes obtained for 10000 vertices by Algorithm ChordalGen. Hence, like Alg 2a, also Alg 1 seems to generate graphs with few big maximal cliques. As can be seen in Table 2, Alg 1 outputs connected

Table 2: Experimental results of our implementation of Alg 1 [2]

n	upper bound coef.	density	# edges	#conn. comp.s	# maximal cliques	min clique size	max clique size	mean clique size	sd of clique sizes
1000	0.00130	0.011	5368.6	1.0	935.1	2.0	56.0	4.7	8.74
1000	0.00300	0.103	51519.5	1.0	755.3	2.0	219.2	31.3	63.85
1000	0.01100	0.499	249288.5	1.0	405.2	2.0	561.8	184.3	231.81
1000	0.03500	0.808	403436.2	1.0	185.8	2.0	793.8	394.5	346.26
2500	0.00053	0.010	31978.1	1.0	2322.9	2.0	154.2	8.5	25.39
2500	0.00120	0.101	316107.4	1.0	1882.4	2.0	549.3	71.3	160.68
2500	0.00440	0.501	1565224.3	1.0	1005.7	2.0	1401.1	458.8	592.22
2500	0.01400	0.807	2519340.1	1.0	470.0	2.0	1980.7	988.2	887.66
5000	0.00027	0.011	134535.7	1.0	4628.4	2.0	320.7	16.0	56.66
5000	0.00062	0.107	1331285.2	1.0	3717.0	2.0	1144.5	144.1	339.34
5000	0.00220	0.503	6289143.9	1.0	2001.7	2.0	2804.3	919.5	1195.48
5000	0.00700	0.804	10049827	1.0	938.5	2.0	3945.8	1945.0	1787.89

chordal graphs for the selected set of average edge density values and number of vertices. The minimum size of the maximal cliques did not show any variation throughout our experiments and always turned out to be two. The consistency in this measure may be an additional indication of the lack of potential to produce a diverse range of chordal graphs.


(a) Results from Algorithm ChordalGen

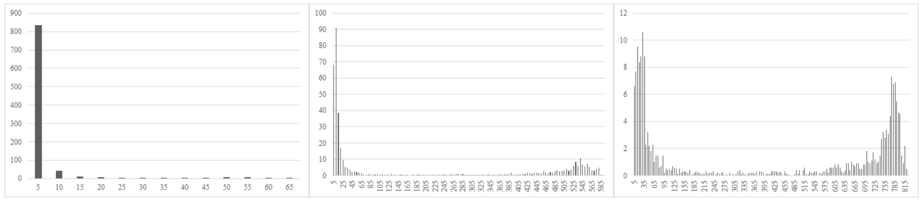
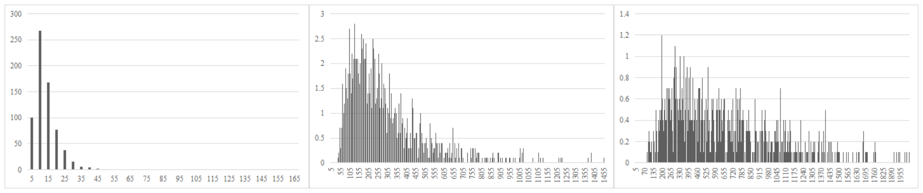
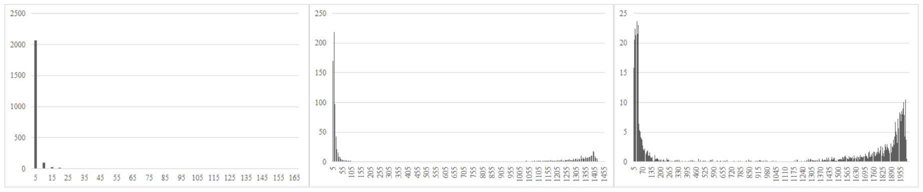

(b) Results from our implementation of Alg 1 [2]

Fig. 1: Histograms of maximal clique sizes for $n = 1000$ and average edge densities 0.01, 0.5, and 0.8 (from left to right)

We wanted to evidence the above conclusions by investigating how the sizes of the maximal cliques are distributed. Figures 1-3 show the average number of maximal cliques across ten independent runs in intervals of width five, for 1000, 2500, and 5000 vertices and varying edge densities. These figures consist of two sub-figures, and each subfigure is comprised of three histograms for three different average edge density values. The top sub-figures show the results from Algorithm ChordalGen and the second those from our implementation of Alg 1 [2]. For a given n and average density value, the ranges of x-axes are kept the same in order to render the histograms comparable. The y-axes, however, have different ranges because maximum frequencies in histograms corresponding to Alg 1 and Algorithm ChordalGen vary drastically. The ratios of the maximum frequencies of Alg 1 to those of Algorithm ChordalGen range roughly from 4 to 160.



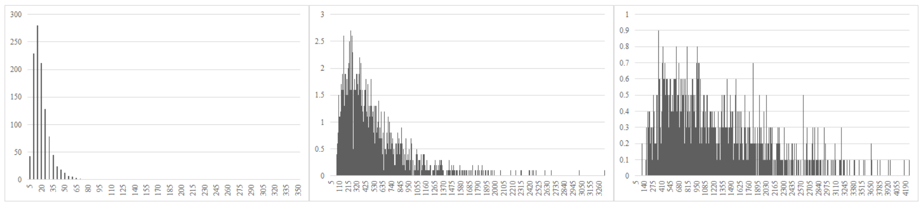
(a) Results from Algorithm ChordalGen



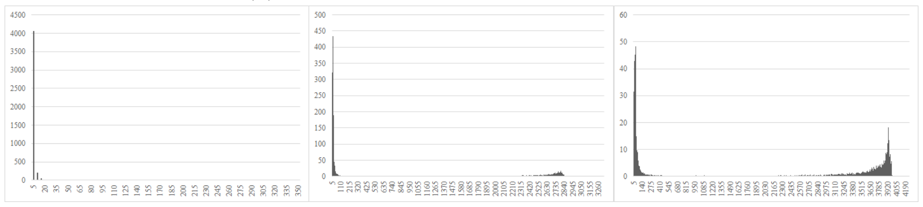
(b) Results from our implementation of Alg 1 [2]

Fig. 2: Histograms of maximal clique sizes for $n = 2500$ and average edge densities 0.01, 0.5, and 0.8 (from left to right)

As Figures 1-3 reveal, the vast majority of maximal cliques of graphs output by Alg 1 have sizes of 2 to 15. With the increase in edge densities, frequencies of large-size maximal cliques become visible relative to the dominant small clique frequencies; however, all but the extremes of the range is barely used regardless of selection of n and edge density. The restricted shape of the distribution of clique sizes indicates that Alg 1 likely produces chordal graphs of limited structure in general. Algorithm ChordalGen, however, does not demonstrate such bias toward the extremes over the range of its maximal clique sizes; output graphs contain maximal cliques of many different sizes. The fair dispersion of clique sizes of Algorithm ChordalGen suggests diversity of its output graphs, which is a desired characteristic of a random chordal graph generator.



(a) Results from Algorithm ChordalGen



(b) Results from our implementation of Alg 1 [2]

Fig. 3: Histograms of maximal clique sizes for $n = 5000$ and average edge densities 0.01, 0.5, and 0.8 (from left to right)

5 Concluding remarks and future work

Algorithm ChordalGen is the first linear-time method in literature for generating chordal graphs. Furthermore, as it can be seen from the test results, it generates the most varied chordal graphs, compared to existing methods. The algorithm is also very general and flexible in the sense that many different methods for subtree generation can be plugged in.

As already mentioned in Section 3, we can further fine-tune the generation of subtrees for special purposes, and we can use other methods for subtree generation in Algorithm ChordalGen instead of SubtreeGen. Two such possible ideas are generating the subtrees by selecting a set of random nodes and connecting them via the paths in the host tree, and selecting a random set of edges to remove from the host tree and selecting one of the resulting connected components as a subtree. We will revisit these methods in the long version of this work.

References

1. F. Abu-Khzam, P. Heggernes. Enumerating minimal dominating sets in chordal graphs. Inf. Process. Lett. 116(12): 739-743 (2016).
2. M. I. Andreou, V. G. Papadopoulou, P. G. Spirakis, B. Theodorides and A. Xeros. Generating and radiocoloring families of perfect graphs. *Experimental and Efficient Algorithms*, pp. 302–314, Springer, 2005.
3. J. R. S. Blair and B. W. Peyton. An Introduction to Chordal Graphs and Clique Trees. In *Graph Theory and Sparse Matrix Computations*, IMA Vol. in Math. Appl. 56: 1-27, Springer, 1993.

4. M. Bougeret, N. Bousquet, R. Giroudeau, and R. Watrigant. Parameterized Complexity of the Sparsest k-Subgraph Problem in Chordal Graphs. SOFSEM 2014: 150–161, Springer.
5. P. Buneman. A characterisation of rigid circuit graphs, Disc. Math. 9(3): 205–212, 1974.
6. A. Brandstädt, V. B. Le, and J. Spinrad. *Graph Classes: A Survey.* SIAM Monographs on Discrete Mathematics and Applications (1999).
7. G. A. Dirac. On rigid circuit graphs. Ann. Math. Sem. Univ. Hamburg 25: 71–76, 1961.
8. D. Fulkerson and O. Gross. Incidence matrices and interval graphs. Pac. J. of Math. 15(3): 835–855, 1965.
9. F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. SIAM J. on Comp. 1(2): 180–187, 1972.
10. F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. J. of Comb. Th. B, 16(1): 47–56, 1974.
11. P. Golovach, P. Heggernes, and D. Kratsch. Enumerating minimal connected dominating sets in graphs of bounded chordality. Theor. Comput. Sci. 630: 63–75 (2016)
12. P. Golovach, P. Heggernes, D. Kratsch, and R. Saei. An exact algorithm for Subset Feedback Vertex Set on chordal graphs. J. of Disc. Alg. 26: 7–15 (2014).
13. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Annals of Disc. Math. 57, Elsevier (2004).
14. A. Hajnal and J. Surányi. Über die Ausflösung von Graphen in vollständige Teilgraphen. *Ann. Univ. Sci. Budapest*, pages 113–121, 1958.
15. P. Heggernes. Minimal triangulations of graphs: A survey. Disc. Math. 306(3): 297–317, 2006.
16. D. Loksthanov. Dagstuhl Seminar 14071 "Graph Modification Problems", 2014.
17. G. S. Lueker and K. S. Booth. A linear time algorithm for deciding interval graph isomorphism. JACM 26(2): 183–195, 1979.
18. L. Markenzon, O. Vernet, and L. H. Araujo. Two methods for the generation of chordal graphs. Ann. of Op. Res. 157(1): 47–60, 2008.
19. D. Marx. Parameterized coloring problems on chordal graphs. Theor. Comput. Sci. 351(3): 407–424, 2006.
20. N. Misra, F. Panolan, A. Rai, V. Raman, S. Saurabh. Parameterized Algorithms for Max Colorable Induced Subgraph Problem on Perfect Graphs. WG 2013: 370-381, Springer.
21. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, 2014.
22. S. V. Pemmaraju, S. Penumatcha and R. Raman. Approximating interval coloring and max-coloring in chordal graphs. J. of Exp. Alg. 10: 2–8, 2005.
23. A. S. Rodionov and H. Choo. On Generating Random Network Structures: Trees. International Conference on Computational Science 2003: 879-887, LNCS 2658, Springer.
24. D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. Graph theory and computing 183: 217, 1972.
25. D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. SIAM J. on Comp. 5(2): 266–283, 1976.
26. J. P. Spinrad. *Efficient graph representations.* AMS, Fields Institute Monograph Series 19 (2003).