Linear Algorithms for Chordal Graphs of Bounded Directed Vertex Leafage

Michel Habib[1] Juraj Stacho[1]

LIAFA – CNRS and Université Denis Diderot – Paris VII, Paris, France

Abstract

The *directed vertex leafage* of a chordal graph G is the smallest integer k such that G is the intersection graph of subtrees of a rooted directed tree where each subtree has at most k leaves. In this note, we show how to find in time $O(kn)$ an optimal colouring, a maximum independent set, a maximum clique, and an optimal clique cover of an n-vertex chordal graph G with directed vertex leafage k if a representation of G is given. In particular, this implies that for any path graph G, the four problems can be solved in time $O(n)$ given a path representation of G.

Keywords: path graph, chordal graph, colouring, clique, linear algorithms

1 Introduction

In the following text, a graph is always simple, undirected and loopless. A *hole* is a cycle with at least four vertices. A graph is *chordal* if it contains no induced holes. *A tree model* of a graph G is a collection of subtrees $\{T_v\}_{v\in V(G)}$ of *a host tree* T such that $uv \in E(G)$ if and only if $T_u \cap T_v$ is non-empty. In other words, G is the intersection graph of the subtrees $\{T_v\}_{v\in V(G)}$. It is well-known[4] that a graph is chordal if and only if it has a tree model.

[1] Email: {habib,jstacho}@liafa.jussieu.fr

Let G be a chordal graph. The *leafage* of G is the minimum integer k such that G has a tree model where the host tree T has k leaves (see [8]). The *vertex leafage* of G is the minimum integer k such that G has a tree model where each subtree T_v has at most k leaves. The *directed vertex leafage* of G is the minimum integer k such that G has a tree model where the host tree T is rooted and directed (each edge is directed away from the root) and each subtree T_v has at most k leaves (vertices of outdegree zero). Using a standard argument (see [6]), one can assume that in all the above tree models, the host tree T has at most $|V(G)|$ vertices. In this paper, we only consider such tree models.

A graph is *interval* if it is the intersection graph of intervals of the real line. A graph is *a directed path graph* if it is the intersection graph of directed paths of a rooted directed tree. A graph is *a path graph* if it is the intersection graph of paths of a tree. All these three types of graphs are chordal. In particular, we observe that a graph G interval if and only if G has leafage two, a graph G is a path graph if and only if G has vertex leafage two, and a graph G is a directed path graph if and only if G has directed vertex leafage one.

An implicit representation [11] of an n-vertex graph G is an assignment of labels of size $O(\log(n))$ bits to each vertex of G such that adjacency of any two vertices can be computed only from their labels.

Any graph on n vertices can be represented by its adjacency matrix using $O(n^2)$ bits. On a random-access machine with $\log(n)$ word length [2], any graph with n vertices and m edges can be stored by listing its vertices and edges in space $O(n+m)$. This is best possible for general but also for chordal graphs (for more details see [11]). However, both representations are not implicit. Indeed, any implicit representation clearly requires only $O(n)$ space.

On the other hand, any interval graph can be represented by storing in each vertex v the endpoints l_v, r_v of the interval representing this vertex. The endpoints of the intervals can be taken from the set $\{1 \ldots n\}$. Hence, this representation requires $O(n)$ space, and it is implicit, since two vertices u, v are adjacent if and only if $l_v \leq l_u \leq r_v$ or $l_u \leq l_v \leq r_u$.

Similarly, for any chordal graph G with (directed) vertex leafage k, we can construct a representation in space $O(kn)$ using a tree model of G. In each vertex v of G, we simply store the root and the leaves of the subtree T_v. If k is fixed, this representation is implicit. [3]

[2] Throughout the paper, we shall use this model for complexity analysis.

[3] To be precise, this requires that each vertex x of the host tree T is labeled with a pair of numbers (d_x, f_x) where d_x is the discovery time and f_x is the finish time of the vertex x in some depth-first traversal of T. In each vertex v of G, we store the labels (d_x, f_x) for the root and the leaves x of T_v. Using these labels, the adjacency testing can be done in time $O(k)$.

The paper is structured as follows. In Sections 2 and 3, we explain two auxiliary problems and introduce data structures we shall need later. Then in Section 4, using the results from the previous two sections, we explain how to find an optimal colouring, a maximum clique, a maximum independent set, and an optimal clique cover of any chordal graph with directed vertex leafage k in time $O(kn)$. We conclude the paper by discussing some remarks regarding the complexity of computing leafage and (directed) vertex leafage.

2 Preprocessing I.

Suppose we are given a rooted directed tree T whose vertices are labeled by some preorder traversal of T, and we are also given an ordered list $\mathcal{P}$ of directed paths of T. For each directed path $P \in \mathcal{P}$, let $\mathsf{top}(P)$ and $\mathsf{leaf}(P)$ denote the two extremes of P such that P is a directed path from $\mathsf{top}(P)$ to $\mathsf{leaf}(P)$.

For paths $P, P' \in \mathcal{P}$, we write $P \prec P'$ if and only if either the label of $\mathsf{leaf}(P)$ is strictly smaller than the label of $\mathsf{leaf}(P')$, or the two labels are equal and P appears on the list $\mathcal{P}$ before P'. (Recall that the vertices of T are labeled by a preorder.) Clearly, $\prec$ is a linear ordering of $\mathcal{P}$.

For each vertex x of T, let $\mathsf{Clique}(x)$ denote the paths $P \in \mathcal{P}$ with $x \in P$ listed in the order $\prec$. Similarly, for each edge xy of T, let $\mathsf{Clique}(xy)$ denote the paths $P \in \mathcal{P}$ with $x, y \in P$ listed in the order $\prec$.

For a path $P \in \mathcal{P}$, let $\mathsf{before}(P)$ denote the path that appears just before P in the list $\mathsf{Clique}(\mathsf{top}(P))$. If there is no such path, let $\mathsf{before}(P) = \mathsf{nil}$.

For each edge xy of T, let $\mathsf{first}(xy)$, respectively, $\mathsf{last}(xy)$ denote the path listed first, respectively, the last on the list $\mathsf{Clique}(xy)$.

To compute these values, we use Algorithm 1.

Proposition 2.1 *Algorithm 1 computes correctly the values* $\mathsf{before}(P)$*,* $\mathsf{first}(xy)$ *and* $\mathsf{last}(xy)$ *for all paths* $P \in \mathcal{P}$ *and all edges* xy *of* T *in time* $O(|V(T)|+|\mathcal{P}|)$*.*

Proof. Correctness is proved by showing by induction that (i) after line 6, the value of the variable $L(x)$ is exactly $\mathsf{Clique}(x)$, and (ii) after line 12, the value of the variable $L(zx)$ is exactly $\mathsf{Clique}(zx)$.

In addition, we store the lists $L(x)$ and $L(zx)$ as doubly linked lists. This allows us to perform all operations (lines 6, 8, and 12-14) on these sets in constant time. Secondly, the sets used in line 4 and in the for cycles in lines 7 and 12 are precomputed in time $O(|V(T)| + |\mathcal{P}|)$ by sorting the paths P of $\mathcal{P}$ according to the labels of $\mathsf{leaf}(P)$, respectively, $\mathsf{top}(P)$.

Finally, each path $P \in \mathcal{P}$ is processed only once in lines 7-8 and 11-12 Hence, $O(|V(T)| + |\mathcal{P}|)$ complexity follows. $\square$

```
1.  $S \leftarrow \emptyset$    // *processed vertices*
2.  **while** $S \neq V(T)$ **do**
3.     pick $x \in V(T) \setminus S$ that has all children in $S$
4.     $L(x) \leftarrow$ all paths $P \in \mathcal{P}$ with $\mathsf{leaf}(P) = x$ listed in the order $\prec$.
5.     **for** all children $y$ of $x$ considered in increasing value of the label of $y$ **do**
6.        $L(x) \leftarrow L(x) * L(xy)$    // *concatenate* $L(x)$ *and* $L(xy)$
7.     **for** each $P \in L(x)$ with $\mathsf{top}(P) = x$ **do**
8.        $\mathsf{before}(P) \leftarrow$ vertex before $P$ in $L(x)$
9.     **if** $x$ has a parent $z$ **then**
10.       $L(zx) \leftarrow L(x)$    // *copy a pointer*
11.       **for** each $P \in L(zx)$ with $\mathsf{top}(P) = x$ **do**
12.          remove $P$ from $L(zx)$
13.       $\mathsf{first}(zx) \leftarrow$ the first element of $L(zx)$
14.       $\mathsf{last}(zx) \leftarrow$ the last element of $L(zx)$
15.    add $x$ to $S$
```

Algorithm 1. Computing the values $\mathsf{before}(P)$, $\mathsf{first}(xy)$, and $\mathsf{last}(xy)$.

3 Preprocessing II.

We also consider another auxiliary problem. Suppose that we are given a rooted directed tree T. Initially, each vertex of T is unmarked.

We want to perform a mixed sequence of the following two operations.

- $mark(x)$: marks the vertex x of T.
- $get(x)$: outputs all unmarked vertices in the subtree of T rooted at x.

We want that the procedure $mark(x)$ has constant amortized time complexity, and the complexity of $get(x)$ is $O(1+d)$ where d is the size of the output of $get(x)$. In addition, we allow $O(|V(T)|)$ preprocessing time.

We implement these two procedures as follows. For each vertex x of T, we keep a value $\mathsf{low}(x)$ initially set to x. In addition, we use a *static tree union structure* on T. That is, we maintain a partition of T into connected subtrees which allows us to perform any mixed sequence of following two operations

- $find(x)$: returns the root of the subtree to which x belongs
- $link(x)$: unites the subtree containing x with the one containing its parent

where the initial partition of T puts each vertex x in its own set $\{x\}$.

Gabow and Tarjan[3] give an implementation of the above procedures on random access machine with $log(|V(T)|)$ word size. Their implementation allows executing any above sequence of length M in time $O(|V(T)|+M)$.

The details of our implementation of the procedures *mark*(x) and *get*(x) can be found as Algorithm 2. During the processing of the sequence of these two operations, we maintain the following invariants: (i) each subtree of the partition of T is a directed path, and (ii) for each subtree of the partition with a root z, $\mathsf{low}(z)$ is the (only) leaf of this subtree, and (iii) whenever *compact*(x) or *output*(x) is called, x is a leaf of a subtree of the partition.

```
procedure mark(x)
1. if $x$ is unmarked then
2.     mark $x$
3.     call compact($x$)

procedure compact(x)
1. if $x$ is marked then
2.     if $x$ has exactly one child $y$ then
3.         $\mathsf{low}(find(x)) \leftarrow \mathsf{low}(y)$
4.         call link($y$)
5.     if $x$ has no children and
               $find(x)$ has a parent $z$ then
6.         remove the edge $(z, find(x))$
7.         call compact($z$)

procedure get(x)
1. call output($\mathsf{low}(find(x))$)

procedure output(x)
1. if $x$ is unmarked then
2.     output $x$
3. for each child $y$ of $x$ do
4.     call output($\mathsf{low}(y)$)
```

Algorithm 2. Implementation of the auxiliary problem

Proposition 3.1 *The implementation of mark(x) and get(x) in Algorithm 2 is correct and has the desired amortized complexity as explained above.*

Proof. Follows from proving invariants (i)-(iii). □

4 Algorithms

In this section, using the algorithms and data structures from the previous sections, we prove the following theorem.

Theorem 4.1 *Given a representation of an n-vertex chordal graph G with directed vertex leafage k, one can find in time $O(kn)$ an optimal colouring, a maximum independent set, a maximum clique, and an optimal clique cover of G.*

Let G be a chordal graph with directed vertex leafage k, and let a rooted tree T with a subtree T_v for each $v \in V(G)$ be a representation of G. For vertices x, y of T, let $\mathsf{lca}(x, y)$ denote the *lowest common ancestor* of x, y.

Recall that each subtree T_v is a rooted directed tree and has at most k leaves (vertices of outdegree zero). Let $v_1, \ldots, v_{t(v)}$ be leaves of T_v ordered increasingly by their label. Let P_v^1 be the directed path from the root of T_v to v_1. For $i \geq 2$, let P_v^i be the directed path from $\mathsf{lca}(v_{i-1}, v_i)$ to v_i.

We observe that T_v is the union of paths $\mathcal{P}(v) = \{P_v^1, \ldots, P_v^{t(v)}\}$. We say that the paths in $\mathcal{P}(v)$ are *associated* with each other, and that the vertex v and each path in $\mathcal{P}(v)$ are *corresponding*. Let $\mathcal{P}$ denote the union of $\mathcal{P}(v)$ over all $v \in V(G)$. By the result of [1], we have that $\mathcal{P}$ can be computed in time $O(kn)$.

4.1 Minimum colouring and maximum clique

Our colouring algorithm follows the idea of $O(n)$ time algorithm for the colouring of interval graphs [7]. We process the vertices of G in the reverse of a perfect elimination ordering while colouring the vertices with colours not used in their neighbourhoods. For that we maintain a list L of coloured vertices adjacent to the currently processed vertex v, and a list $Free$ of free colours (that is, the list of coloured vertices that were most recently coloured and are not adjacent to v). To colour v, we first remove a vertex from $Free$ and use its colour to colour v. Since we maintain that no vertex in $Free$ is adjacent to v, this always gives a proper colouring. If $Free$ is empty, the coloured neighbours of v use all colours available. But since they also form a clique, we use a new colour to colour v. The only difficulty is to maintain the lists L and $Free$. For that we use the preprocessing from Section 2.

We now explain more details. Instead of storing L and $Free$ as lists of vertices of G, we store them as doubly linked lists of paths of $\mathcal{P}$ corresponding to those vertices. The algorithm then goes as follows. First, we run Algorithm 1 computing the values $\mathsf{before}(P)$, $\mathsf{first}(xy)$ and $\mathsf{last}(xy)$ for each path $P \in \mathcal{P}$ and each edge xy of T. Then, we process the vertices of T from the root to the leaves while colouring the vertices of G. We maintain the lists L and $Free$ using the precomputed values $\mathsf{before}(P)$, $\mathsf{first}(xy)$ and $\mathsf{last}(xy)$. In addition, for each path $P \in \mathcal{P}$, we use $\mathsf{vertex}(P)$ to denote the vertex of G corresponding to P, and for each coloured vertex v of G, we keep a value $\mathsf{represent}(v)$ pointing to a particular path of $\mathcal{P}$ corresponding to v.

The algorithm is described in detail as Algorithm 3. In the algorithm, we maintain the following invariants: (i) after line 8, the value of L is $\mathsf{Clique}(x)$ (ii) after line 19, the value of L is $\mathsf{Clique}(xy)$, and (iii) after line 11 and after line 18, no two paths of L correspond to the same vertex.

Proposition 4.2 *Algorithm 3 correctly computes an optimal colouring of G in time $O(kn)$.*

```
procedure $colour()$
1.  run Algorithm 1 on $\mathcal{P}$
2.  call $process(root(T), \emptyset, \emptyset)$

procedure $process(x, L, Free)$
1.  for all paths $P \in \mathcal{P}$ with $\mathsf{top}(P) = x$ considered in the order $\prec$ do
2.      insert $P$ into $L$ after $\mathsf{before}(P)$
3.      if $\mathsf{vertex}(P)$ is not coloured then
4.          $\mathsf{represent}(\mathsf{vertex}(P)) \leftarrow P$   // make $P$ a representative of $\mathsf{vertex}(P)$
5.          if $Free$ is not empty then
6.              remove the first element $P_0$ from $Free$
                    and colour $\mathsf{vertex}(P)$ with the colour of $\mathsf{vertex}(P_0)$
7.          else colour $\mathsf{vertex}(P)$ with a new colour
8.  for all paths $P \in \mathcal{P}$ with $\mathsf{top}(P) = x$ do
9.      if $P \neq \mathsf{represent}(\mathsf{vertex}(P))$ then remove $P$ from $L$
10. for all children $y$ of $x$ considered in increasing value of the label of $y$ do
11.     for all paths $P \in \mathcal{P}$ with $\mathsf{top}(P) = x$ and $y \in P$ considered in $\prec$ do
12.         if $P$ is not in $L$ then
13.             insert $P$ into $L$ after $\mathsf{before}(P)$
14.             remove $\mathsf{represent}(\mathsf{vertex}(P))$ from $L$
15.             $\mathsf{represent}(\mathsf{vertex}(P)) \leftarrow P$
16.     remove from $L$ all paths before $\mathsf{first}(xy)$ and all paths after $\mathsf{last}(xy)$
            and add these paths to $Free$
17.     call $process(y, L, Free)$
18.     restore the values of $L$ and $Free$ to those before step 19.
19. restore the values of $L$ and $Free$ and the values of $\mathsf{represent}(\mathsf{vertex}(P))$
            for each $P \in \mathcal{P}$ with $\mathsf{top}(P) = x$ to those before step 1.
```

Algorithm 3. The colouring algorithm.

Proof. The correctness follows from proving invariants (i)-(iii). Regarding the complexity, since L is a doubly linked list, insertions, deletions, concatenations and splitting can be implemented each in constant time. In line 19, we compute the new values of L and $Free$ by splitting L into three parts, and then concatenating two of them to obtain $Free$. To be able to reverse this process, we just need to store in addition a pointer to the first path before $\mathsf{first}(xy)$ in L and a pointer to the first path after $\mathsf{last}(xy)$ in L. Similarly, we can reverse the process from lines 1-8 and 12-15.

Finally, we observe that each path of $\mathcal{P}$ is used only a constant number of times, and T has $O(n)$ vertices. The complexity now follows. □

Now, since G is chordal, the number of colours used in the colouring procedure above yields precisely the size of the largest clique. In fact, if v is the last vertex of G that was coloured with a new colour, the paths in the list L during the processing of v correspond to a largest clique of G.

Proposition 4.3 *Algorithm 3 can be used to find a maximum clique of G.*

4.2 Maximum independent set and minimum clique cover

Our algorithm for maximum independent set follows the standard algorithm for chordal graphs by Gavril [6]. This algorithm processes the vertices of G in a perfect elimination ordering. Each time a vertex v is processed, it is placed in the independent set and all its neighbours are removed from G. Since the neighbours of v necessarily form a clique, this way we construct an independent set of maximum size and also an optimal clique covering of G.

The algorithm of Gavril runs in time $O(n)$ but requires complete information about the neighbourhoods of all vertices. However, it can be seen that we only need to know the neighbourhoods of the vertices which are placed in the independent set. Our $O(kn)$ time algorithm uses this fact and computes these neighbourhoods on demand. For that we use the data structure from Section 3.

We now explain more details. We initialize the algorithm by assigning to each vertex x of T a list $L_{\mathsf{top}}(x)$ consisting of all paths $P \in \mathcal{P}$ with $\mathsf{top}(P) = x$, and a list $L_{\mathsf{leaf}}(x)$ consisting of all paths $P \in \mathcal{P}$ with $\mathsf{leaf}(P) = x$. For each x with empty $L_{\mathsf{leaf}}(x)$, we call $mark(x)$.

Then, we construct a maximum independent set of G by processing the vertices of T from the leaves to the root. Each vertex x of T is processed by

1. choosing a path $P \in L_{\mathsf{top}}(x)$,
2. adding the vertex corresponding to P to the independent set,
3. removing from lists L_{leaf} and L_{top} the path P and all paths associated with P (representing the same vertex), and
4. removing from lists L_{leaf} and L_{top} all paths P' (also removing their associated paths) such that $\mathsf{leaf}(P')$ belongs to the subtree of T rooted at x.

If in step 1 the list $L_{\mathsf{top}}(x)$ is empty, we skip the steps 2-4.

To do this efficiently, we utilize the structure from Section 3. Each time a list $L_{\mathsf{leaf}}(x)$ becomes empty, we immediately call $mark(x)$. To obtain the list of all paths P' in step (iv), we call $get(x)$ and output the lists $L_{\mathsf{leaf}}(z)$ for all vertices z we obtain from $get(x)$.

Finally, note that it can be shown that all paths that we remove while processing each vertex x of T correspond to vertices of G forming a clique. Hence, using the above algorithm we also obtain an optimal clique cover.

Proposition 4.4 *The above algorithm correctly computes the maximum independent set and an optimal clique cover of G in time $O(kn)$.*

Proof. In the algorithm, we maintain that (i) all unmarked vertices of x have non-empty list $L_{\mathsf{leaf}}(x)$. The correctness follows. Also, it can be seen that each path of $\mathcal{P}$ is considered only once. Since there are $O(kn)$ paths in $\mathcal{P}$ and the tree T is of size $O(n)$, the complexity follows from Proposition 3.1. $\square$

5 Conclusion

In the conclusion, we mention some remarks on the complexity of computing leafage, vertex leafage, and directed vertex leafage of chordal graphs.

Since interval graphs and path graphs can be recognized in polynomial time [2,10], deciding that the leafage, respectively, the vertex leafage of a chordal graph is at most two is polynomially solvable. Moreover, by the result of [9], deciding that the leafage is at most three is also polynomial time solvable.

However, for other fixed values of k, the complexity of the two problems is not known. Moreover, the complexity when k is part of the input is also not yet established [12].

We have similar situation with directed vertex leafage. The complexity is open for fixed $k \geq 2$ or when k is part of the input, whereas, for $k = 1$, there exists a polynomial time algorithm [5].

References

[1] Bender, M. A. and M. Farach-Colton, *The LCA problem revisited*, in: *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, 2000, pp. 88–94.

[2] Booth, K. S. and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms.*, J. Comput. Syst. Sci. **13** (1976), pp. 335–379.

[3] Gabow, H. N. and R. E. Tarjan, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. Syst. Sci. **30** (1985), pp. 209–221.

[4] Gavril, F., *The intersection graphs of subtrees in trees are exactly the chordal graphs*, Journal of Combinatorial Theory B **16** (1974), pp. 47–56.

[5] Gavril, F., *A recognition algorithm for the intersection graphs of directed paths in directed trees*, Discrete Mathematics **13** (1975), pp. 237–249.

[6] Golumbic, M. C., "Algorithmic Graph Theory and Perfect Graphs," Academic Press, New York, 1980.

[7] Gupta, U. I., D. T. Lee and J. Y.-T. Leung, *An optimal solution for the channel assignment problem*, IEEE Trans. Computers **C-28** (1979), pp. 807–810.

[8] Lin, I.-J., T. A. McKee and D. B. West, *The leafage of a chordal graph*, Discussiones Mathematicae Graph Theory **18** (1998), pp. 23–48.

[9] Prisner, E., *Representing triangulated graphs in stars*, Abhandlungen aus dem Mathematischen Seminar der Universitt Hamburg **62** (1992), pp. 29–41.

[10] Schäffer, A. A., *A faster algorithm to recognize undirected path graphs*, Discrete Applied Mathematics **43** (1993), pp. 261–295.

[11] Spinrad, J. P., "Efficient Graph Representations," American Mathematical Society, 2003.

[12] West, D. B. (2008), personal communication.