Succinct Data Structures for Path Graphs and Chordal Graphs Revisited

Meng He*, J. Ian Munro†, and Kaiyu Wu*

*Faculty of Computer Science
Dalhousie University
Halifax, NS, Canada
{mhe@cs.,kevin.wu@}dal.ca

†Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
imunro@uwaterloo.ca

Abstract

We enhance space efficient representations of two types of intersection graphs. We refine the data structure for path graphs of Balakrishnan et al. to give a succinct data structure of $n \log n + o(n \log n)$ bits that supports adjacency test, degree and neighbourhood queries in $O(\frac{\log n}{\log \log n})$ time (for neighbourhood queries, this is the amount of time for each neighbour reported). To achieve $O(1)$ query times, we give a data structure using $(3 + \varepsilon)n \log n + o(n \log n)$ bits for any constant $\varepsilon > 0$. Furthermore, we are able to support both the distance and shortest path queries on unweighted path graphs using $(2+\varepsilon)n \log n + o(n \log n)$ bits in $O(\log n / \log \log n)$ time (shortest path uses an additional $O(1)$ time per vertex on the path). This is the first compact distance oracles for path graphs. Turning to chordal graphs, we enhance the succinct data structure of Munro and Wu to reduce all query times including performing adjacency test in $O(1)$ time.

1 Introduction

In this paper we examine data structures for path graphs and chordal graphs that are both time and space efficient. The ultimate goal is to achieve representations that are "succinct" (i.e. using within a lower order term of the information theoretic amount of space). We include some methods that are "compact" (i.e. within a constant factor of that space) but give better run times. Both path graphs and chordal graphs are *intersection graphs*, where the edges of the graphs are encoded in the intersection structure of a collection of sets. By restricting the types of sets, we obtain different graph classes, and path graphs are graphs obtained when the sets are the nodes of a simple path in a tree, and chordal graphs are graphs obtained when the sets are the nodes of a subtree of a tree. Chordal graphs in particular has applications in compiler construction [1] and data bases [2].

A variety of other classes of graphs that have been considered in the literature, we have [3, 4] for interval graphs, proper interval graphs and circular arc graphs. Chakraborty and Jo [5] deals with interval graphs with degree or chromatic number at most σ. Farzan and Kamali [6] deals with graphs of bounded treewidth, Chakraborty et al. [7] covers series-parallel, block-cactus and 3-leaf power graphs, and Tsakalidis et al. [8] focuses on permutation graphs.

The queries supported by these graph data structures are the standard navigational operations: `adjacency`: given two vertices return whether they are adjacent; `degree`: given a vertex, return its degree; `neighbourhood`: given a vertex, return

Type	Authors	Space	`adjacency`	`neighbourhood`	`degree`	`dist`[a]
Path Graph	[9]	$n \log n$	$O(\log n)$	$O(d \log n)$	$O(\log^2 n)$	-
	Thm. 1	$n \log n$	$O(\frac{\log n}{\log \log n})$	$O(d\frac{\log n}{\log \log n})$	$O(\frac{\log n}{\log \log n})$	-
	Thm. 3	$(2+\varepsilon)n \log n$	$O(\frac{\log n}{\log \log n})$	$O(d\frac{\log n}{\log \log n})$	$O(\frac{\log n}{\log \log n})$	$O(\frac{\log n}{\varepsilon \log \log n})$
	[9]	$O(n \log^2 n)$	$O(1)$	$O(d)$	$O(1)$	-
	Thm. 2	$(3+\varepsilon)n \log n^b$	$O(1)$	$O(d)$	$O(1)$	-
Chordal Graph	[10]	$n^2/4$	$O(f(n))^c$	$O(df(n)^2)$	$O(1)$	$O(nf(n))$
	Thm. 4	$n^2/4$	$O(1)$	$O(df(n))$	$O(1)$	$O(n)$

[a]We use - to denote that it is not supported. When it is supported, `shortest_path` is also supported using an additional $O(1)$ time per vertex returned)

[b]Time dependence on ε is $O(\frac{1}{\varepsilon})$

[c]For any $f(n) \in \omega(1)$.

Table 1: Comparison of our results with previous results. Here d denotes the degree of the vertex. Lower order terms in the space are omitted.

all vertices adjacent to it. Some of the data structures also support distance related operations: `shortest_path`: given two vertices, return a shortest path between them; and `dist`: given two vertices, return the length of a shortest path between them.

Our Results

For a comparison of our results to previous results, see Table 1. In section 3, we give succinct and compact data structures for path graphs, improving the result of Balakrishnan et al. [9]. In particular, our succinct data structure (using $n \log n + o(n \log n)$ bits of space) is able to answer queries in $O(\frac{\log n}{\log \log n})$ time, rather than $O(\log n)$ (or $O(\log^2 n)$ time for `degree`) time, and is much simpler. To achieve optimal query times, our compact data structure occupies $(3+\varepsilon+o(1))n \log n$ bits rather than $O(n \log^2 n)$ bits.

To achieve these results, we will first use simpler path intersection criteria and furthermore leverage the folklore result on solving the 2-dimensional 3-sided range reporting queries in linear bits of space, when the coordinates of the points are easily calculated. The simpler intersection criteria also allows our data structures to be much simpler as well. This simplicity allows us to adapt the succinct data structure to achieve $O(1)$ query times using only around thrice the optimal space.

In section 4, we show that we can augment the succinct path graph data structure using an extra $(1+\varepsilon)n \log n$ bits to support the `dist` query in $O(\log n / \log \log n)$ time. This is the first compact distance oracle for path graphs.

Finally in section 5, we improve the run times of the two foundational operations: `adjacency`, `decode`, used by Munro and Wu [10] in their succinct chordal graph data structure. We reduce the running time from $O(f(n))$ for any $f \in \omega(1)$ (any non-constant increasing function) to constant $O(1)$. As all the operations are built upon these, consequently, we obtain speedups in the graph operations.

Due to space constraints, omitted details and proofs can be found in [11].

2 Preliminaries

In this paper, we will use standard graph theoretic notation. We will use $G = (V, E)$ to denote a graph with vertex set V and edge set E. We will use $n = |V|$ and $m = |E|$ to denote the number of vertices and edges. All of our graphs will be unweighted. As we will be discussing both trees and graphs in general, we will use vertices to denote the vertices of a graph which may or may not be a tree, and nodes to denote the vertices of a tree. We assume the word-RAM model with $\Theta(\log n)$-size words. We use $\log(\cdot)$ to denote $\log_2(\cdot)$.

Intersection Graphs

Let U be a set and X be a collection of subsets of U. We may define a graph G with $V = X$ (one vertex v_s for each $s \in X$) with edges defined by $(v_s, v_t) \in E$ exactly when $s \cap t \neq \emptyset$. In this way, we say that X is an intersection model for G. Erdös et al. [12] showed that all graphs can be represented in this way, with $|U| \leq n^2/4$, and thus, we restrict both U and X to generate interesting subsets of graphs.

Following this, a chordal graph can be defined as a graph where we take U to be the nodes of a tree T, and insist that any $s \in X$ must be a subtree of T [13]. A path graph is a graph where again we take U to be the nodes of a tree, but now we insist that any $s \in X$ must be a (simple) path in T [14, 9] [1]. An interval graph is a graph where we now insist that U is a path itself [15], though typically it is defined using intervals on the real line. It is easy to see that these definitions are the same. By these definitions it is clear that interval graphs $\subseteq$ path graphs $\subseteq$ chordal graphs.

A maximal clique is a clique that is maximal under inclusions. It is known that for a chordal graph G, the number of maximal cliques of G is at most n [16], and thus this holds true for both path graphs and interval graphs. Furthermore, for chordal graphs, we may arrange the maximal cliques as the nodes of a tree T so that for every vertex v, the nodes corresponding to maximal cliques which contain v form a (connected) subtree of T [17]. Similarly, for a path graph, we may arrange the maximal cliques as the nodes of a tree T so that for every vertex v, the nodes containing v form a (simple) path in T [14]. We will denote a tree obtained in this way, which has at most n nodes as a *clique tree* of the graph. In both of these cases, a clique tree of the graph is an intersection model for G.

Succinct Data Structures

A bit vector is a length n array of bits, that supports the queries `rank`(i): given an index, return the number of 1s up to index i, `select`(j): given a number j, return the index of the jth one in the array, and `access`(i): return the bit at index i.

Lemma 1 ([18])**.** *A bit vector of length n can be succinctly represented using $n + o(n)$ bits to support* `rank`*,* `select` *and* `access` *in $O(1)$ time.*

We will be working with trees and will be using various tree operations.

[1] Path graphs are also commonly used to refer to paths themselves, but as in the previous works, we will use it to refer to this class of intersection graphs.

Lemma 2 ([4]). *An ordinal tree on n nodes can be represented succinctly using $2n + o(n)$ bits and can support a variety of operations in $O(1)$ time.*[2] *These operations include* `level-anc`(v, i)*: return the ancestor of v at depth i,* `LCA`(u, v)*: return the lowest common ancestor of u and v, and* `last_child`(v)*: return the last (rightmost) child of v.*

Permutations: Let P be a permutation of $[1 \dots n]$. The two operations we are interested in are computing both $P[i]$ and $P^{-1}[i]$.

Lemma 3 ([19]). *Let P be a permutation. Then P may be represented using $(1 + 1/f(n))n \log n + o(n \log n)$ bits to support the computation of P and P^{-1} in $O(f(n))$ time ($1 \leq f(n) \leq n$). In particular, if we set $f(n) = \frac{1}{\varepsilon}$ for constant $\varepsilon > 0$, then the space is $(1 + \varepsilon)n \log n + o(n \log n)$ bits and the time is $O(1)$.*

Orthogonal Range Queries: In these data structures, we store n d-dimensional points. The queries we wish to answer are: given a d-dimensional axis aligned rectangle $[p_1, p_2] \times [p_3, p_4] \dots [p_{2d-1}, p_{2d}]$, *emptiness*: does the rectangle contain a point? *count*: how many points does the rectangle contain? *reporting*: return each point contained in the rectangle. We say that the rectangle is k-sided if there are at most k coordinates that are finite. The following data structures sort the points and stores stores them as indices in that sorted order, and thus we will define the operations `rsrank` and `rsdecode` to convert between the actual coordinates of points and their indices in the sorted order (typically referred to as *rank-space*). In the case of 2-dimensional 3-sided reporting queries, the following is a folklore result using range minimum queries:

Lemma 4. *Given n 2-dimensional points, and let f be the time cost of* `rsrank`*, g be the time cost of* `rsdecode`*. 3-sided reporting queries (of the form $[x_1, x_2] \times [-\infty, y]$, to support the symmetric cases, we duplicate the structure) can be supported in time $O(f + k \cdot g)$ where k is the number of points reported. The space cost is $2n + o(n)$ plus the space needed to support the* `rsrank` *and* `rsdecode` *operations.*

We will also use the result of Bose et al. [20] for 4-sided queries.

Lemma 5 ([20]). *Let Q be a set of points from the universe $M = [1..n] \times [1..n]$, where $n = |Q|$. Q can be represented using $n \log n + o(n \log n)$ bits to support orthogonal range count in $O(\log n / \log \log n)$ time, and orthogonal range reporting in $O(k \log n / \log \log n)$ time, where k is the size of the output.*

Path-Sum Query: We are given a tree T, where each vertex v has a non-negative integer weight w_v of at most σ.[3] Given a simple path in T consisting of the vertices $v_1, \dots v_k$, we want to compute the sum $\sum w_{v_i} \mod \sigma$. Chan et al. [21] gave a solution:

Lemma 6. *Let T be an ordinal tree on n nodes, each having a weight of at most σ. Then T can be stored using $n \log \sigma + 2n + o(n \log \sigma)$ bits of space and $O(n)$ construction time to support path sum queries in $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function.*

[2] For the full list see Table 1 of their paper.

[3] The problem is defined over general semi-groups, which abstract the addition operation.

This can be easily modified to give a time-space trade-off (proof omitted):

Lemma 7. *Let T be an ordinal tree on n nodes, each having a weight of at most σ. Fix Δ. Then T can be stored using $(n \log \sigma)/\Delta + 3n + o((n \log \sigma)/\Delta)$ bits of space to support path sum (modulo σ) queries in $O(\alpha(n))$ time plus $O(\Delta)$ accesses to weights of the tree, where $\alpha(n)$ is the inverse Ackermann function.*

3 Path Graph Data Structure

In this section, we will give two path graph data structures for supporting `adjacency`, `degree` and `neighbourhood`, though much of the details for the compact data structure will be omitted. The information-theoretic lower bound for path graphs is $n \log n - o(n \log n)$ bits, as path graphs are a superclass of interval graphs, the interval graph lower bound [3] applies. A matching upper bound is given by [9].

Tree Structure: We will begin with the clique tree T of G - an arrangement of the maximal cliques as the nodes of a tree so that for every vertex v of G, the nodes of the tree containing v form a simple path P_v. We will make several modifications to T while preserving the path intersections. This will allow our data structures to be simpler. First we root the tree arbitrarily. Let w be an endpoint of a path P_v. Create a child of w and extend the path to the child. We note that since P_v is the only path through the child, no additional intersections are created. By doing so, we add $2n$ nodes to T and guarantee that all endpoints of paths are unique.

Next for each internal node w, add a child node c_w as its last child. Consider the preorder numbers of the nodes in the subtree rooted at w. The smallest number is w and the largest is c_w. This also has the nice property that, if we are given a node c and are told that it is the largest preorder numbered node of some subtree (that is not the subtree rooted at c in the case that c is a leaf), we know that this subtree must be the one rooted at the parent of c. This at most adds another n nodes to T.

Finally, for each internal node a, such that a is the lowest depth node of some path (which we denote as the *apex* of the path), consider all paths with a as its apex. Each path passes through two children of a, and we "mark" the child of a that the left branch of the path passes through. Reorder the children of a such that marked children come before unmarked children (in a stable manner, the order of any two marked children should not be changed - thus for any path, the left branch remains to the left of the right branch) [4].

We will refer to the nodes of T by their preorder numbers. Each vertex v is associated with a path $P_v = (l_v, r_v)$ in T, where $l_v < r_v$ are the two endpoints of the path. We will name the paths based on the values of the left endpoints, so that vertex $v \in [1, n]$ has the v-th smallest (as endpoints are unique, we do not have ties) left endpoint. We will abuse notation and use v to refer to the vertex in G, and the path P_v as well.

For a path $v = (l_v, r_v)$, the apex a_v is the node $\texttt{LCA}(l_v, r_v)$ in T. We note that since we extended the paths to "dummy" vertices, $a_v \neq l_v, r_v$. Given the apex of the

[4] It is worth noting that the last child node c_w added previously remains the last node, since it does not have any path through it and thus is unmarked.

path, denote the last node (by preorder numbers, and by construction this is the last child) in the subtree rooted at a_v by z_v.

Succinct Data Structure

We first define the internal operation $\texttt{apexlist}(a)$ which returns a list of all paths u with $a_u = a$. Our data structure will consist of the following pieces:
- We will store 4 bitvectors, L, R, A, Z where $X[i] = 1$ if the i-th node in preorder is the left endpoint/right endpoint/apex/z_v for some vertex v for $X = L, R, A, Z$ respectively.
- Since we do not know which left endpoint matches which right endpoint, we will consider the permutation $P[i] = j$ where the i-th path is $l_i = \texttt{select}(L, i), r_i = \texttt{select}(R, j)$. We will store this permutation P in a 2D range search data structure RS (for *range search*), with the points $(i, P[i])$ using $n \log n + o(n \log n)$ bits using Lemma 5. Given a rectangle using preorder numbers as its coordinates, we will need to convert it into rank space that is stored. For example, the range $[x_1, x_2]$ will need to be converted to $[r_1, r_2]$ where $r_1 = \texttt{rank}(L, x_1 - 1) + 1$ (rank of the first left endpoint in the range) and $r_2 = \texttt{rank}(L, x_2)$ (rank of the last left endpoint in the range). To convert back, given a rank r the corresponding preorder number is $\texttt{select}(L, r)$. To compute $P[v]$, we report the single point in the rectangle $[v, v] \times [-\infty, \infty]$ using $O(\log n / \log \log n)$ time, similarly for $P^{-1}[j]$.
- For every node a of the tree that is an apex, we store the point (a, z) (z being the last node in the subtree rooted at a) in a 2D 3-sided range reporting data structure $RSoA$ (for *range search on apex*) using Lemma 4. As we will need to support 2 different types of 3-sided queries, will need to store 2 such data structures.
- We wish to store the number of paths with a given apex, i.e. $|\texttt{apexlist}(a)|$. To do so, we store the bit-vector $AL = 10^{k_1}10^{k_2}\ldots$, where $k_i = |\texttt{apexlist}(\texttt{select}(A, i))|$, the number of paths with the i-th apex in pre-order. We note that since every path has exactly 1 apex, $\sum k_i = n$, and thus AL has length at most $2n$.
- We will store the tree T succinctly, using $8n + o(n)$ bits [5].
- We will store a semi-group path sum data structure using Lemma 7, with $\Delta = \frac{\log n}{\log \log n}$. The tree is the modified clique tree T, and the weight of each node is the number of paths with that node as apex: $|\texttt{apexlist}(a)|$. Since the total sum of the weights is n, we may use the semi-group integers modulo n under addition.

Therefore, the space required for all the data structures is $n \log n + o(n \log n)$ bits.

We now consider some internal operations for our data structure. First are the features of a path. Given a vertex v, we may compute its features as: $l_v = \texttt{select}(L, v)$, $r_v = \texttt{select}(R, P[v])$, $a_v = \texttt{LCA}(l_v, r_v)$ and $z_v = \texttt{last\_child}(a_v)$. Therefore it takes $O(1)$ time to compute l_v, but $O(\log n / \log \log n)$ time to compute r_v, a_v or z_v.

Implementing `apexlist`**:** Given an apex a, let $w_1, \ldots, w_k$ be the children of a and let $z_1, \ldots, z_k$ denote the last node in the subtree rooted at $w_1, \ldots, w_k$. A path with apex a has its two branches in different children of a. Thus we will capture this by the rectangles $[w_i, z_i] \times [w_{i+1}, z_k]$ in the range search data structure RS, which captures the fact that the left endpoint of the path is a descendant of w_i and the right endpoint

[5] This and the other linear terms can be optimized further, see the thesis [11]

is a descendant of one of $w_{i+1}, \ldots, w_k$. We stop once one of the rectangles is empty. By construction, as we have ordered the children of each apex, the non-empty rectangles will be at the start. This allows us to get each path in $O(\log n / \log\log n)$ time.

Implementing $|$`apexlist`$|$: Given a node a, we first check that a is an apex by checking that $A[a] = 1$. If a is not an apex, then $|\texttt{apexlist}(a)| = 0$. Otherwise, we may find its rank among apex by $\texttt{rank}(A, a)$. To find the number of paths with a as its apex, we find the indices in AL of a and the next apex in preorder - $\texttt{select}(AL, \texttt{rank}(A, a))$ and $\texttt{select}(AL, \texttt{rank}(A, a) + 1)$ respectively, and find the number of 0s bits between them. This takes $O(1)$ time.

Now we implement the navigational queries:

Implementing `adjacency` Query: Given vertices u, v, we obtain l_u, r_u, l_v, r_v, a_u, z_u, a_v, z_v, taking $O(\log n / \log\log n)$ time. To check adjacency, we have 3 cases:

Case 1: If a_u, a_v have no ancestor/descendant relationship, then u, v are not adjacent.

Case 2: If $a_u = a_v$, then they are adjacent.

Case 3: If a_u is an ancestor of a_v. Then u, v are adjacent iff exactly one of l_u and r_u is a descendant of a_v.

These can be checked in $O(1)$ time since a node w is a descendant of an apex a_u exactly when $a_u \leq w \leq z_u$.

Implementing `neighbourhood` Query: Given a vertex v, we consider the vertices u which fall into cases 2 and 3 above. For case 2, the vertices u with $a_u = a_v$ can be listed using $\texttt{apexlist}(a_v)$. For case 3, we distinguish between the two cases: whether a_u is an ancestor of a_v.

If a_u is an ancestor of a_v: we need to list out all paths u such that exactly one of l_u, r_u is a descendant of a_v. In RS on the points (l_u, r_u), such paths u are exactly those with points lying in either of the rectangles $(-\infty, a_v) \times (a_v, z_v)$ and $(a_v, z_v) \times (z_v, \infty)$.

If a_u is a descendant of a_v: we need list out all paths u such that exactly one of l_v, r_v is a descendant of a_u. In $RSoA$ on the points (a_u, z_u), such paths u are exactly those with points lying in either of the rectangles $(-\infty, l_v) \times (l_v, r_v)$ and $(l_v, r_v) \times (r_v, \infty)$. Since this returns a set of apexes, for each returned apex a, we list the paths with $\texttt{apexlist}(a)$.

Implementing `degree` Query: Given a vertex v, there are two types of neighbours: u with a_u an ancestor of a_v and those that are a_v or descendants. In the first case, we use range count. In the second, it is not hard to see that a_u is on the path P_v, and thus the number of paths can be found using our path-sum data structure.

Theorem 1. *Let G be a path graph. G can be represented using $n \log n + o(n \log n)$ bits to support `adjacency`, `degree` in $O(\log n / \log\log n)$ time and `neighbourhood` using $O(\log n / \log\log n)$ time per neighbour.*

To achieve $O(1)$ query times, we replace RS with several structures. We store P using Lemma 3, 3-sided queries using Lemma 4, its role in `apexlist` using an array storing the paths and `degree` using an array storing all the degrees. Details are omitted.

Theorem 2. *Let G be a path graph. G can be represented using $(3 + \varepsilon)n \log n + o(n \log n)$ bits to support `adjacency`, `degree` in $O(1)$ time and `neighbourhood` using $O(1)$ time per neighbour.*

4 Path Graph Distances

As path graphs are a subclass of chordal graphs, we will use the same technique as Munro and Wu [10]. We begin with a clique tree T of G. For a vertex v, we define the apex a_v of v to be the node with the smallest depth on the subtree corresponding to v. We create a distance tree T_D as follows: for each vertex v, define the parent of v in T_D as the vertex adjacent to v with the highest (smallest depth) apex - breaking ties arbitrarily. To be consistent, for all vertices with the apex at the root of T, choose one and set it as the parent of the other paths that have the root as their apex. Thus T_D has n nodes, one for each vertex. The distance or shortest path calculation presented by Munro and Wu [10] can be summarized as the following lemma:

Lemma 8. *Let G be a chordal graph, and T be a clique tree, and T_D be the distance tree. Let u, v be two vertices, and let a_u, a_v be apexes of u, v respectively. In the case that one of a_u, a_v is not the ancestor of the other,* [6] *let $h = \texttt{LCA}(a_u, a_v)$. Let u' be the ancestor of u (in T_D) of smallest depth such that $a_{u'}$ is a descendant of h, similarly for v'. Then the distance between u and v is: $\texttt{dist}(u, v) = \texttt{dist}(u, u') + \texttt{dist}(v, v') + 2 + \mathbf{1}(C)$. Where C is the condition: there does not exist any vertex w such that the subtree corresponding to w contains both the nodes $a_{u'}$ and $a_{v'}$.* [7]

In the path graph case, our vertices are associated with paths and not arbitrary subtrees. Thus the condition for a path w containing both $a_{u'}$ and $a_{v'}$ (with $a_{u'} < a_{v'}$) is that l_w is a descendant of $a_{u'}$ and r_w is a descendant of $a_{v'}$. Such a w would satisfy the rectangle $[a_{u'}, z_{u'}] \times [a_{v'}, z_{v'}]$ on the points (l_w, r_w) (i.e. the data structure RS).

As this is a four sided query, we will use our succinct data structure to support this.

Theorem 3 (Corollary to Theorem 1)**.** *In addition to theorem 1, we can answer the $\texttt{dist}$ query in $O(\frac{\log n}{\log \log n})$ time and the $\texttt{shortest\_path}$ query in $O(\frac{\log n}{\log \log n} + \texttt{dist})$ time, where $\texttt{dist}$ is the distance between the two vertices. The extra space required for these two queries is $(1 + \varepsilon)n \log n + O(n)$ bits.*

Proof. The extra space comes from storing T_D and a mapping between T_D and the vertices of the graph. To compute `dist` and `shortest_path` we need to decide C, which we can using the above rectangle. □

5 Chordal Graphs

Munro and Wu [10] modified the clique tree T (rooted arbitrarily) of a chordal graph G so that it contained exactly n nodes. For each vertex v, they associated it with the highest node (smallest depth) T_v of the subtree corresponding to v. Their modification

[6] The case that they are is equivalent to distances in interval graphs and is much easier

[7] u', v' can be found using `level-anc` on T_D and $\texttt{dist}(u, u')$ can be calculated as $\texttt{depth}_{T_D}(u) - \texttt{depth}_{T_D}(u')$, similarly for $\texttt{dist}(v, v')$. A shortest path can be found by taking the paths $u \to u'$, $v \to v'$ in T_D joined by either $u' \to w \to v'$ if w exists or $u' \to \texttt{parent}(T_D, u') \to \texttt{parent}(T_D, v') \to v'$ if not.

ensured that for any two vertices u, v, $T_u \neq T_v$, so v is also used refer to the node T_v for tree operations. For each vertex v, the set S_v is the set of vertices whose subtrees pass through v. By definition $v \in S_v$, and if $u \in S_v$ then the node T_v is a descendant of T_u. Again by definition, we obtain that $S_v \subseteq S_{\texttt{parent}(T,v)} \cup \{v\}$ as any subtree passing through v (except v) must also pass through $\texttt{parent}(T, v)$. We number the vertices v by the preorder number of T_v so that any $u \in S_v$ has $u \leq v$. Munro and Wu considered the following two operations: $\texttt{adjacency}(x, v)$: Given $x \in [n]$ and $v \in T$, is $x \in S_v$? $\texttt{decode}(v, i)$: Given $i \in [n]$ and $v \in T$, what is the i-th smallest element in S_v?

The first is aptly named `adjacency` because $u < v$ are adjacent iff $u \in S_v$. The second operation allows us to return the elements of S_v one at a time, which is needed in `neighbourhood`. Ultimately, Munro and Wu reduced all the queries to those two operations summarized by the following corollary:

Corollary 1. *Let G be a chordal graph. Let D be a data structure answering* ***adjacency*** *and* ***decode*** *in $O(f(n))$ time, which occupies $n^2/4 + o(n^2)$ bits of space. Then we may support* ***adjacency*** *in $O(f(n))$ time,* ***degree*** *in $O(1)$ time,* ***neighbourhood*** *in $O(f(n)g(n))$ time per neighbour for any function $g(n) \in \omega(1)$, and* ***dist***, ***shortest_path*** *in $O(nf(n))$ time. The space is succinct: $n^2/4 + o(n^2)$ bits.*

Munro and Wu [10] in their paper gave a way to construct D to support the queries with $f(n) \in \omega(1)$, and for simplicity, set $f = g$. In this section, we show how to construct D to support the queries with $f(n) = O(1)$.

Theorem 4. *Let G be a chordal graph. There is a succinct data structure occupying $n^2/4 + o(n^2)$ bits of space which supports* ***adjacency*** *in $O(1)$ time,* ***degree*** *in $O(1)$ time,* ***neighbourhood*** *in $O(g(n))$ time per neighbour for any function $g(n) \in \omega(1)$, and* ***dist***, ***shortest_path*** *in $O(n)$ time.*

Proof. (Sketch, full proof in [11]) For a vertex v, we say that v is enlarging if $S_v = S_{\texttt{parent}(T,v)} \cup \{v\}$. We select $O(\sqrt{n})$ shortcut nodes in T so that any path to the root has a shortcut node after at most $\sqrt{n}$ steps. For an enlarging node v, we store a pointer to its nearest non-enlarging ancestor v'. For a non-enlarging node v, we store a pointer to its nearest shortcut ancestor v'. By definition of enlarging, if $v_1, \ldots v_k$ are the nodes between v and v', then $S_v = S_{v'} \cup \{v_1, \ldots v_k, v\}$ if v is enlarging. If v is not, then $S_v \subseteq S_{v'} \cup \{v_1, \ldots, v_k, v\}$. Since there are $\sqrt{n}$ shortcut nodes, we may store the set at each explicitly. For non-enlarging nodes, we store the subset explicitly. For enlarging nodes, we store nothing. In either case, we can reduce the query to $S_{v'}$ if the queried element is not among $\{v_1, \ldots, v_k, v\}$. And $S_{v'}$ is either a shortcut node or is non-enlarging which then reduces to a shortcut node. An analysis of the space used similar to [10] gives $n^2/4 + o(n^2)$ bits. □

6 References

[1] Fernando Magno Quintão Pereira and Jens Palsberg, "Register allocation via coloring of chordal graphs," in *Programming Languages and Systems*, Kwangkeun Yi, Ed., Berlin, Heidelberg, 2005, pp. 315–329, Springer Berlin Heidelberg.

[2] Ronald Fagin, "Degrees of acyclicity for hypergraphs and relational database schemes," *J. ACM*, vol. 30, no. 3, pp. 514–550, jul 1983.

[3] Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti, "Succinct encodings for families of interval graphs," *Algorithmica*, vol. 83, no. 3, pp. 776–794, 2021.

[4] 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 *31st International Symposium on Algorithms and Computation, ISAAC 2020,*. 2020, vol. 181 of *LIPIcs*, pp. 25:1–25:18, Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[5] Sankardeep Chakraborty and Seungbum Jo, "Compact representation of interval graphs of bounded degree and chromatic number," in *Data Compression Conference, DCC 2022, Snowbird, UT, USA, March 22-25, 2022*. 2022, pp. 103–112, IEEE.

[6] Arash Farzan and Shahin Kamali, "Compact navigation and distance oracles for graphs with small treewidth," *Algorithmica*, vol. 69, no. 1, pp. 92–116, 2014.

[7] Sankardeep Chakraborty, Seungbum Jo, Kunihiko Sadakane, and Srinivasa Rao Satti, "Succinct data structures for series-parallel, block-cactus and 3-leaf power graphs," in *Combinatorial Optimization and Applications, COCOA 2021*. 2021, vol. 13135 of *Lecture Notes in Computer Science*, pp. 416–430, Springer.

[8] Konstantinos Tsakalidis, Sebastian Wild, and Viktor Zamaraev, "Succinct permutation graphs," *Algorithmica*, vol. 85, no. 2, pp. 509–543, 2023.

[9] Girish Balakrishnan, Sankardeep Chakraborty, N.S. Narayanaswamy, and Kunihiko Sadakane, "Succinct data structure for path graphs," *Information and Computation*, vol. 296, pp. 105124, 2024.

[10] J. Ian Munro and Kaiyu Wu, "Succinct data structures for chordal graphs," in *29th International Symposium on Algorithms and Computation, ISAAC 2018*. 2018, vol. 123 of *LIPIcs*, pp. 67:1–67:12, Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[11] Kaiyu Wu, *Succinct and Compact Data Structures for Intersection Graphs*, Ph.D. thesis, University of Waterloo, 2023.

[12] Paul Erdös, A. W. Goodman, and Louis Pósa, "The representation of a graph by set intersections," *Canadian Journal of Mathematics*, vol. 18, pp. 106–112, 1966.

[13] James R. Walter, "Representations of chordal graphs as subtrees of a tree," *Journal of Graph Theory*, vol. 2, no. 3, pp. 265–267, 1978.

[14] Fănică Gavril, "A recognition algorithm for the intersection graphs of paths in trees," *Discrete Mathematics*, vol. 23, no. 3, pp. 211–227, 1978.

[15] C Lekkeikerker and Johan Boland, "Representation of a finite graph by a set of intervals on the real line," *Fundamenta Mathematicae*, vol. 51, pp. 45–64, 1962.

[16] Delbert Ray Fulkerson and Oliver Alfred Gross, "Incidence matrices and interval graphs," *Pacific Journal of Mathematics*, vol. 15, pp. 835–855, 1965.

[17] Fănică Gavril, "The intersection graphs of subtrees in trees are exactly the chordal graphs," *Journal of Combinatorial Theory, Series B*, vol. 16, no. 1, pp. 47–56, 1974.

[18] Clark, David, *Compact PAT trees*, Ph.D. thesis, University of Waterloo, 1997.

[19] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao S., "Succinct representations of permutations and functions," *Theoretical Computer Science*, vol. 438, pp. 74–88, 2012.

[20] Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin, "Succinct orthogonal range search structures on a grid with applications to text indexing," in *WADS*, 2009.

[21] Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou, "Succinct indices for path minimum, with applications," *Algorithmica*, vol. 78, no. 2, pp. 453–491, jun 2017.