Succinct Data Structures for Chordal Graph with Bounded Leafage or Vertex Leafage

Meng He
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Kaiyu Wu
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

We improve the recent succinct data structure result of Balakrishnan et al. for chordal graphs with bounded vertex leafage (SWAT 2024). A chordal graph is a widely studied graph class which can be characterized as the intersection graph of subtrees of a host tree, denoted as a tree representation of the chordal graph. The vertex leafage and leafage parameters of a chordal graph deal with the existence of a tree representation with a bounded number of leaves in either the subtrees representing the vertices or the host tree itself.

We simplify the lower bound proof of Balakrishnan et al. which applied to only chordal graphs with bounded vertex leafage, and extend it to a lower bound proof for chordal graphs with bounded leafage as well. For both classes of graphs, the information-theoretic lower bound we (re-)obtain for $k = o(n)$ is $(k-1)n \log n - kn \log k - o(kn \log n)$ bits, where the leafage or vertex leafage of the graph is at most $k = o(n)$. We further extend the range of the parameter k to $\Theta(n)$ as well.

Then we give a succinct data structure using $(k-1)n \log(n/k) + o(kn \log n)$ bits to answer `adjacent` queries, which test the adjacency between pairs of vertices, in $O(\frac{\log k}{\log \log n} + 1)$ time compared to the $O(k \log n)$ time of the data structure of Balakrishnan et al. For the `neighborhood` query which lists the neighbours of a given vertex, our query time is $O(\frac{\log n}{\log \log n})$ per neighbour compared to $O(k^2 \log n)$ per neighbour.

We also extend the data structure ideas to obtain a succinct data structure for chordal graphs with bounded leafage k, answering an open question of Balakrishnan et al. Our succinct data structure, which uses $(k-1)n \log(n/k) + o(kn \log n)$ bits, has query time $O(1)$ for the `adjacent` query and $O(1)$ per neighbour for the `neighborhood` query. Using slightly more space (an additional $(1+\varepsilon)n \log n$ bits for any $\varepsilon > 0$) allows `distance` queries, which compute the number of edges in the shortest path between two given vertices, to be answered in $O(1)$ time as well.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Data structures design and analysis

Keywords and phrases Chordal Graph, Leafage, Vertex Leafage, Succinct Data Structure

Digital Object Identifier 10.4230/LIPIcs.WADS.2025.35

Funding This work is supported by NSERC.

1 Introduction

Chordal graphs and related graph classes are one of the most well studied classes of graphs [13, 10, 20, 18, 23, 11, 29, 6]. One of the first instances where chordal graphs are encountered is in the study of Gaussian elimination of sparse matrices [29], which leads to a characterization of chordal graphs based on an elimination ordering. Another characterization is that there are no induced cycles of length four or more [10]. A third characterization, which is more suitable towards data structures, is the characterization as an intersection graph. For each chordal graph G, there exists a host tree T (which is unrooted) and n subtrees $T(v)$ of T (one for each vertex), such that two vertices u and v are adjacent in G if and only if $T(u)$ and $T(v)$ intersect at some node of T [18].

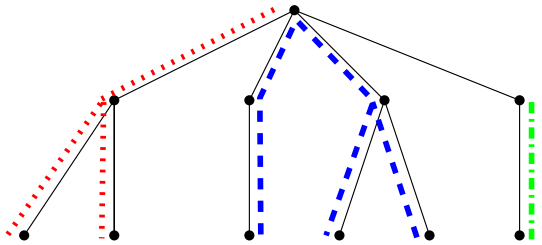

Figure 1 A tree representation with the host tree in black solid lines. The host tree has 6 leaves, so this chordal graph has leafage at most 6. 3 subtrees are depicted, a red dotted subtree, a blue dashed and a green dotted-dashed. The red subtree has 3 leaves, the blue subtree also has 3 leaves and green subtree has 2. So this chordal graph has vertex leafage at most 3.

It is this last characterization of chordal graphs that allows us to (easily) generate many interesting subclasses of chordal graphs, by imposing conditions[1] on either the subtree $T(v)$ or the host tree T. For example, if we impose the condition that the host tree is a path (which also imposes the condition that the subtrees themselves are paths), then the subclass of graphs we generate is the class of *interval graphs* [7]. If we instead only impose that the subtrees themselves are paths, then we generate the class of *path graphs* [17]. If we further impose conditions on these path subtrees, such as selecting a root in the host tree and insisting that each path is a subpath of a root-to-leaf path, then we obtain a RDV (rooted, directed, vertex) graph [24].

If we impose the condition that the host tree has at most k leaves (as the tree is unrooted, a leaf is a degree 1 node), then this parameter k is known as the *leafage* of the chordal graph [23]. We may alternatively impose that only the subtrees $T(v)$ rather than the host tree have at most k leaves each (i.e at most k degree 1 nodes). Then this parameter is known as the *vertex leafage* [11]. A chordal graph with bounded leafage k has a tree representation where the host tree has at most k leaves, and a chordal graph with bounded vertex leafage k has a tree representation where all subtrees have at most k leaves. For $k = 2$, this corresponds to interval graphs and path graphs, respectively, whereas for $k = n$, we obtain all chordal graphs. Thus, the leafage parameter measures how close a chordal graph is to an interval graph, and the vertex leafage parameter measures how close a chordal graph is to a path graph. See Figure 1.

We observe that if the host tree has at most k leaves, then all subtrees would as well. Therefore, chordal graphs with leafage k is a subset of chordal graphs with vertex leafage k. Consequently, any data structure for vertex leafage applies to leafage, and any lower bound for leafage applies to vertex leafage.

An enumeration of chordal graphs and path graphs gives lower bounds for any data structure which is able to compute adjacency, degree and neighbourhood queries. A chordal graph requires at least $n^2/4 - o(n^2)$ bits [25, 31] while both interval graphs and path graphs ([9] plus the above observation) requires at least $n \log n - O(n)$ bits [2], which gives some bounds for the extreme values of the leafage parameter.

In this paper, we study (static) data structures for (unweighted) chordal graphs parametrized by leafage and vertex leafage parameters, with an emphasize on space usage. Our aim is to store a chordal graph parametrized by either the leafage or vertex leafage parameters,

[1] This is not the only way to characterize these subclasses of graphs. For instance, interval graphs can also be characterized by being chordal and having no asteroid triples.

[2] We will use log to denote $\log_2$

using information-theoretic minimal space within an additive lower-order term, while supporting the following queries efficiently: `adjacent`(u, v), which tests the adjacency of the vertices u and v, `neighborhood`(v), which returns a list of the neighbours of the vertex v, and `distance`(u, v), which returns the distance (the number of edges on the shortest path) between u and v. We aim to improve the work of Balakrishnan et al. [3] by eliminating their linear dependence on the vertex leafage parameter in the `adjacent` query, and the quadratic dependence in the `neighborhood` query for chordal graphs with bounded vertex leafage. Furthermore, we investigate an open question of theirs which asks for a simple representation of chordal graphs with bounded leafage that supports faster queries; they only specifically considered bounded vertex leafage, and provided lower and upper bounds accordingly.

1.1 Related Work

The most pertinent work is the recent data structure of Balakrishnan et al. [3] for chordal graphs with bounded vertex leafage. They showed an information-theoretic lower bound of $(k-1)n\log n - kn\log k - O(\log n)$ bits, for parameter range $k = n^{o(1)}$. For the upper bound, they gave a data structure occupying $(k-1)n\log n + o(kn\log n)$ bits which answers `adjacent` queries in $O(k\log n)$ time. For `neighborhood` queries, they require an additional $2n\log n$ bits, with query time $O(k^2\log n \cdot d_v + \log^2 n)$ where d_v is the degree of the query vertex.

There are many succinct data structures for various classes of graphs related to chordal graphs. Munro and Wu [25] gave a succinct data structure for general chordal graphs, occupying $\frac{1}{4}n^2 + o(n^2)$ bits, with query times $O(f(n))$ for `adjacent`, $O(f^2(n))$ for `neighborhood` and $O(nf(n))$ for `distance`, for any $f(n) = \omega(1)$. This was improved by He et al. [22] to remove a factor of $f(n)$ in the time bound for queries, so that `adjacent` has $O(1)$ time, `neighborhood` has $O(f(n))$ time, and `distance` has $O(n)$ time.

The two graph classes with leafage and vertex leafage 2, interval graphs and path graphs, are studied by Acan et al. [1], He et al. [21], Balakrishnan et al. [4] and He et al. [22]. The result of these works is that interval graphs can be represented succinctly using $n\log n + o(n\log n)$ bits which supports `adjacent`, `neighborhood` [1] and `distance` [21] all in optimal time. For path graphs, a succinct $n\log n + o(n\log n)$ bit data structure supports `adjacent` in $O(\log n/\log\log n)$ time and `neighborhood` in the same time per neighbour. To achieve $O(1)$ time, they use $(3+\varepsilon)n\log n + o(n\log n)$ bits, for any constant $\varepsilon > 0$ [22].

Lastly, there are results for other related classes of intersection graphs (of various geometric and combinatorial objects), such as permutation graphs (line segments between two parallel lines) [30], graphs with small tree-width (subgraphs of chordal graphs with small maximal clique) [14], d-boxicity graphs (higher dimensional analog of interval graphs) [2].

1.2 Our Results

We give lower bounds and data structures for chordal graphs with bounded vertex leafage or bounded leafage. Our results for the former class of graphs extends the range of parameter values applicable in the lower bounds and improves the query times in the data structures, compared to the work of Balakrishnan et al. [3]. Our results for the later class of graphs answers an open question posed by them.

Lower Bounds. With respect to lower bounds for chordal graphs with bounded vertex leafage k, Balakrishnan et al. showed an information-theoretic lower bound of $(k-1)n\log n - kn\log k - O(n\log n)$ bits, applying to the parameter range $k = n^{o(1)}$. We extend the range

Table 1 Comparison of our data structure results for chordal graphs with bounded vertex leafage k with the results of Balakrishnan et al. [3]. The 4th column (*) is the extra space needed to support the `neighborhood` query. Time for `neighborhood` query is per neighbour returned.

Source	Space for Adjacency (bits)	Adjacency	*	Neighbour
[3]	$(k-1)n\log n + o(kn\log n)$	$O(k\log n)$	$2n\log n$	$O(k^2\log n)$
Thm 11	$(k-1)n\log(n/k) + o(kn\log n)$	$O(\frac{\log k}{\log\log n}+1)$	$kn\log n$	$O(\frac{\log n}{\log\log n})$
Thm 12	$(k-1)n\log n + o(kn\log n)$	$O(\frac{\log n}{\log\log n})$	$n\log n$	$O(\frac{\log n}{\log\log n})$

to $k = o(n)$ with the same lower bound, by using a simpler host tree, and using a simpler full colouring scheme rather than a partial colouring scheme. This further allows us to extend our construction to give a lower bound for $k = \Theta(n)$ as well, covering all values of k. In this range, the lower bound we obtain have the form $f(c)n^2 + o(n^2)$, where $k = cn$.

For chordal graphs with bounded leafage, we prove that they have the same lower bound as chordal graphs with bounded vertex leafage, namely $(k-1)n\log n - kn\log k - o(kn\log n)$ bits, when $k = o(n)$. Our construction also allows us to extend the parameter k to $k = \Theta(n)$. Again, the lower bound we obtain have the form $f(c)n^2 + o(n^2)$, where $k = cn$ (for a different function f). This is the first information-theoretic lower bound for this class of graphs.

Data Structures for Bounded Vertex Leafage. We give a succinct data structure occupying $(k-1)n\log(n/k) + o(kn\log n)$ bits compared to that of $(k-1)n\log n + o(kn\log n)$ bits of Balakrishnan et al. [3]. Thus our data structure is succinct for $k = o(n)$ rather than just $k = n^{o(1)}$. Furthermore, we achieve $O(\log k/\log\log n)$ query time for `adjacent` query compared to $O(k\log n)$ time for their data structure. For the `neighborhood` query, we use $O(\log n/\log\log n)$ time per neighbour rather than $O(k^2\log n)$ per neighbour. This is around a factor of k faster for `adjacent` and a factor of k^2 for `neighborhood`, which is significant, especially if k is large. A comparison can be seen in Table 1.

To do this, rather than breaking the subtree $T(v)$ into a set of paths, and storing them within a path graph data structure, we simply store the preorder numbers of the leaves of the tree within a dictionary. This allows us to both compress the space and forgo data structures which is needed to identify paths that come from the same subtree, which saves a factor of k in the time complexities. Next, by storing the leaves of the subtree together, we are able to identify a neighbour u of a vertex v using a single leaf of $T(u)$, rather than possibly returning every leaf of $T(u)$, saving the second factor of k in the `neighborhood` query.

Data Structures for Bounded Leafage. Using the additional structure that the host tree has at most k leaves, we construct the first succinct data structure for chordal graphs with bounded leafage k such that $k = o(n)$, using $(k-1)n\log(n/k) + o(kn\log n)$ bits of space. Our data structure supports `adjacent` in $O(1)$ time and `neighborhood` in $O(1)$ time per neighbour. Using an additional $(1+\varepsilon)n\log n$ bits (for any constant $\varepsilon > 0$)), we also support `distance` in $O(1)$ time. We do this by breaking the host tree into $k-1$ paths, and storing the leaf of the subtree $T(v)$ on each path (if one exists). This allows us to identify leaves of $T(v)$ by this path number, leading to more efficient accesses.

All our results are obtained in the word RAM model with $\Theta(\log n)$ bit words.

2 Preliminaries

In this section, we introduce the existing data structures that our solution use as building blocks. As we will be discussing both graphs and trees at the same time, we will use the term *vertex* to refer vertices in the graph and *node* for trees.

Sets of Integers. We begin by considering various methods of storing sets of non-negative integers (viewed in sorted order).

A key succinct data structure is the bitvector: a length n array of bits. A bitvector B supports three operations. **1)** $\texttt{access}(B, i)$ or simply $B[i]$: return the i-th bit in the bitvector. **2)** $\texttt{rank}_1(B, i)$: return the number of 1 bits in the prefix $B[1, i]$. Symmetrically for $\texttt{rank}_0$. **3)** $\texttt{select}_1(B, i)$: return the index of the i-th 1 bit in B. Symmetrically for $\texttt{select}_0$. We note that a bitvector can be viewed as storing the set of integers corresponding to the index of the 1 bits. Thus, the $\texttt{select}(i)$ for set of integers will return the i-th smallest integer. The bitvector we will use is the result of Clark and Munro:

▶ **Lemma 1** ([12]). *A bit vector of length n can be succinctly represented using $n + o(n)$ bits to support* $\texttt{rank}, \texttt{select}$ *and* $\texttt{access}$ *in $O(1)$ time.*

For a sparse set of integers, we will use the dictionary result of Raman et al. [28].

▶ **Lemma 2.** *A subset of n integers from $[0, \ldots, m-1]$ can be represented using $\lceil \log \binom{m}{n} \rceil + o(n) + O(\log \log m)$ bits, to support* $\texttt{select}$ *queries in $O(1)$ time.*

The predecessor and successor queries on a set S of integers are: **1)** $\texttt{pred}(k)$: which returns the largest element of S no greater than k, i.e. $\max\{s \leq k \mid s \in S\}$, and symmetrically, **2)** $\texttt{succ}(k)$: which returns the smallest element of S no less than than k. When the size of the integers are bounded (and allowing duplicates), we may employ the folklore solution, which stores the termwise differences of the sequence as unary in a bitvector.

▶ **Lemma 3.** *Let A be a multi-set of N non-negative integers bounded by M. Then A can be stored in $N + M + o(N + M)$ bits answer* $\texttt{select}, \texttt{pred}$*, and* $\texttt{succ}$ *queries in $O(1)$ time.*

Ordinal Trees. As we will be studying chordal graphs, using primarily the characterization that it is the intersection graph of subtrees in a tree, we will need a data structure storing said tree. Succinct ordinal tree data structures have a long line of study, with new queries being added as recently as He et al. [21]

▶ **Lemma 4** ([21]). *An ordinal tree can be represented succinctly in $2n + o(n)$ bits to support a wide variety of operations in $O(1)$ time. These operations include:* ***1)*** $\texttt{node\_rank}_{\texttt{PRE}}(x)$*, which returns the index in the preorder traversal that we encounter the node x,* ***2)*** $\texttt{LCA}(x, y)$*, which returns the lowest common ancestor of nodes x and y,* ***3)*** $\texttt{parent}(x)$*, which returns the parent of a node x and* ***4)*** $\texttt{child}(x, i)$*, which returns the i-th child of a node x.*

We will refer to $\texttt{node\_rank}_{\texttt{PRE}}$ of a node as its *preorder number*, and use it to identify nodes. Thus, when we refer to a node $x \in T$, x is treated as an integer, and we may write $x+1$ for the next node encountered in the preorder traversal of T. One application of ordinal trees is to range minimum/maximum queries, through a Cartesian tree. For an array $A[1, n]$ of integers, a *range minimum query* or RMQ takes two indices i, j with $i \leq j$ and returns the index of the minimum (or symmetrically, maximum) value in the subarray $A[i, j]$. We use the following result:

▶ **Lemma 5** ([15]). *There is a data structure for the RMQ problem that uses $2n + o(n)$ bits of space and answers queries in $O(1)$ query time.*

Orthogonal Range Searching. In the 2-dimensional *orthogonal range reporting* query, we are asked to preprocess a set P of n 2-dimensional points, so that given an axis aligned query rectangle $R = [x_1, x_2] \times [y_1, y_2]$, return the list of points lying in the rectangle, i.e $P \cap R$. Bose et al. [8] designed a succinct data structure for this problem when the points have integral coordinates in the $n \times n$ grid.

We will need the following two variants of this query, which deals with many points sharing the same y-coordinate. First is the *orthogonal range distinctness* query: Given an axis-aligned rectangle R, return the set of distinct y-coordinates $y^1, \ldots, y^p$ such that there exists a point $(-, y^i)$ that lies in the rectangle R. Second is the *orthogonal range distinct maximum/minimum* query: Given an axis-aligned rectangle R, and for each distinct y-coordinate in the set of points $P \cap R$, return the point (x^i, y^i) with the maximum/minimum x-coordinate among the points in the rectangle with y-coordinate y^i.

We may solve these two queries by a slight modification of the data structure of Bose et al. An inspection of their orthogonal range reporting query algorithm shows the following. Their algorithm identifies the set of y-coordinates each of which has a point in the given query rectangle. For each such y-coordinate identified, the set of points with that y-coordinate is represented in a bitvector, and the query rectangle defines a subrange of that bitvector. The algorithm traverses the bitvector by using `select` for each of the 1 bits, and then recovers the coordinates of the points represented. Thus, we may `select` for the first or the last 1 in the subrange of the bitvector defined by the query rectangle, which returns the minimum or maximum points.

▶ **Lemma 6.** *Let P be a set of n points in an $n \times n$ grid. P can be represented using $n \log n + o(n \log n)$ bits to support orthogonal range reporting in $O(k \log n / \log \log n)$ time, where k is the size of the output. It can further support orthogonal range distinctness, and orthogonal range distinct minimum/maximum queries in $O(k' \log n / \log \log n)$ time, where k' is the number of distinct y-coordinates among the k points in the rectangle.*

3 Lower Bound for Bounded Leafage and Vertex Leafage

In this section, we will re-derive the lower bound result of Balakrishnan et al. [3] for graphs with bounded vertex leafage k. We will then generalize the result for a larger set of values for k. We will also derive lower bound results for graphs with bounded leafage k, which was not considered by Balakrishnan et al.

Since chordal graphs with bounded leafage are also graphs with bounded vertex leafage, a lower bound for chordal graphs with bounded leafage also applies to chordal graphs with bounded vertex leafage. We will still give different constructions for these two classes of graphs, to emphasize how they can be extended when $k = \Theta(n)$.

3.1 Bounded Vertex Leafage

As in the lower bounds shown in Munro and Wu [25], Wormald [31] and Bender et al. [5], we will mainly look at *split graphs*. A split graph (similar to a bipartite graph) is a graph where the vertices are separated into two groups V_1 and V_2. V_1 is a complete graph, and V_2 is an independent set. Any edge between a vertex $v_1 \in V_1$ and $v_2 \in V_2$ is allowed. It is easy to see that any split graph is a chordal graph by constructing a tree representation as follows: the host tree (a star) T will consist of 1 node R as the root and $|V_2|$ children of the root as leaves, one for each vertex of V_2. Let $x_1, \ldots, x_{|V_2|}$ denote the leaves corresponding to the vertices $v_1, \ldots, v_{|V_2|} \in V_2$. For each vertex v_i of V_2, the subtree $T(v_i)$ that corresponds to it is the

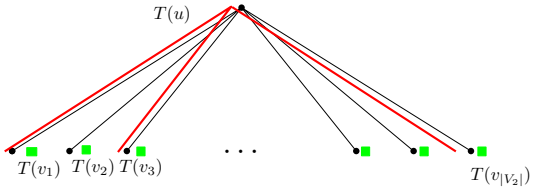

Figure 2 Tree representation of a split graph. For each vertex $v_i \in V_2$ its subtree $T(v_i)$ is a single node at one of the leaves (depicted by the green squares). A vertex $u \in V_1$ has its subtree $T(u)$ consisting of the root plus any nodes representing its neighbours in V_2. Here u would be adjacent to v_1, v_3 and $v_{|V_2|}$.

singular node x_i. For each vertex u of V_1, the subtree $T(u)$ that corresponds to it is the root of the tree r plus the nodes of $|V_2|$ adjacent to u (i.e. $T(u) = \{r\} \cup \{x_i : (u, v_i) \in E, v_i \in V_2\}$). See Figure 2. In this fashion, the subtrees corresponding to vertices of V_1 always intersect each other at the root, so the vertices of V_1 forms a clique as required. The subtrees corresponding to vertices of V_2 never intersect, so V_2 forms an independent set as required, and every edge that exist between V_1 and V_2 exists since the subtree of $u \in V_1$ extends to the node that represents $v_i \in V_2$.

We will now give an information-theoretic lower bound for chordal graphs with bounded leafage k by counting a subclass of these graphs.

▶ **Theorem 7.** *Any data structure supporting adjacency queries on chordal graphs on n vertices with bounded vertex leafage k such that $k = o(n)$ requires at least $(k-1)n \log n - kn \log k - o(nk \log n)$ bits.*

Furthermore, if $k = cn$ for a fixed constant $0 < c \leq 1/4$ then it requires at least $\widehat{f}(c)n^2 - o(n^2)$ bits where $\widehat{f}(c)$ is the maximum of the function

$$f(l,c) = \begin{cases} (1-l)l & \text{if } c \leq l \leq 2c \\ (1-l)(l\log(l/(l-c)) - c\log(c/(l-c))) & \text{if } 2c \leq l \leq 1 \end{cases}$$

in the range $c \leq l \leq 1$, and this maximum is always achieved in the range $2c \leq l \leq 1/2$. If $c \geq 1/4$, it requires $\frac{1}{4}n^2 - o(n^2)$ bits.

Proof. First consider the case where $k = o(n)$.

Let $|V_2| = o(n)$, $|V_2| = \omega(k)$ and let $|V_1| = n - |V_2| = n - o(n)$ be the sizes of the independents set and clique respectively for our split graph. Furthermore, assume that $|V_2|$ is large enough that we have $\log |V_2| = \log n - o(\log n)$ (i.e. $|V_2| = \Omega(n/\texttt{polylog}(n))$). We also have the estimate $\log \binom{|V_2|}{k} \approx k \log |V_2| - k \log k$. This arises from Stirling's approximation and that $k = o(|V_2|)$:

$$\begin{aligned} &\log \binom{|V_2|}{k} \\ &= |V_2| \log |V_2| - k \log k - (|V_2| - k) \log(|V_2| - k) + \frac{1}{2}\left(\log |V_2| - \log k - \log(|V_2| - k)\right) + O(1) \\ &= k \log |V_2| - k \log k + k \left(\frac{|V_2| - k}{k}\right) \log\left(1 + \frac{k}{|V_2| - k}\right) - \frac{1}{2} \log k + O(1) \\ &= k \log |V_2| - k \log k + \Theta(k) \end{aligned}$$

The last step uses $\left(\frac{|V_2|-k}{k}\right) \log\left(1 + \frac{k}{|V_2|-k}\right) = \Theta(1)$ by the limit definition of e: $e = \lim_{x \to \infty}(1 + 1/x)^x$.

For each vertex u of V_1 we can select k vertices from $|V_2|$ to be adjacent to. So all subtrees in the tree representation constructed above has at most k leaves, and thus the graph obtained has bounded vertex leafage k. The number of graphs we can generate is $\binom{|V_2|}{k}^{|V_1|}$ since each vertex of V_1's neighbourhood could be any size k subset of V_2. Note that if we treat this construction as a labelled graph, then all graphs we construct are distinct, as different sets of vertices of $|V_2|$ leads to a different neighbourhood and hence a different graph. Moving from labelled graphs to unlabelled graphs, we generate each unlabelled graph at most $n!$ times due to the number of potential isomorphisms of the graph. Thus the lower bound on the space needed for k vertex leafage graphs is at least

$$\begin{aligned}
&\log\left(\binom{|V_2|}{k}^{|V_1|}\right) - n\log n \\
&= k|V_1|\log(|V_2|) - k|V_1|\log k - n\log n + O(k|V_1|) \\
&= (k-1)n\log n - kn\log k - o(kn\log n)
\end{aligned}$$

Now consider the case in which $k = cn$ for some constant $c \leq 1/4$. [3] Let l be variable such that $c \leq l \leq 1$. Set $|V_1| = (1-l)n$ and $|V_2| = ln$. Applying Stirling's approximation, the number of graphs we generate is

$$\begin{aligned}
&\log\left(\binom{|V_2|}{k}^{|V_1|}\right) - n\log n \\
&= |V_1|\log\left(\binom{ln}{cn}\right) - n\log n \\
&= n(1-l)(ln\log(ln) - cn\log(cn) - (l-c)n\log((l-c)n) - O(\log n)) - n\log n \\
&= n^2(1-l)(l\log l - c\log c - (l-c)\log(l-c)) - o(n^2) \\
&= n^2(1-l)(l\log(l/(l-c)) - c\log(c/(l-c))) - o(n^2)
\end{aligned}$$

assuming that $2c \leq l$. In the case that $c \leq l \leq 2c$, then we do not choose k elements from V_2 but rather choose $ln/2$ elements since that maximizes the binomial coefficient. Doing so gives

$$\begin{aligned}
&\log\left(\binom{|V_2|}{|V_2|/2}^{|V_1|}\right) - n\log n \\
&= n(1-l)(ln) - o(n^2) \\
&= n^2(1-l)l - o(n^2)
\end{aligned}$$

Thus the lower bound is $\widehat{f}(c)n^2 - o(n^2)$ where $\widehat{f}(c)$ given a fixed $c \leq 1/4$ is the maximum of the function $f(l,c)$ on the range $c \leq l \leq 1$ defined by

$$f(l,c) = \begin{cases} (1-l)l & \text{if } c \leq l \leq 2c \\ (1-l)(l\log(l/(l-c)) - c\log(c/(l-c))) & \text{if } 2c \leq l \leq 1 \end{cases}$$

$f(l,c)$ is clearly increasing on the first interval, positive when $l = 2c$ and zero when $l = 1$. Taking the derivative, we see that $\widehat{f}(l,c)$ is obtained at some zero of the function:

$$d/dl(1-l)(l\log(l/(l-c)) - c\log(c/(l-c))) = (1-2l)\log(l/(l-c)) + c\log(c/(l-c))$$

[3] We will show that for $c = 1/4$ the lower bound we obtain is $\frac{1}{4}n^2 - o(n^2)$ bits which matches the lower bound for all chordal graphs. Thus, this would be the lower bound for all $k \geq n/4$ as well.

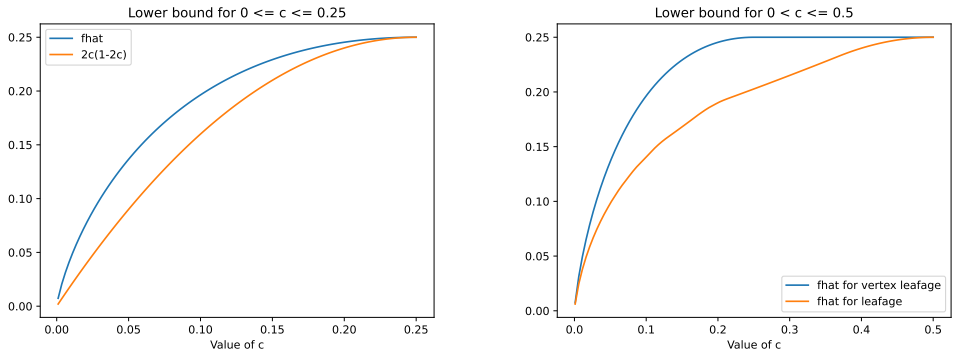

Figure 3 Left: A graph showing the value of $\widehat{f}$ for range of values of c. Right: Comparing the lower bound values for subclasses of chordal graphs (bounded vertex leafage vs bounded leafage).

Since $l \geq 2c$ in the second range, we see that $\log(l/(l-c)) > 0$ and $c\log(c/(l-c)) < 0$. Thus, in order for this sum to evaluate to 0, $1 - 2l \geq 0$, so that $l \leq 1/2$. When $c = 1/4$, we see that $l = 1/2$ is indeed a solution, which evaluates $\widehat{f}(1/4) = (1 - 1/2)(1/2) = 1/4$ matching the original chordal graph lower bound. ◀

For a graph of the value of $\widehat{f}$ (compared to the value of $f(c, 2c)$), see Figure 3.

3.2 Bounded Leafage

We now consider chordal graphs with bounded leafage k (i.e. the host tree has at most k leaves), by subdividing the edges of tree representation of the split graph with k leaves.

▶ **Theorem 8.** *Any data structure supporting adjacency queries on chordal graphs on n vertices with bounded (vertex) leafage k such that $k = o(n)$ requires at least $(k-1)n\log n - kn\log k - o(kn\log n)$ bits.*

If $k = cn$ for some constant fixed constant $0 < c \leq 1/2$, then it requires at least $\widehat{f}n^2 - o(n^2)$ bits, where $\widehat{f}$ is the maximum of the function:

$$f(c,l) = n^2 c(1-l)\left(\log\left(\lfloor l/c\rfloor + 1\right) + [l/c]\log\left(1 + \frac{1}{1+\lfloor l/c\rfloor}\right)\right)$$

Here $[x]$ denotes the fractional part of x.

Proof. We define the host tree T to be an (extended) star, with central node v_0 of degree k together with k paths of length $n/(k\log n)$, between v_0 and a leaf. We set the subtrees representing $n/\log n$ vertices of the chordal graph (which we will call *static* vertices) to be distinct singleton node in the paths, excluding v_0. For the remainder of $n - n/\log n$ vertices (which we will call *variable* vertices), their subtrees will consist of the root v_0 plus a (potentially empty) prefix of each of the paths (i.e. it is the union of k such paths). Thus each variable vertex that has a different subtree than another variable vertex will have a different set of static vertices as neighbours. See Figure 4.

The number of different possible variable vertices is $(n/(k\log n))^k$ since for each of the k paths, we have $n/(k\log n)$ different lengths. Since each of the $n - n/\log n$ variable vertices can be one of these, the total number of ways we can choose our variable vertices is $(n/(k\log n))^{k(n-n/\log n)}$. The total number of graphs we can generate is thus $(n/k\log n)^{k(n-n/\log n)}/n!$ where each graph may be generated at most $n!$ times. Thus our information theoretic lower bound is:

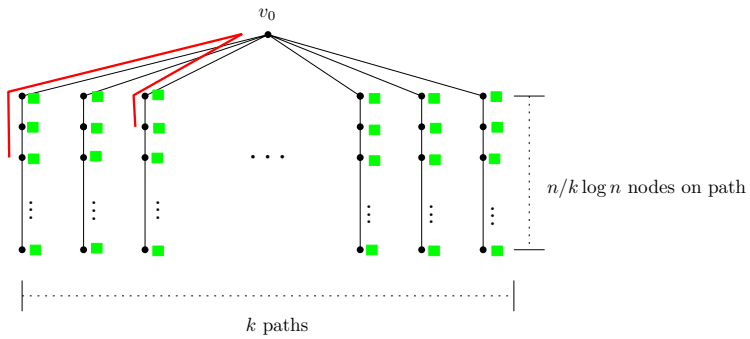

Figure 4 The host tree is a star, with k paths, each of length $n/k \log n$. Green squares represent the subtrees for vertices in V_1. A possible subtree for a vertex in $V \setminus V_1$ in red bolded lines.

$$\log\left((n/k\log n)^{k(n-n/\log n)}/n!\right) = kn(1-1/\log n)(\log n - \log k - \log\log n) - n\log n$$
$$= (k-1)n\log n - kn\log k - o(kn\log n)$$

This applies for $k \leq n/\log n$. To extend to $k = o(n)$ we may set $f(n) = o(n/k)$ and choose the number of static vertices to be $\frac{n}{kf(n)} = o(n)$ and the number of variable vertices to be $n - n/kf = n - o(n)$ without changing the above bound.

In the case that $k = cn$, we set the size of T to be $|T| = ln$ with $1 \geq l \geq c$. We set the length of each of the paths to be the same. That is, there are $\lceil l/c \rceil \times cn$ paths of length $\lfloor l/c \rfloor + 1$ and $(1 - \lceil l/c \rceil) \times cn$ paths of length $\lfloor l/c \rfloor$ (recall that we use $\lceil x \rceil = x - \lfloor x \rfloor$ to denote the fractional part of x). Enumerating as above, we obtain a lower bound that is:

$$\log\left((\lfloor l/c \rfloor + 2)^{\lceil l/c \rceil \times cn \times (1-l)n} (\lfloor l/c \rfloor + 1)^{(1-\lceil l/c \rceil) \times cn \times (1-l)n}\right)$$
$$= n^2 c(1-l)\left(\log(\lfloor l/c \rfloor + 1) + \lceil l/c \rceil \log\left(1 + \frac{1}{\lfloor l/c \rfloor + 1}\right)\right)$$

Finally, we choose l (depending on c) as to maximize this term. ◀

For a diagram of the value of this lower bound, compared to that of the lower bound for bounded vertex leafage chordal graphs in the previous subsection, see Figure 3.

4 Data Structures for Chordal Graphs with bounded Vertex Leafage

In this section we will give data structures for chordal graphs with bounded vertex leafage k. We will aim for $(k-1)n\log n - (k-1)n\log k + o(kn\log n)$ bits of space but as in Balakrishnan et al. [3], allow some extra space to support neighbourhood more efficiently. We note that this differs from the lower bound (outside of lower order terms) only from the $(k-1)n\log k$ versus $kn\log k$ terms (i.e. a difference of $n\log k$ bits). We note that if $k = n^{o(1)}$, $\log k = o(\log n)$ so this is a lower order term and can be ignored. Otherwise, if $\log k = \Theta(\log n)$, then the difference $n\log k$ is a lower order term: $n\log k = o(kn\log k)$. Thus in both instances, we may ignore this $n\log k$ difference as it is a lower order term. Hence such a data structure would be succinct for all $k = o(n)$. We give two data structures, where the second uses slightly more space and more time for the `adjacent` query, but uses less space for the `neighborhood` query.

We will begin with a tree representation of G, T, such that each $T(v)$ has at most k leaves. Root the tree T arbitrarily. For a subtree $T(v)$, denote the unique node a_v with the smallest depth as the *apex* of $T(v)$ (or of v). Furthermore, using the following lemma, we may assume that every node of T is the apex of some subtree $T(v)$.

▶ **Lemma 9.** *Let G be a chordal graph with bounded vertex leafage k. Then there is a tree representation T such that the subtree $T(v)$ of any vertex v of G has at most k leaves and that every node of T is the apex of some vertex. Furthermore, if G has bounded leafage k, then there exists a tree representation T with at most k leaves such that every node of T is the apex of some vertex.*

Proof. We will convert any tree representation of G into one with the property that all nodes of T is the apex of some vertex.

Suppose we have a tree representation such that some node $x \in T$ is not the apex of any vertex. Then any subtrees $T(v)$ containing x must also contain the parent of x. Merge x into its parent (i.e. the children of x become the children of the parent of x). Any two subtrees $T(u)$ and $T(v)$ that intersect at x also intersect in the parent of x, so no new intersections are created nor destroyed, maintaining a valid tree representation of G. Merging the node x into its parent does not increase the number of leaves of T, nor the number of leaves of any subtree $T(v)$, and so is a valid tree representation with bounded leafage or vertex leafage k.

Repeating this process to eliminate all such nodes x, we obtain a tree representation with the property that every node is the apex of some subtree $T(v)$. ◀

Since each node of T is the apex of some vertex, there are at most n nodes in T. Next, we make the following modification to the tree representation to ensure that the apex of each vertex a_v has at least two children in $T(v)$. This allows us to compute a_v via `LCA` of the leftmost and rightmost leaves of $T(v)$. We achieve this goal by adding two new children to each node a of T as the right most children of a. For each vertex v, whose apex a_v has 1 child in $T(v)$, we expand $T(v)$ to one of the newly added child of a_v in T. For each vertex v, whose apex a_v has 0 children in $T(v)$, we expand $T(v)$ to both newly added children of a_v in $T(v)$. This adds $2n$ new nodes to T so that $|T| = O(n)$ and retains the property that every *internal* node is the apex of some subtree $T(v)$ (leaves of T may not any more).

We first give a succinct data structure which allows us to recover the graph. We sort the vertices v by the preorder number of its leftmost leaf, and label the vertices $1, \dots n$. Let $L = 10^{N_1} \cdots 10^{N_{|T|}}$ where N_i is the number of vertices whose leftmost leaf is the tree node i. Then, $\texttt{rank}_1(L, \texttt{select}_0(L, v))$ gives the preorder number of the leftmost leaf of v.

For each vertex v, we will need to compute predecessor and successor queries on the preorder numbers of the leaves of $T(v)$. For each vertex v, let $Leaf(v, i)$ be the i-th leaf of $T(v)$ in preorder. We store the set of at most $k-1$ integers, $\{Leaf(v,i) \mid i \geq 2\}$ (i.e. excluding the leftmost leaf) using Lemma 2, in $(k-1)\log n - (k-1)\log k + o(k + \log n)$ bits. Using `select`, we may retrieve the i-th leaf in $O(1)$ time. Thus for simplicity, we may view $Leaf(v)$ as an array. Using binary search, we can support `pred` and `succ` queries on this array in $O(\log k)$ time.

We may speed this up slightly by storing a subset of the array to obtain an approximate predecessor. As this subset is sparse, we will use a *fusion tree* [16, 27].

▶ **Lemma 10** ([16, 27])**.** *A set of n elements can be stored in $O(n \log n)$ bits to support* `select`, `pred` *and* `succ` *queries in $O(\log_w(n)) = O(\log n / \log w)$ time, where w is the word size in the word RAM model.*

We store $f(k, n)$ evenly spaced elements of the array $Leaf(v)$ in a fusion tree (where $f(k,n) = o(k)$), using $O(nf(k,n)\log n) = o(nk \log n)$ space. To compute a predecessor, we first compute an approximate predecessor using the fusion tree in $O(\log(f(k,n))/\log\log n + 1)$ time. This gives a predecessor which may be too small. We linearly scan the segment defined

by this entry and the next sampled entry to find the true predecessor, using $O(k/f(k,n))$ time. Setting $f(k,n) = k/\log\log k$, we can find the predecessor using $O(\log k/\log\log n + 1)$ time.

This is enough for us to reconstruct the graph, as for each vertex v, the k leaves of $T(v)$ can be computed: the left-most using L, and the other $k-1$ leaves by scanning $Leaf(v)$. This defines the tree since the apex a_v is the lowest common ancestor of the first and last leaves of v (as we ensured that it has degree ≥ 2 in $T(v)$).

We concatenate the arrays $Leaf(v)$, padding ones that are smaller (i.e. when the subtree $T(v)$ has fewer than k leaves) so that they are all the same size. Thus we may access each array in $O(1)$ time, without spending extra overhead for pointers to each individual array. We will do this for all other similar structures (such as the fusion trees) to avoid the overhead from pointers. The total space used so far is the bitvector L, the n arrays $Leaf(v)$, n fusion trees, and the succinct tree storing T. Thus the total space needed so far is $(k-1)n\log(n/k) + o(kn\log n)$ bits.

4.1 Supporting `adjacent` Query

We first find the leftmost and rightmost leaves of $T(v)$. The leftmost leaf can be computed using L as above. The rightmost is $\texttt{pred}(Leaf(v), |T|)$. We then compute the apex a_v using `LCA` on the host tree T. We compare the apex of the two subtrees a_u and a_v for an ancestor-descendant relationship (i.e. check if $\texttt{LCA}(a_u, a_v) \in \{a_u, a_v\}$). If they are not in such a relationship, then $T(u)$ and $T(v)$ are disjoint, and u and v are not adjacent. When a_u and a_v is in an ancestor-descendant relationship, we may without loss of generality, assume that a_u is an ancestor of a_v.

We claim that $T(u)$ intersects $T(v)$ if and only if some leaf of $T(u)$ is a descendant of a_v. For one direction, if some leaf of $T(u)$ is a descendant of $T(v)$, then the path from u to this leaf contains a_v. Since this path is contained in $T(u)$, $a_v \in T(u) \cap T(v)$. Conversely, suppose that $T(u)$ intersects $T(v)$ at some node x. Then the path from x to a_u again contains a_v, so x is a descendant of a_v. Any leaf descendant of x in $T(u)$ is a descendant of a_v.

To check this condition (some leaf of $T(u)$ is a descendant of a_v), we first compute the range of preorder numbers of the subtree rooted at a_v. The smallest such preorder number is a_v, and the largest is the last child of a_v. Let $[l,r]$ be this range of preorder numbers. We wish to check if $[l,r]$ contains any leaf nodes of $T(u)$. To do so, we look at the predecessors of l and r in the sequence of leaf nodes of $T(u)$: $Leaf(u,l)$ and $Leaf(u,r)$. We observe that either l or r are leaf nodes (so $Leaf(u,l) = l$ or $Leaf(u,r) = r$) or a leaf in the range (l,r) would cause $Leaf(u,l) \neq Leaf(u,r)$. Thus computing whether a leaf of $T(u)$ is a descendant of v can be done using a constant number of bitvector, succinct tree, and predecessor queries, which takes $O(\log k/\log\log n + 1)$ time.

4.2 Supporting `neighborhood` Queries

We now consider efficiently computing the neighbourhood of a vertex v. We will break the neighbourhood into two cases, those vertices u such that a_u is a descendant of (or equal to) a_v, and those vertices u such that a_u is an ancestor of a_v.

4.2.1 Case 1: vertices u with a_u a descendant of a_v

For any case 1 neighbours u, $a_u \in T(v)$. We will traverse $T(v)$, and list the vertices u whose apex is the node of T that we currently traverse. To traverse $T(v)$, we may simulate a post-order traversal of $T(v)$, by starting at the leftmost leaf of $T(v)$ and using `parent` and `LCA`. This takes $O(|T(v)|)$ time.

To report the vertices whose apex is a given node of T, we define the operation `apex_list` which takes a node x of T and returns all vertices u such that $a_u = x$.

To support `apex_list`, we store a bitvector $A = 1^{M_1}0 \dots 1^{M_{|T|}}0$ where M_i is this number of vertices with apex at node with preorder number i. Then, $\texttt{rank}_1(A, \texttt{select}_0(A, x)) - \texttt{rank}_1(A, \texttt{select}_0(A, x-1))$ is the number of vertices with apex x (using the convention that the root of the tree T has preorder number 1). We also store an array AL of the vertices in order by their apexes. Then $\texttt{apex\_list}(x)$ computes the range that stores the appropriate vertices and returns them from AL, that is,

$$AL[\texttt{select}_1(A, x), \texttt{select}_1(A, x+1) - 1]$$

As `apex_list` computes each neighbour with a given apex in $O(1)$ per neighbour, and we ensured each internal node of T is the apex of some vertex, the time complexity is $O(k + d_v)$ where d_v is the degree of vertex v. The k term is for the k leaves, which may not give a neighbour. We may easily remove this term, by storing a length k bitvector, which for each leaf of $T(v)$, stores a 1 if the leaf contains the apex of some neighbour u, or it is the leftmost child of an internal node of $T(v)$ (so we do not skip any internal nodes of $T(v)$ if all its children contributes no neighbours). During the post-order traversal of $T(v)$ we may skip the leaves that is a 0 using `select`, to only visit those leaves which has a neighbour. Thus, we return all case 1 neighbours in $O(1)$ time per neighbour.

The total space needed is a length $n + |T| = O(n)$ bit vector A, a length k bitvector for each vertex (total $O(kn)$ bits) and a size $n \log n$ bit array AL.

Lastly, we note that this is essentially storing a permutation of the vertices, between the two orderings obtained by sorting leftmost leaf of $T(v)$, and sorting by apex of $T(v)$.

4.2.2 Case 2: vertices u with a_u an ancestor of a_v

As discussed in `adjacent`, vertices u in this case has the property that some leaf of $T(u)$ is a descendant of a_v. To compute these, we will store a 2D orthogonal range reporting data structure of Lemma 6. The points we will store is the set

$$\{(l_i, u) \mid u \in V,\ l_i \text{ is the } i\text{-th leaf of } T(u),\ 2 \le i \le k\}$$

This structure stores the leaves of all vertices u. Thus we will convert the adjacency criteria into something that can be expressed using axis-aligned rectangles.

We again compute the range of preorder numbers of the subtree rooted at a_v as $[l, r]$, where $l = a_v$ and r is the rightmost child of a_v. For the x-coordinate, we may use l and r to ensure that the leaf is within the subtree rooted at a_v. For the y-coordinate, since we use the label of the vertex rather than its preorder number, we need to covert l and r into the appropriate vertex labels. To determine the set of labels of vertices whose leftmost leaf falls within the subtree rooted at a_v, we find the 0-bits (which represents vertices) which belong to the l-th and r-th 1 bit in L (which denote the delimiting nodes in T). That is we have $l' = \texttt{rank}_0(L, \texttt{select}_1(L, l)) + 1$, which finds the first 0 after the l-th 1 in L and $r' = \texttt{rank}_0(L, \texttt{select}_1(L, r+1))$ which finds the first 0 before the $r+1$-th 1 in L.

Consider $T(u)$ which contains a_v and such that a_u is an ancestor of a_v. Since a_u is an ancestor of a_v, at least one leaf is outside of the subtree rooted at a_v (since a_u has degree at least 2) and since $T(u)$ contains the node a_v, at least one leaf is within the subtree rooted at a_v. There are two cases, either the leftmost leaf of $T(u)$ is outside of the subtree rooted at a_v or if the leftmost leaf falls within this subtree, at least one other leaf falls outside of the subtree.

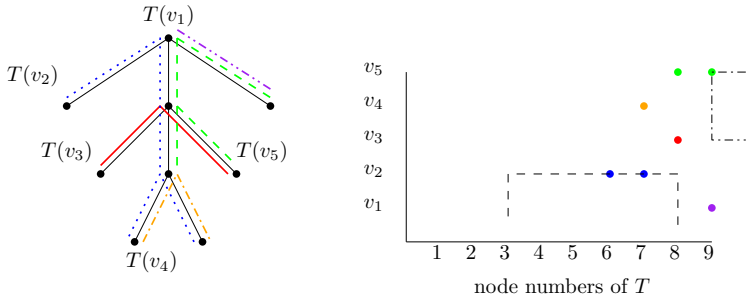

Figure 5 Example of our range search data structure. Left: a tree representation of a chordal graph (leafage 3), with 5 vertices. $T(v_1)$ in purple dash-dot-dot lines, $T(v_2)$ in blue dotted lines, $T(v_3)$ in red unbroken lines, $T(v_4)$ in orange dash-dot lines, and $T(v_5)$ in green dashed lines. For clarity, none of the simplifying operations (such as ensuring every node is the apex of some subtree) are applied to this tree representation. Right: The range search data structure for these subtrees. For v_1, the $(9, 1)$ is stored (since we do not store a point corresponding to the leftmost leaf) etc. The two types of rectangles are given for `neighborhood`(v_3). The subtree rooted at $a_{v_3} = 3$ spans the nodes $[3, 8]$ in T. The vertices whose leftmost leaf falling in this subtree are $[3, 5]$. Thus the first rectangle (in dashed lines) is $[3, 8] \times [-\infty, 2]$ and the second rectangle (in dash-dot lines) is $[9, \infty] \times [3, 5]$. In this case, the two vertices returned are v_2 and v_5.

In the first case, consider the rectangle, $[l, r] \times [-\infty, l')$. A point (l_i, u) that falls within this rectangle has the property that $l_i \in [l, r]$ so that the leaf l_i is a descendant of a_v. Furthermore, since $u < l'$, the leftmost leaf of $T(u)$ falls outside of the subtree rooted at a_v. See Figure 5.

In the second case where the leftmost leaf of the vertex u falls in the subtree rooted at a_v but some other leaf does not. This is captured by the rectangle $(r, \infty] \times [l', r']$.

Using the orthogonal range distinctness query, we retrieve each vertex once, since all the points corresponding to the same vertex has the same y-coordinate by construction. Thus, by Lemma 6 the time complexity is $O(\log n / \log \log n)$ per neighbour. The space cost of the 2D orthogonal range reporting data structure (on $(k-1)n$ points) is $(k-1)n \log kn + o(kn \log n)$ bits.

▶ **Theorem 11.** *Let G be a chordal graph with bounded vertex leafage k. Then there exists a data structure occupying $(k-1)n \log n - (k-1)n \log k + o(kn \log n)$ bits which answers* `adjacent` *in $O(\log k / \log \log n + 1)$ time. With additional $kn \log(kn)$ bits, it can answer* `neighborhood` *in $O(\log n / \log \log n)$ time per neighbour.*

4.3 Second data structure

We note that the previous data structure stores the coordinates of the leaves of the subtrees $T(v)$ twice, once in the array $Leaf(v)$ and once in the orthogonal range searching data structure to support `neighborhood` queries. This causes the space to be unnecessarily large to support `neighborhood` queries. We observe that the orthogonal range search data structure can be used to mimic access into the $Leaf$ array. By doing so, the time complexity for `adjacent` increases slightly, from $O(\log k / \log \log n)$ to $O(\log n / \log \log n)$, but the overall space cost can be reduced.

Recall that our orthogonal range search data structure stores the points:

$$\{(l_i, u) \mid u \in V, l_i \text{ is the } i\text{-th leaf of } T(u),\ 2 \leq i \leq k\}$$

To be able to compute all the queries, we need to be able to simulate access to the array $Leaf(v)$. For our purposes, this means predecessor and successor queries.

First we note that the relevant points corresponds to the rectangle $[-\infty, \infty] \times [v, v]$. Thus in $O(k \log n / \log \log n)$ time, we may retrieve the preorder numbers of each of the leaves for a given vertex.

For the predecessor and successor queries, the relevant points our query forms a horizontal strip, containing a single y-coordinate. Since predecessor queries corresponds to the point in the rectangle $[-\infty, i] \times [v, v]$ with the largest x-coordinate this corresponds to a orthogonal range distinct maximum query, with a single y-coordinate in the output. Thus, we may support it in $O(\log n / \log \log n)$ time. Symmetrically for successor queries.

Therefore, as accessing the array $Leaf(v)$ using predecessor and successor queries, using the orthogonal range search data structure incurs a cost of $O(\log n / \log \log n)$, we obtain the following theorem:

▶ **Theorem 12.** *Let G be a chordal graph with bounded vertex leafage k. There exists a data structure occupying $(k-1)n \log n + (k-1)n \log k + o(kn \log n)$ bits which answers* `adjacent` *in $O(\log n / \log \log n)$ time and, using $n \log n + o(n \log n)$ additional bits,* `neighborhood` *in $O(\log n / \log \log n)$ time per neighbour.*

4.4 Distances

In this section we investigate the `distance` query in chordal graphs with bounded vertex leafage k. In the same vein as Munro and Wu, we show a close relationship to a variation of the set intersection oracle problem. The set intersection oracle (or set disjointness) problem is the following:

Preprocess a collection of sets $\{S_1, \ldots, S_n\}$ with each set $S_i \subseteq U$ from some universe U. Answer queries of the form: given i, j is $S_i \cap S_j = \emptyset$?

There are a few ways to measure the size of this problem. One way is n, the number of sets. Another is $N = \sum_i |S_i|$, the total number of elements in the sets. As such, there are conjectures related to both of these measurements.

Patrascu and Roditty [26] gave the following conjecture, based on the number of sets in the input.

▶ **Conjecture 13** (Conj. 3 [26]). *Let $\{S_1, \ldots, S_n\}$ be a collection of sets, with each set $S_i \subseteq U$ from some universe U. Then any data structure answering set disjointness queries in $O(1)$ time, must use $\Omega(n^2)$ bits of space, even if $|U| = \log^c n$ for a large enough constant c.*

Observe that with n^2 bits we can write down the result matrix $M[i, j] = 1$ if $|S_i \cap S_j| > 0$ which allows the computation of the query in $O(1)$ time, and so this conjecture essentially states that this is the best we can do, even with small universes.

Using the total sizes of the sets as the input size, Goldstein et al. [19] gave the following conjecture.

▶ **Conjecture 14** (Conj 3. [19]). *Let $S_1, \ldots, S_n$ be a collection of sets, and let $N = \sum_i |S_i|$ be the total number of elements of the sets. Then any data structures answering set disjointness queries in T time must use $\Omega(N^2/T^2)$ space.*

These conjectures suggest that the set intersection oracle problem is quite difficult to solve without using trivial solutions such as writing down all the answers, or writing down the answers for large sets and naively iterating through smaller sets (i.e. we could check for every $x \in S_i$, is $x \in S_j$?, which would be quick if $|S_i|$ was small).

Recall our lower bound construction proof. For $k = o(n)$ we constructed two sets V_1 (which is a clique) and V_2 (which is an independent set). In this construction, for two vertices $v_1, v_2 \in V_2$, $\texttt{distance}(v_1, v_2) = 2$ exactly when there exists some vertex $u \in V_1$ that is adjacent to both. Otherwise, we pick any neighbour of v_1 and any neighbour of v_2. They both lie in V_1 which is a clique, so are adjacent. Hence the distance would be 3. Thus $\texttt{distance}(v_1, v_2)$ is either 2 or 3 depending on the condition: there exists some $u \in V_1$ adjacent to both v_1 and v_2.

Fix $v_i \in V_2$ and construct the set $S_i = \{u \in V_1 \mid (u, v_i) \in E\}$. Then the existence of u adjacent to both v_i and v_j is exactly the non-emptiness of the intersection $S_i \cap S_j$.

Munro and Wu further gave the reduction as follows: Given the sets $S_1, \ldots S_n \subseteq [1, \ldots, n]$ construct the split graph with $V_1 = [1, \ldots n]$ and $|V_2| = n$. For each $v_i \in V_2$, the neighbours of V_i is exactly the set S_i. Then the `distance` query on G answers the set disjointness problem.

If we do exactly the same thing, we obtain an obstacle. Since each vertex in V_1 has at most k neighbours in V_2 in a bounded vertex leafage chordal graph, this corresponds to the condition that: for every element $i \in U$, it belongs to at most k of the sets $S_1, \ldots, S_n$. This implies that $N \leq kn$, and also implies that $k^2 n$ bits suffice to write down the results of all queries, since for element $i \in U$, it causes at most k^2 pairs of sets to intersect.

Using this sparsity like condition, we give the following conjecture:

▶ **Conjecture 15.** *Let $\{S_1, \ldots, S_n\}$ be a collection of sets with universe $|U| = n$ such that for every $i \in U$, i appears in at most k sets. Any data structure which has query time $O(1)$ must use $\Omega(\min(k^2 n, n^2))$ bits of space.*

For chordal graphs with bounded vertex leafage, unless the above conjecture were to be false, we would be unable to answer `distance` queries in $O(1)$ time using succinct space (or even a little bit more space).

5 Data Structure for Chordal Graphs with Bounded Leafage

In this section, we consider data structures for chordal graphs with bounded leafage k. We note that a chordal graph with bounded leafage k is also a chordal graph with bounded vertex leafage k, and thus the data structures in the previous section applies. As this is a more structured class of graphs, our goal is to achieve much better query times, on top of not requiring additional space for neighbourhood queries. Furthermore, this structure allows us to bypass the difficulties with supporting the `distance` query. We will highlight some of the differences in our data structure that is made possible by this additional structure, which allows for more efficient computation.

By definition, a chordal graph with bounded leafage k has a tree representation that consists of a host tree T with at most k leaves. First, root T at one of its leaves. By doing so, we only need to worry about the $k-1$ other leaves of the tree. We may apply Lemma 9, to ensure that $|T| \leq n$ and that every node of T is the apex of some subtree $T(v)$.

Since there are at most $k-1$ leaves besides the root, we may break the tree down into $k-1$ disjoint paths $P_1, \ldots, P_{k-1}$, where each path contains 1 leaf of the tree, except P_1 which contains both a leaf and the root (which was also a leaf of the unrooted tree by construction). For simplicity, we may choose the paths in a preorder traversal, by descending down the tree until we reach the first leaf to obtain the first path, then following the preorder traversal to reach the second leaf to obtain the second path etc. In this way, the preorder traversal of the tree T gives $k-1$ blocks of nodes corresponding to the $k-1$ paths. In the following, we

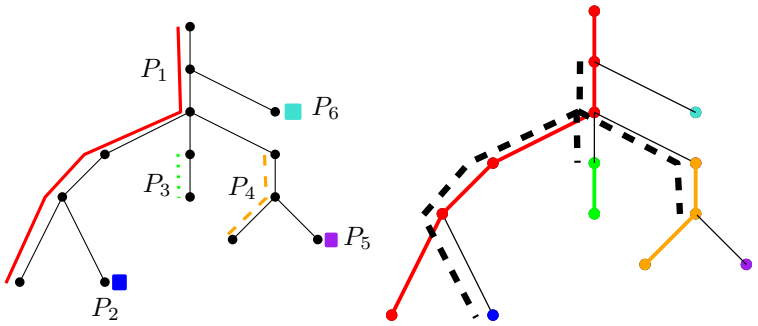

Figure 6 Left: An example host tree with 7 leaves (leafage 7), broken down into 6 paths so that the nodes of the paths are consecutive in preorder. The bitvector PN for this tree would be 10000011010011. Right: an example subtree $T(v)$ in black dashed with the host tree's path decomposition coloured in. The list of depths of the deepest node of $T(v)$ in the paths in order is 4, 0, 0, 1, -1, -1.

will refer to leaves of a subtree $T(v)$ which lie on some path P_i. However it may be the case that $T(v) \cap P_i$ is non-empty but no leaf of $T(v)$ exists on P_i. To ensure that each path P_i which contains nodes from $T(v)$ "has" a leaf of $T(v)$, we will consider the the largest depth node of $T(v)$ on a path P_i to be a leaf. See Figure 6, the subtree on the right, depicted by dashed black lines, has a "leaf" at the node of depth 4, on the red path (P_1).

Because there are only $k-1$ paths, for a subtree $T(v)$ to have k leaves, one of its leaves must also be the apex. Therefore, *rather than sorting by leftmost leaf as in the case of bounded vertex leafage, we now sort by their apex instead*, as obtaining the apex, will also obtain the leftmost leaf, if that is the case.

Sort the vertices by the preorder number of their apex, breaking ties arbitrarily and label the vertices as $1, \ldots, n$ by their index in this sorted order. From now on, when we refer to a vertex v, we refer to its label (i.e. v is an integer between 1 and n). Store a bitvector $L = 10^{N_1-1} \cdots 10^{N_{|T|}-1}$, where N_i is the number of vertices whose apex is the node with preorder number i. Then the preorder number of the apex of a vertex v is $\texttt{rank}_1(L, v)$. We note that by Lemma 9, this bitvector is well defined, as $N_i > 0$.

For each vertex v and for each of the $k-1$ paths, we store the depth (in the path) of the largest depth node (i.e. the leaf of $T(v)$ on that path) of $T(v)$ on that path, and if $T(v)$ does not intersect that path, we store -1 instead. Let $Leaf(v, j)$ be the depth of this leaf of $T(v)$ on path P_j. We store the partial sums $\sum_{1 \le \le j}(Leaf(v, j) + 2)$ (to ensure that the terms are increasing). Applying Lemma 2, and using the fact that the total sum equals $n + k$ (as the paths are disjoint, so their total lengths equal $|T| \le n$), and that $k = o(n)$, this uses $(k-1)\log(n+k) - (k-1)\log k + o(k + \log n)$ bits. To compute $Leaf(v, j)$, we have $Leaf(v, j) = \texttt{select}(j) - \texttt{select}(j-1) - 2$.

We note that since the i-th leaf always falls on the i path, instead of being arbitrarily placed in the tree as in the bounded vertex leafage case, we may retrieve it in $O(1)$ time, rather than requiring predecessor queries which takes longer.

Finally we store a bitvector PN (path number) in the preorder traversal of the tree T, with a 1 bit if a node is the first node of some path P_i. Given a node $x \in T$ (with preorder number x), it lies on the path $\texttt{rank}_1(PN, x)$.

The total space needed so far is $(k-1)n \log n - kn \log k + o(kn \log n)$ bits.

Again, using these data structures, we are able to compute the exact subtree $T(v)$ for any vertex v given as an integer denoting its position in sorted order: the root of $T(v)$ is the node with preorder number $\texttt{rank}_1(L, v)$, and we store the at most $k-1$ leaves of

the tree $T(v)$ by their depths $Leaf(v, i)$. For path i, the first node (by preorder number) of the path is $\texttt{select}_1(PN, i)$, so the leaf of $T(v)$ is the node of T with preorder number $\texttt{select}_1(PN, i) + Leaf(v, i)$ (if $Leaf(v, i) \neq -1$). Thus by reconstructing the subtress $T(v)$ we are able to recover the graph.

Now we will show how to answer the queries efficiently.

5.1 `adjacent` (u, v)

Find a_u and a_v using L. Then by performing `LCA` in T, determine if one is the ancestor of the other. If not, return not adjacent. Otherwise, suppose that a_u is an ancestor of a_v. Let j be the path number of a_v. We check if the j-th leaf of $T(u)$, $Leaf(u, j)$, has depth at least that of a_v (relative to P_j). If so, return adjacent, and return not adjacent otherwise. This uses a constant number of bitvector and succinct tree operations, and so takes $O(1)$ time.

5.2 `neighborhood` (v)

We consider two cases. Case 1: vertices u adjacent to v such that a_u is an ancestor of a_v. Case 2: vertices u adjacent to v such that a_u is a descendant of (or equal to) a_v.

Case 1. Let j be the path number of a_v, and let d_v be the depth of a_v relative to this path. As in `adjacent`, we want vertices u whose j-th leaf has depth at least as much as d_v. *Compared to the previous section, knowing that we only need to consider the j-th leaf, rather than an arbitrary leaf allows us to forgo a orthogonal range search data structure.* Let $D_j[u] = Leaf(u, j)$ be a conceptual array. We are looking for indices u such that $u < v$ (so that it is an ancestor) and $D_j[u] > d_v$. It is well known that this can be accomplished in $O(1)$ time per index by equipping D_j with a RMQ data structure (Lemma 5). The space needed is $O(kn)$ bits as we need one RMQ data structure per path.

Case 2. We consider each path in order. For each vertex, we store a length k bitvector indicating whether the j-th leaf is -1, and using `select` we iterate over each path with a node in $T(v)$. For path j if $T(v) \cap P_j$ is non-empty, let x_1, x_2 be the first and last vertices of $T(v) \cap P_j$. x_2 is simply the j-th leaf, and x_1 is the first node of the path, which we obtain using $\texttt{select}(PN, j)$, except on the path that a_v is on, in which case, $x_1 = a_v$. Since we sorted the vertices by their apex, the set of vertices with apex in $[x_1, x_2]$ forms a consecutive block, which we may find using `select` on L as $[\texttt{select}_1(L, x_1), \texttt{select}_1(L, x_2 + 1) - 1]$. This takes $O(1)$ time per neighbour.

5.3 Iteration through Neighbourhood

It maybe the case that for a graph algorithm, storing the entire neighbourhood of a vertex, especially over a large number of recursive calls is too costly. Instead it would be desirable to iterate over the list of neighbours, so that at any point, only a constant amount of space is needed to store where in the list of neighbours the computation is currently at.

To do so, we consider the two cases in the $\texttt{neighborhood}(v)$ query. Let a_v be the apex of v. First consider the second case, where we output neighbours u of v such that a_u is a descendant of a_v in the host tree T. In this case, we examined each of the $(k-1)$-paths which contained a node of $T(v)$, then for each node on the j-th path $P_j \cap T(v)$, we returned all vertices with that node as its apex. The vertices whose apex is on this subpath $P_j \cap T(v)$ forms a single interval of vertex labels. Therefore, to store where we are in this computation,

we need to store the current path number (from which we may compute the subpath in $O(1)$ time), and the current node in this single interval that we have returned. This uses $O(1)$ words of space.

Now consider the first case, where we output neighbours u of v such at a_u is an ancestor of a_v. Let i be the path number of a_v. In this case, we used a RMQ query on the conceptual array D_i which stored the leaf numbers of each vertex in the i-th path (which may be -1 if a vertex did not have a vertex on the i-th path). In this case, we wished to output all indices u in the prefix $D_i[1, \ldots, v']$ such that $D_i[u] \geq \texttt{depth}_{P_j}(a_v)$.

A solution to this was given by Tsakalidis et al. [30] in their work for permutation graphs. Their extension to the regular RMQ data structure can be stated as:

▶ **Lemma 16** (Lemma 3.2 of [30]). *Let $A[1..n]$ be a static array of comparable elements. For any constant $\varepsilon > 0$, there is a data structure using εn bits of space on top of A that supports the following queries in $O(\frac{1}{\varepsilon})$ time (making as many queries to A) and using $O(1)$ words of working memory:*

1. *range-maximum queries (or symmetrically range-minimum queries), $RMQ_A(\ell, r)$,*
2. *next-above queries (or symmetrically next-below), $next\_above_A(\ell, r, y; i)$, enumerating $\{i \in [\ell, r] : A[i] \geq y\}$ in amortized $O(\frac{1}{\varepsilon})$ time.*
   Formally, next_above implicitly defines a sequence $(i_j)_{j \geq 0}$ via $i_0 = RMQ_A(\ell, r)$ if $A[i_0] \geq y$ and $i_0 = null$ otherwise, and $i_{j+1} = next\_above_A(\ell, r, y; i_j)$ if $i_j \neq null$, and $i_{j+1} = null$ otherwise. Then we require $\{i_j : i_j \neq null\} = i \in [\ell, r] : A[i] \geq y$.

We note that this data structure requires access to the array A, as opposed to the data structure of Lemma 5. Here $A = D_i$, and accessing elements of $D_i[u] = Leaf(u, i)$ can be done in $O(1)$ time. Lastly, we note that the *next_above* operation iterates over all indices that we need.

Therefore, combining these two cases, we may store a reference to where we are in the list of neighbours in $O(1)$ words.

5.4 `distance`(u, v)

In the previous section, for chordal graphs with bounded vertex leafage, we argued that it is difficult to support the distance query, due to a close connection to the set disjointness problem. Applying the same reduction, we see that we no longer have such an obstacle. Indeed, as the host tree contains at most k leaves, the instances of the set disjointness problem that can be reduced to chordal graphs with bounded leafage are exactly those with at most k input sets. As we noted, we may store the output matrix in this case using k^2 bits, which is much less than $(k-1)n \log n$ bits. Thus, the distance query seems hopeful to be solved for the bounded leafage case.

To compute the distance between two vertices, we will use the greedy algorithm of Munro and Wu [25] for general chordal graphs. Informally, to connect two subtrees $T(u)$ and $T(v)$ (using a sequence of subtrees), at least one subtree must pass through the lowest common ancestor of a_u and a_v. To reach this LCA from a_u, we greedily pick the subtree $T(w)$ intersecting $T(u)$ such that a_w is as high up in the tree as possible (and repeat).

The successor operation on the tree T is defined by: $succ(x) = y$ (x, y are nodes of T) where y is the smallest depth node such that there exists a vertex w with $y = a_w$ and $x \in T(w)$. A shortest path between u and v can be found by first computing the node $h = \texttt{LCA}(a_u, a_v)$, then repeatedly take the successor operation from a_u and a_v. We stop when the next successor operation would give a node that has smaller depth than h. The result of the two chains of successor operations are two nodes h_u (an ancestor of a_u) and h_v (an

ancestor of a_v). Lastly, we compute whether there exists a subtree $T(w)$ such that $h_u \in T(w)$ and $h_v \in T(w)$. The shortest path between any vertex with apex h_u and any vertex with apex h_v is $h_u \to w \to h_v$ if such a subtree $T(w)$ exists and $h_u \to succ(h_u) \to succ(h_v) \to h_v$ if no such $T(w)$ exists.

We assume that $h \neq a_u, a_v$ (i.e. h_u and h_v are not in an ancestor-descendant relationship), since if one is the ancestor of the other, then there is a single chain of successor operations and we do not need to compute if $T(w)$ exists or not.

Suppose that we are given h_u and h_v, two nodes of T. Let P_u, P_v be the two paths containing h_u and h_v. For now, assume that P_u and P_v are such that no node of one path is an ancestor of another path. For each node of P_u and P_v we wish to store a bit which is 1 if there exists a vertex w whose subtree $T(w)$ contains the two nodes. This allows us to finish the distance computation by checking this bit using the nodes h_u and h_v. Naively, this will use $|P_u||P_v|$ bits of space if we store a bit for each pair of nodes of the two paths.

Let $B_{u,v}[i][j]$ be this bit for the node at depth i in P_u and depth j in P_v.

▶ **Lemma 17.** *The bitarray $B_{u,v}[i]$ consists of a block of 1s followed by a block of 0s.*

Proof. Let x be the node of depth i on the path P_u. Let j be an index such that $B_{u,v}[i][j] = 1$. Then there exists some subtree $T(w)$, which contains both x and the depth j node of P_v. Since we assume that no node of P_u is an ancestor of P_v (and vice versa), $T(w)$ must also contain the node at depth $j-1$ on path P_v. Hence any 1 bit of $B_{u,v}[i]$ is preceded by another 1 bit, and we conclude that this bitarray consists of a block of 1s followed by a block of 0s. ◀

Using the above lemma, we may describe this bitvector $B_{u,v}[i]$ using a single number: the index of the last 1. Thus, consider the array $A_{u,v}$ of length $|P_u|$ which contains these indices of the last 1.

▶ **Lemma 18.** *The array $A_{u,v}$ is non-increasing.*

Proof. We show that $A_{u,v}[i] \geq A_{u,v}[i+1]$ for all i. Let $T(w_i)$ be the subtree which realizes $A_{u,v}[i]$. That is, it is the subtree which contains the node x_i of depth i on P_u (among all such subtrees) which contains the deepest node on P_v (which is of depth $A_{u,v}[i]$). Similarly, let $T(w_{i+1})$ be the subtree which realizes $A_{u,v}[i+1]$. Note that since $T(w_{i+1})$ contains the node x_{i+1} it must also contain x_i (we assumed that no node of either path is an ancestor of nodes of the other path). Therefore, by definition of $T(w_i)$, its deepest nodes on P_v is at least as deep as that of $T(w_{i+1})$. ◀

Now, as $A_{u,v}$ is a non-increasing sequence (i.e. sorted) of length $|P_u|$ with maximum value $|P_v|$, we may store it using $|P_u| + |P_v| + o(|P_u| + |P_v|)$ bits by Lemma 3. To obtain $B_{u,v}[i][j]$, we compute $j \leq A_{u,v}[i]$, and if so the bit is 1.

Next we deal with the case that some nodes of P_u and P_v and in an ancestor-descendant relationship. Since we assume that h_u and h_v are not in an ancestor-descendant relationship, if the paths P_u and P_v contain nodes that are in an ancestor-descendant relationship, we may ignore them, as we will never query those pairs. Suppose that $P_u[0, l]$ are ancestors of the nodes of P_v (observe that only one path can contain such nodes and they must be a prefix), then we perform the above calculations using $P'_u = P_u[l+1, -]$, and store $A[0] = A[1] = \cdots = A[l] = A[l+1]$ so that sequence is still non-increasing. Note that for this pair of paths, $A[0, \ldots, l]$ will never be accessed so their values can be anything. See Figure 7.

The total size of these non-increasing sequences, stored using Lemma 3 is then $\sum_{i<j} |P_i| + |P_j| \leq k \sum_i |P_i| = kn + o(kn)$ bits. The time to compute the distance query is $O(1)$. We may also compute the shortest path by storing the vertex w_i giving rise to the value of $A_{u,v}[i]$. This takes a further $kn \log n$ bits.

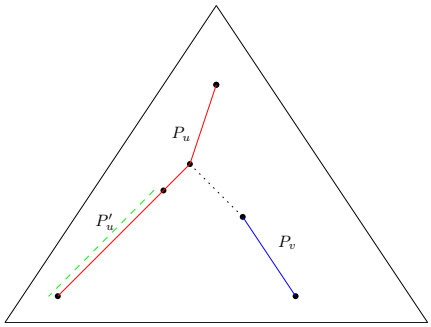

Figure 7 The case when some nodes of one path are ancestors of nodes of the other path. We ignore such nodes and work with the subpath that are not ancestors P'_u, before padding the final sequence with the ignored elements.

The total space needed is $(1+\varepsilon)n\log n$ bits for the necessary data structures from Munro and Wu's data structure. Our data structure for computing the existence of the vertex w uses $O(kn)$ bits for the `distance` query. However $kn\log n$ bits is needed to store the vertices w themselves for the `spath` query which asks for the vertices on a shortest path.

Putting the above together, we obtain the following theorem:

▶ **Theorem 19.** *Let G be a chordal graph with bounded leafage k. There exists a data structure occupying $(k-1)n\log n-(k-1)n\log k+o(kn\log n)$ bits of space supporting* `adjacent` *queries in $O(1)$ time, and* `neighborhood` *in $O(1)$ time per neighbour. Furthermore, we may iterate through the neighbourhood using $O(1)$ words to denote the current neighbour in the list, in $O(1)$ amortized time per neighbour.*

We may support `distance` *queries using $(1+\varepsilon)n\log n+O(kn)$ additional bits in $O(1)$ time (for any constant $\varepsilon>0$), and shortest path queries using $(k+1+\varepsilon)n\log n$ additional bits in $O(1)$ time per vertex on the path.*

References

1 Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct encodings for families of interval graphs. *Algorithmica*, 83(3):776–794, 2021. doi:10.1007/s00453-020-00710-w.

2 Girish Balakrishnan, Sankardeep Chakraborty, Seungbum Jo, N. S. Narayanaswamy, and Kunihiko Sadakane. Succinct data structure for graphs with d-dimensional t-representation. In Ali Bilgin, James E. Fowler, Joan Serra-Sagristà, Yan Ye, and James A. Storer, editors, *Data Compression Conference, DCC 2024, Snowbird, UT, USA, March 19-22, 2024*, page 546. IEEE, 2024. doi:10.1109/DCC58796.2024.00063.

3 Girish Balakrishnan, Sankardeep Chakraborty, N. S. Narayanaswamy, and Kunihiko Sadakane. Succinct data structure for chordal graphs with bounded vertex leafage. In Hans L. Bodlaender, editor, *19th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2024, June 12-14, 2024, Helsinki, Finland*, volume 294 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SWAT.2024.4.

4 Girish Balakrishnan, Sankardeep Chakraborty, N.S. Narayanaswamy, and Kunihiko Sadakane. Succinct data structure for path graphs. *Information and Computation*, 296:105124, 2024. doi:10.1016/j.ic.2023.105124.

5 E. A. Bender, L. B. Richmond, and N. C. Wormald. Almost all chordal graphs split. *Journal of the Australian Mathematical Society. Series A. Pure Mathematics and Statistics*, 38(2):214–221, 1985. doi:10.1017/S1446788700023077.

6 C. Berge. Some classes of perfect graphs. In *Graph Theory and Theoretical Physics*, pages 155–165. Academic Press, London-New York, 1967.

7 Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976. doi:10.1016/S0022-0000(76)80045-1.

8 Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In Frank K. H. A. Dehne, Marina L. Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth, editors, *Algorithms and Data Structures, 11th International Symposium, WADS 2009, Banff, Canada, August 21-23, 2009. Proceedings*, volume 5664 of *Lecture Notes in Computer Science*, pages 98–109. Springer, 2009. doi:10.1007/978-3-642-03367-4_9.

9 Boris Bukh and R. Amzi Jeffs. Enumeration of interval graphs and d-representable complexes, 2023. doi:10.48550/arXiv.2203.12063.

10 Peter Buneman. A characterisation of rigid circuit graphs. *Discret. Math.*, 9(3):205–212, 1974. doi:10.1016/0012-365X(74)90002-8.

11 Steven Chaplick and Juraj Stacho. The vertex leafage of chordal graphs. *Discret. Appl. Math.*, 168:14–25, 2014. Fifth Workshop on Graph Classes, Optimization, and Width Parameters, Daejeon, Korea, October 2011. doi:10.1016/j.dam.2012.12.006.

12 Clark, David. *Compact PAT trees*. PhD thesis, University of Waterloo, 1997. URL: http://hdl.handle.net/10012/64.

13 G. A. Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 25:71–76, 1961. URL: https://api.semanticscholar.org/CorpusID:120608513.

14 Arash Farzan and Shahin Kamali. Compact navigation and distance oracles for graphs with small treewidth. *Algorithmica*, 69(1):92–116, 2014. doi:10.1007/s00453-012-9712-9.

15 Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74450-4_41.

16 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

17 Fănică Gavril. A recognition algorithm for the intersection graphs of paths in trees. *Discrete Mathematics*, 23(3):211–227, 1978. doi:10.1016/0012-365X(78)90003-1.

18 Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974. doi:10.1016/0095-8956(74)90094-X.

19 Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In Faith Ellen, Antonina Kolokolova, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 421–436, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-62127-2_36.

20 Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, 1980.

21 Meng He, J. Ian Munro, Yakov Nekrich, Sebastian Wild, and Kaiyu Wu. Distance oracles for interval graphs via breadth-first rank/select in succinct trees. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ISAAC.2020.25.

22 Meng He, J. Ian Munro, and Kaiyu Wu. Succinct data structures for path graphs and chordal graphs revisited. In Ali Bilgin, James E. Fowler, Joan Serra-Sagristà, Yan Ye, and James A. Storer, editors, *Data Compression Conference, DCC 2024, Snowbird, UT, USA, March 19-22, 2024*, pages 492–501. IEEE, 2024. doi:10.1109/DCC58796.2024.00057.

23 Douglas B. West In-Jen Lin, Terry A. McKee. The leafage of a chordal graph. *Discussiones Mathematicae Graph Theory*, 18(1):23–48, 1998. doi:10.7151/dmgt.1061.

24 Clyde L. Monma and Victor K.-W. Wei. Intersection graphs of paths in a tree. *Journal of Combinatorial Theory, Series B*, 41(2):141–181, 1986. doi:10.1016/0095-8956(86)90042-0.

25 J. Ian Munro and Kaiyu Wu. Succinct data structures for chordal graphs. In *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, pages 67:1–67:12, 2018. doi:10.4230/LIPIcs.ISAAC.2018.67.

26 Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. *SIAM J. Comput.*, 43(1):300–311, 2014. doi:10.1137/11084128X.

27 Mihai Patrascu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 166–175, 2014. doi:10.1109/FOCS.2014.26.

28 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, November 2007. doi:10.1145/1290672.1290680.

29 Donald J Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970. doi:10.1016/0022-247X(70)90282-9.

30 Konstantinos Tsakalidis, Sebastian Wild, and Viktor Zamaraev. Succinct permutation graphs. *Algorithmica*, 85(2):509–543, 2023. doi:10.1007/s00453-022-01039-2.

31 Nicholas C. Wormald. Counting labelled chordal graphs. *Graphs and Combinatorics*, 1(1):193–200, 1985. doi:10.1007/BF02582944.