Succinct Data Structure for Path Graphs

Girish Balakrishnan[*], Sankardeep Chakraborty[†], N S Narayanaswamy[*],
and Kunihiko Sadakane[†]

[*]Indian Institute of Technology Madras,
Chennai, India
girishb@cse.iitm.ac.in
swamy@cse.iitm.ac.in

[†]University of Tokyo,
Tokyo, Japan
sankardeep.chakraborty@gmail.com
sada@mist.i.u-tokyo.ac.jp

Abstract

We consider the problem of designing a succinct data structure for *path graphs* (which are a proper subclass of chordal graphs and a proper superclass of interval graphs) on n vertices while supporting degree, adjacency, and neighborhood queries efficiently. We provide the following two solutions for this problem:

1. an $n \log n + o(n \log n)$-bit succinct data structure that supports adjacency query in $O(\log n)$ time, neighborhood query in $O(d \log n)$ time and finally, degree query in $\min\{O(\log^2 n), O(d \log n)\}$ where d is the degree of the queried vertex.

2. an $O(n \log^2 n)$-bit space-efficient data structure that supports adjacency and degree queries in $O(1)$ time, and the neighborhood query in $O(d)$ time where d is the degree of the queried vertex.

Central to our data structures is the usage of the classical heavy path decomposition by Sleator and Tarjan [1], followed by a careful bookkeeping using an orthogonal range search data structure using wavelet trees [2] among others, which maybe of independent interest for designing succinct data structures for other graph classes.

1 Introduction

An *intersection graph* $G = (V, E)$ is an undirected graph whose vertices are mapped by f to a family of sets F such that vertex a_1 is adjacent to a_2 in G if and only if $f(a_1) \cap f(a_2) \neq \phi$. Based on the family of sets F we get different graph classes. For instance, if F is the set of intervals on the real number line, then we get *interval graphs*. Yet another example is *chordal graphs*, defined as the intersection graph of sub-trees of a tree. *Path graphs* is the class of graphs obtained when F is the set of paths, $P_1, \ldots, P_n$ in a tree T such that two paths intersect if and only if the corresponding vertices are adjacent. It is well-known that the class of path graphs is a proper subclass of chordal graphs and a proper superclass of interval graphs [3].

In this work, we address the problem of designing a succinct data structure for the class of path graphs so that basic navigational queries such as degree, adjacency, and neighborhood can be answered efficiently. Formally, given a set T consisting of combinatorial objects with a certain property, our goal is to store any arbitrary member $x \in T$ using the information-theoretic minimum of $\log(|T|) + o(\log(|T|))$ bits (throughout this paper, log denotes the logarithm to the base 2) while still being able to support the queries efficiently on x. Recently, Acan et al. [4] showed

that the information-theoretic lower bound for representing unlabeled interval graphs with n vertices is at least $n \log n$ bits, and as path graphs are a proper superclass of interval graphs, this lower bound also holds true for path graphs. Interestingly, we manage to construct an $n \log n + o(n \log n)$-bit data structure for representing path graphs matching this information-theoretic lower bound, thus, obtaining succinct data structure for path graphs for the first time in literature. This is the main contribution of this work. We leave the question of whether path graphs are only a constant factor larger in size than the class of interval graphs as an open problem.

Previous Related Work. There already exists a huge body of work on representing various classes of graphs succinctly. A partial list of such special graph classes would be trees [5,6], planar graphs [7], partial k-tree [8], and arbitrary graphs [9]. Recent results have appeared in literature for intersection graphs like interval graphs due to Acan et al [4] and chordal graphs due to Munro and Wu[10]. For interval graphs, [4] gives an $n \log n + O(n)$ bit succinct data structure that supports degree and adjacency queries in $O(1)$ time while neighborhood query in constant time per neighbour. In the case of chordal graphs, [10] gives an $n^2/4 + o(n^2)$ bit succinct data structure that supports adjacency query in $f(n)$ time where $f(n) \in \omega(1)$, degree of a vertex in $O(1)$ time and neighborhood in $(f(n))^2$ time per neighbour. The main motivation behind our work stems from these two above-mentioned works. Since path graphs is a strict subclass of chordal graphs and a strict superclass of interval graphs it would be interesting to see whether one can design such an efficient data structure for path graphs as well.

Our Results. Before we get to our results, note the following terminology for graph $G = (V, E)$:

- for $u, v \in V$, adjacency query checks if $\{u, v\} \in E$,

- for $u \in V$, the neighborhood query returns all the vertices that are adjacent to u in G, and

- for $u \in V$, the degree query returns the number of vertices adjacent to u in G.

Our primary result in this work is an $n \log n + o(n \log n)$-bit succinct representation for unlabelled connected path graphs. It is obtained from the clique tree representation $(T, P_1, \ldots, P_n)$ [11] [12] on the input path graph. Here T is the clique tree [12] and $P_i, 1 \leq i \leq n$, are the paths in it. We then store T succinctly along with the end-points of the paths P_i in it. Formally we have the following result.

Theorem 1. *Path graphs have an $n \log n + o(n \log n)$-bit succinct representation. The succinct representation constructed from the clique tree representation supports for a vertex u the following queries:*

1. *adjacency query in $O(\log n)$ time,*
2. *the neighborhood query in $O(d_u \log n)$ time, and*
3. *the degree query in $\min\{O(\log^2 n), O(d_u \log n)\}$ time*

where d_u is the degree of vertex u.

The central tool that we use in obtaining the above succinct data structure result and the space-efficient data structure is heavy path decomposition (HPD) [1] performed on the clique tree T. The HPD when performed on the clique tree T gives the heavy path tree $\mathcal{T}$; explained in Section 2.2. Each node of the heavy path tree $\mathcal{T}$ corresponds to a heavy path of the clique tree T. The property that heavy path tree $\mathcal{T}$ has at most $\lceil \log n \rceil$ levels helps us achieve the query times of Theorem 1. Additionally, we observe that the intersection of the paths $P_1, \ldots, P_n$ with each heavy path defines a natural interval graph giving us the space-efficient data structure for path graphs. Further we observe that the union of these interval graphs corresponding to nodes in the same level of the heavy path tree is also an interval graph. To obtain the space-efficient data structure, we store the interval graphs at each level of the heavy path tree using the results from [4] and organize them into at most $\log n$ levels. Even though we use additional $\log n$ factor storage in the space-efficient data structure over the succinct representation, we can respond to all the queries more efficiently. This is our second result.

Theorem 2. *There exists a space-efficient representation for path graphs using $O(n \log^2 n)$ bits. The representation supports the following queries for a vertex u:*

1. *the adjacency and degree queries in $O(1)$ time,*
2. *the neighborhood query in $O(d_u)$ time where d_u is the degree of the vertex u.*

The increased efficiency of the space-efficient data structure comes at the expense of increased space which arises due to the duplication of edges of path graph among the $\log n$ interval graphs. Another difference is that the succinct data structure performs orthogonal range search to implement the queries while the space-efficient data structure delegates the queries to those of the underlying interval graph as implemented in [4].

All the preliminary terminology and concepts required for rest of the sections are in Section 2 and 3. In Section 4, a succinct representation for path graphs is presented and Section 5 describes the space-efficient data structure.

2   Preliminaries

For a graph G, through out the paper we denote the set of vertices and edges by $V(G)$ and $E(G)$, respectively. Familiarity with basic graph theory as in [13] and graph algorithms as given in [14] is expected.

2.1   Path Graphs and Its Properties

A graph G is a *path graph* if there exists a tree T and family of paths $\mathcal{P} = \{P_1, \ldots, P_n\}$ in T such that G is the intersection graph of paths in $\mathcal{P}$. G is said to have the representation $(T, \mathcal{P})$; see Figure 1. A vertex $a \in V(G)$ is *simplicial* if the set of vertices adjacent to a, denoted $N(a)$, induces a complete sub-graph of G [3]. The ordering $\rho = [a_1, \ldots, a_n]$ of $V(G)$ is called a *perfect elimination scheme* if for all $i, X_i = \{a_j \in N(a_i) \mid j > i,\}$ is complete. Every path graph has a simplicial

vertex and a perfect elimination scheme. It is well known that any simplicial vertex can start a perfect elimination scheme; see Theorem 4.1 and Lemma 4.2 of [3] for more details. Let $\mathcal{C}$ be the set of maximal cliques of G and for every $a \in V(G)$ let $\mathcal{C}_a = \{C | C \in \mathcal{C}$ and $a \in V(C)\}$. Consider a tree T with $V(T) = \mathcal{C}$ such that for every $a \in V(G)$, $\mathcal{C}_a$ induces a sub-tree T_a of T. G is a choral graph if it is the intersection graph of set of such induced sub-trees. For chordal graphs such a tree T is called the *clique tree* of G [3]. Clique tree can be computed in polynomial time [11]. The following is a characterisation of path graphs as a sub-class of chordal graphs [11][12].

Figure 1: (a) Path graph G with maximal cliques $\mathcal{C} = \{C_1, \ldots, C_8\}$, (b) Clique tree representation of G where each node i corresponds to $C_i \in \mathcal{C}$. For vertices $u, v \in V(G)$ we have paths P_u, P_v such that $V(P_u) \cap V(P_v) \neq \phi$ if and only if $\{u, v\} \in E(G)$. The clique tree shown here is pre-processed as explained in Section 3.

Theorem 3. *The graph G is a path graph if and only if there exists a clique tree T, such that for every $v \in V(G)$, the set of maximal cliques containing v form a path in T.*

In this paper, we are concerned with the construction of a succinct representation for path graphs and the construction mechanism takes as input, $(T, \mathcal{P})$. Also, apart from clique tree T we will introduce the heavy path tree $\mathcal{T}$ in the next section. Elements of $V(T)$ and $V(\mathcal{T})$ will be henceforth referred to as nodes of T and $\mathcal{T}$, respectively whereas for any graph G, elements of $V(G)$ will be referred to as its vertices. The following is known from [11].

Remark 4. *The number of maximal cliques in a path graph G with n vertices is at most n.*

2.2 Heavy Path Decomposition

Heavy path decomposition (HPD) was introduced in [1] and used in [15] and [16] for rooted trees. In the heavy path decomposition for a rooted tree T, each internal node u selects an edge (u, v) such that the child v has the maximum number of descendants among the children of u. In the case of a tie among two children of u pick any one arbitrarily. The edge (u, v) is called a *heavy edge*. Thus, each internal node of T

selects exactly one edge as heavy edge. Further, it is known that each vertex has at most two heavy edges incident on it, one with its parent and second with one of its children. An edge that is not chosen as a heavy edge by any internal node is called a *light edge*. Consider the forest of paths obtained by removing light edges from T. We refer to each path in this forest as a *heavy path*. The heavy path decomposition of T partitions the nodes of T into the set of heavy paths denoted by $\mathcal{H}$. Also, it partitions the edges of T into heavy and light edges; see Figure 2.

Heavy Path Decomposition

Heavy Path Tree

Figure 2: Heavy path decomposition of a rooted tree with $n = 12$ nodes. At node 1, $\{1, 2\}$ is picked as the heavy edge as 2 is the child of 1 with maximum number of descendants. Edge $\{1, 12\}$ is a light edge. Path $\{1, 2, 3, 4, 5\}$ of the clique tree is compressed as a single node in the heavy path tree $\mathcal{T}$. As a light edge connects a node to a sub-tree that is at least halved in size, the heavy path tree will have at most $\log n$ levels. Thick red lines indicate heavy edges and thin blue lines indicate light edges respectively.

Remark 5. *The heavy paths in $\mathcal{H}$ are of two types: those that contain at least one heavy edge and those which do not contain a heavy edge. A heavy path which does not contain a heavy edge is a leaf node whose incident edge in T is a light edge. For instance, in Figure 2 heavy path $\{1, 2, 3, 4, 5\}$ is of former type whereas heavy path $\{6\}$ is of later type.*

Using the heavy path decomposition of T we define a tree which we call the *heavy path tree* denoted by $\mathcal{T}$. Let $\Phi : \mathcal{H} \to V(\mathcal{T})$ be a bijection such that for $u \in V(H_1)$ and $v \in V(H_2), e = \{u, v\}$ is a light edge if and only if there exists an edge $\{\Phi(H_1), \Phi(H_2)\}$ in $\mathcal{T}$. Subsequently, whenever H_1 and H_2 satisfy this property we will call H_1 and H_2 *light edge separable* heavy paths. In other words, $H_1, H_2 \in \mathcal{H}$ are light edge separable heavy paths in T if and only if $\Phi(H_1)$ and $\Phi(H_2)$ are adjacent in $\mathcal{T}$. Further, we refer to the edge $\{\Phi(H_1), \Phi(H_2)\} \in E(\mathcal{T})$ using the light edge between and H_1 and H_2, which in this case is e. The heavy path that contains the root of T is the root of $\mathcal{T}$.

The level of the root node is 0, and each other node has a level which is its distance from the root. A sub-path of a heavy path will be referred to as *heavy sub-path*. The following remark is important for the rest of the paper. The following lemmata are well-known [15], and we extensively use them here.

Lemma 6. *The number of levels in $\mathcal{T}$ is at most $\lceil \log n \rceil$. Further, a path in $\mathcal{T}$ has at most $2\lceil \log n \rceil$ edges.*

Lemma 7. *For $u, v \in V(\mathcal{T}), V(\Phi^{-1}(v)) \cap V(\Phi^{-1}(u)) = \phi$ and $V(T) = \bigcup_{v \in V(\mathcal{T})} V(\Phi^{-1}(v))$. In other words, the nodes of T are partitioned among the nodes of $\mathcal{T}$.*

Lemma 8. *Let P be a path in T. P can be partitioned into heavy sub-paths $\Pi = \{\pi_1, \ldots, \pi_k\}, 1 \leq k \leq 2\lceil \log n \rceil + 1.$*

Proof. Consider a path $P = v_1 e_2 v_2 e_2 \ldots v_{l-1} e_l v_l, 1 \leq l \leq n$ in T where $v_1, \ldots, v_l \in V(P)$ and $e_2, \ldots, e_l \in E(P)$. Let $f_1, \ldots, f_t$ denote the $t \geq 0$ light edges in P indexed in the order in which they occur in P from v_1 to v_l. Then, we consider $P = \pi_1 f_1 \pi_2 f_2 \ldots f_t \pi_{t+1}$ such that for each $1 \leq i \leq t + 1$, π_i is a maximal heavy sub-path in P. f_i is the light edge between the last vertex of π_i and the first vertex of π_{i+1}. For each $1 \leq i \leq t + 1$, let H_i denote the heavy path in $\mathcal{H}$ such that π_i is a heavy sub-path of H_i. By our convention on the edge label in $\mathcal{T}$, the edge $\{\Phi(H_i), \Phi(H_{i+1})\}$ is considered to be f_i. Thus, for P in T we have a path $P' = \Phi(H_1) f_1 \Phi(H_2) f_2 \ldots f_t \Phi(H_{t+1})$ in $\mathcal{T}$. Let $|V(P')| = t + 1$ and from Lemma 6 we know that $t \leq 2\lceil \log n \rceil$. Thus, the number of heavy sub-paths, $k \leq 2\lceil \log n \rceil + 1$. $\square$

The following propositions are straightforward and merely stated explicitly in the context of heavy path trees.

Proposition 9. *Let P be a path in $\mathcal{T}$ and $l \leq \log n$ be an integer. P consists of at most two nodes with level l.*

Through out this paper we will use the notation $[n] = \{1, 2, \ldots, n\}$.

2.3 Useful Succinct Data Structures

Table 1 summarises the set of data structures we use in this work which we will explain in this section starting with ordinal trees.
Succinct Data Structure for Ordinal Trees. Let the children of any $u \in V(T)$ be $\{u_1, \ldots, u_z\}$ for some $z > 0$. Tree T is called an *ordinal tree* if for $i < j$, u_i is to the left of u_j [17]. By considering ordinal trees as balanced parenthesis Navarro and Sadakane [17] has given a $2n + o(n)$ bit succinct data structure.

Lemma 10. *For any ordinal tree T with n nodes, there exists a $2n + o(n)$ bit Balanced Parentheses (BP) based data structure that supports the following four functions among others in constant time :*

1. $\mathtt{lca}(i, j)$, *returns the lowest common ancestor of two nodes i, j in T,*
2. $\mathtt{parent}(i)$, *returns the parent of node i in T, and*

3. `first_child`(i), *returns the first child of node i in T.*
4. `rmost_leaf`(i), *returns the rightmost leaf of sub-tree rooted at node i in T.*
5. `child_rank`(i), *returns the number of siblings to the left of node i in T.*

Rank-Select Data Structure. Bit-vectors are extensively used in the succinct representation given in Section 4. The following data structure due to Golynski et al. [20] and the functions supported by it are useful.

Lemma 11. *Let B be an $n-$bit vector and $b \in \{0,1\}$. There exists an $n + o(n)$ bit data structure that supports the following functions in constant time:*

1. `rank`(B, b, i): *Returns the number of b's up to and including position i in the bit vector B from the left.*
2. `select`(B, b, i): *Returns the position of the i-th b in the bit vector B from left. For $i \notin [n]$ it returns 0.*

Non-decreasing Integer Sequence Data Structure. Given a set of positive integers in the non-decreasing order we can store them efficiently using the differential encoding scheme for increasing numbers; see Section 2.8 of [19]. Let S be the data structure that supports differential encoding for increasing numbers then the function `accessNS`(S, i) returns the $i-$th number in the sequence.

Lemma 12. *Let S be a sequence of n non-decreasing positive integers $a_1, \ldots, a_n, 1 \leq a_i \leq n$. There exists a $2n + o(n)$ bit data structure that supports `accessNS`(S, i) in constant time.*

Proof. We will prove the lemma by giving a construction of such a data structure. a_1 will be represented by a sequence of a_1 1's followed by a 0. Subsequently a_i's are represented by storing $a_i - a_{i-1}$ many 1's followed by a 0. It will take $2n$ bits since there are n 0's and n 1's. Let this bit string be stored using the data structure of Lemma 11 and be denoted as B. B takes $2n + o(n)$ bits. `accessNS`(S, i) can be implemented using `rank`$(B, 1, \text{select}(B, 0, i))$ on the bit string obtained. $\square$

Wavelet Trees. Central to the design of the succinct data structure of Section 4 is the two-dimensional orthogonal range search data structure used to store points in the two-dimensional plane. Specifically, we use the $n \log n + o(n \log n)$ bit succinct *wavelet trees* due to Makinen and Navarro [2] that requires the n points to have distinct integer-valued $x-$ and $y-$coordinates in the range $[n] \times [n]$. The wavelet tree has the following properties.

1. The wavelet tree is a balanced binary search tree. Each node of the tree is associated with an interval range of the x-coordinate.

2. The range at the root of the wavelet tree has the interval $[1, n]$ and the interval at each leaf is of the form $[a, a], 1 \leq a \leq n$.

3. The range $[a, a'], 1 \leq a < a' \leq n$, at an internal node is partitioned among the ranges $[a, a'']$ and $[a'' + 1, a']$ at its children, that is, $[a, a'] = [a, a''] \bigcup [a'' + 1, a']$.

We use the following result regarding wavelet trees from [2].

Lemma 13. *Given a set of n points $\{(a_1, b_1), \ldots, (a_n, b_n)\}$ where $a_i, b_i \in [n], 1 \leq i \leq n$ such that $a_i \neq a_j$ and $b_i \neq b_j$ for $i \neq j$, there exists an $n \log n(1 + o(1))$ bit orthogonal range search data structure S that supports the following functions:*

1. $\texttt{search}(S, [i, i'], [j, j'])$: *Returns the points in the input range $[i, i'] \times [j, j'], 1 \leq i \leq i' \leq n, 1 \leq j \leq j' \leq n$, in the increasing order of the $x-$coordinate taking $O(\log n)$ time per point.*
2. $\texttt{count}(S, [i, i'], [j, j'])$: *Returns the number of points in the input range $[i, i'] \times [j, j'], 1 \leq i \leq i' \leq n, 1 \leq j \leq j' \leq n$, in $O(\log n)$ time.*
3. $\texttt{access}(S, i)$: *Returns the $y-$coordinate of the point stored in S with $x-$coordinate i in $O(\log n)$ time.*

3   Clique Tree Pre-processing = HPD + Pre-order Traversal

The pre-processing of the clique tree allows the paths in $\mathcal{P}$ to be stored efficiently so that adjacency and neighbourhood queries can be supported. This involves the following two steps:

1. heavy path decomposition of T, and

2. transformation of T into an ordinal tree which is labeled based on the pre-order traversal

A clique tree is organized as an ordinal tree as explained below.

HPD + Pre-order traversal of T. Fix a root node for T and perform heavy path decomposition on it. For $v \in V(T)$ order its children $(w_1, \ldots, w_c)$ such that $\{v, w_1\}$ is a heavy edge. Let the children adjacent to v by light edges $(w_2, \ldots, w_c)$, be ordered arbitrarily. This ordering of children of a node of the clique tree makes it an ordinal tree. Label the nodes of this ordinal tree based on the pre-order traversal; see Section 12.1 of [14] for more details of pre-order traversal of trees. Labels assigned to nodes in this manner are called the *pre-order label of the nodes*. Through out the rest of our paper this ordinal rooted clique tree labeled with pre-order will be referred to as the clique tree.

Representing paths as tuples. Path $P \in \mathcal{P}$ is represented as $P = (l, r), l, r \in V(T)$, where l and r are the end points of P such that $l \leq r$. We say that l and r are the starting and ending nodes of the path, respectively. Let T_a and T_b be the sub-trees of T rooted at a and b, respectively. For $e = \texttt{lca}(l, r)$, the following propositions regarding the clique tree T follow from pre-order traversal.

Proposition 14. *Let $b \in V(T_a)$ for $a \in V(T)$. The following hold.*

1. *The sub-tree rooted at b is contained in T_a.*
2. *Let $V(P) \nsubseteq V(T_a)$. $|V(P) \cap V(T_a)| \geq 1$ if and only if $a \in V(P)$.*
3. *$l \in V(T_a)$ and $e < a$ if and only if $r \notin V(T_a)$ and $a \in V(P)$.*
4. *If $l \in V(T_a)$ and $e = a$ then $r \in V(T_a)$.*
5. *If there are z descendants of a in T_a then the nodes of $V(T_a)$ is the set $\{a, a+1, \ldots, a+z\}$.*

Proposition 15. *Let π be a heavy path of length $k \geq 1$ in T with a and b as its end points such that $a \leq b$. Then $b = a + k$.*

We will subsequently use the notation $[a, b]$ to refer to the ordered set of nodes $(a, a + 1, \ldots, b - 1, b)$. a and b are referred to as the starting and ending points of $[a, b]$. Figure 1 shows the pre-processed clique tree for the example path graph G, also shown in the figure. The heavy path starting at 1 and ending at 4 have contiguous numbering and is denoted as $[1, 4]$. We emphasize that a sub-path of a heavy path is also represented using the same notation. A heavy path or a heavy sub-path $[a, a]$ contains only the vertex a.

Lemma 16. *For any heavy path $H = [a', b']$ of T a path $\pi = [a, b]$ such that $a' \leq a \leq b \leq b'$ is a heavy sub-path of H. For the heavy sub-path π of H let c and d be the rightmost leaves in the sub-tree rooted at b and a, respectively. The following are true about π.*

1. *If $a \leq u \leq b$ then $u \in V(\pi)$.*
2. *$a \leq b \leq c \leq d$.*
3. *Let $Q = (s, t)$ be a path in T. If $s > d$ then π and Q are vertex disjoint paths in T.*

Proof. The proofs are as follows:

1. This is true due to Proposition 15.
2. $a \leq b$ by definition of π. $b \leq c \leq d$ since the labels are based on pre-order traversal.
3. From Proposition 14 we know that, if there are z descendants of a in T_a, then the nodes of $V(T_a)$ is the set $\{a, a + 1, \ldots, a + z\}$. Since d is the label of the rightmost descendant of a, it follows that $d = a + z$. Since $a + z = d < s$, we know that Q starts at a node that is visited after the nodes of T_a. Since $t > s$ the pre-order labels of nodes in Q is not in $V(T_a)$. Thus, Q and T_a are vertex disjoint, and thus Q and π are vertex disjoint. ☐

Lemma 17. *Let $a \in V(T)$ and a_l be a child of a such that $l \in V(T_{a_l})$.*

1. *If $\mathtt{lca}(l, r) < a$ then $r \in [\mathtt{rmost\_leaf}(a) + 1, n]$.*
2. *If $\mathtt{lca}(l, r) = a$ then $r \in [\mathtt{rmost\_leaf}(a_l) + 1, \mathtt{rmost\_leaf}(a)]$.*

Proof. The proof is as follows:

1. Since $\mathtt{lca}(l, r) < a$ and $r \geq l$, r is a node that is visited after the nodes in T_a are visited, that is, $r \in [\mathtt{rmost\_leaf}(a) + 1, n]$.
2. If $\mathtt{lca}(l, r) = a$ then there exists a child of a, say a_r such that $r \in T_{a_r}$. $a_l < a_r$ since $l < r$. Thus, $r \in [\mathtt{rmost\_leaf}(a_l) + 1, \mathtt{rmost\_leaf}(a)]$. ☐

3.1 Organizing the heavy paths and light edges of T

Let $\mathcal{H}$ be the set of heavy paths of T; recall from Section 2.2. Let $H, H' \in \mathcal{H}$ such that $H = [a, b]$ and $H' = [a', b']$. We define a total order $(\mathcal{H}, \prec_{\mathcal{H}})$ as follows. $H \prec_{\mathcal{H}} H'$ if $a < a'$. In other words, $H \prec_{\mathcal{H}} H'$ if H is visited before H' in the pre-order traversal of T.

Total order on the heavy sub-paths of paths in T. $\prec_{\mathcal{H}}$, extends to the set of heavy sub-paths of a path. Let $\Pi = \{\pi_1, \ldots, \pi_k\}, 1 \leq k \leq 2\lceil \log n \rceil + 1$, be the set of heavy sub-paths of path P; see Lemma 8 for details regarding heavy sub-paths of a path. For any two $\pi, \pi' \in \Pi$, let $H, H' \in \mathcal{H}$ be such that π and π' are heavy sub-paths of H and H', respectively. $\pi \prec \pi'$ if $H \prec_{\mathcal{H}} H'$. In other words, we order the heavy sub-paths according to the order of the heavy paths that contain it.

Convention: In the rest of this section, P denotes the path (l, r) in T, and Π is the decomposition of P into heavy sub-paths. In other words, for the path P, $\Pi = (\pi_1, \ldots, \pi_k), 1 \leq k \leq 2\lceil \log n \rceil + 1, \pi_i \prec \pi_j$ if $1 \leq i < j \leq k$. Also, for every $\pi_i \in \Pi$, $\pi_i = (a_i, b_i)$. In Section 5, we assume that the heavy paths of T are numbered such that $H_i \prec_{\mathcal{H}} H_j$ if and only if $1 \leq i < j \leq n$.

3.2 Characterising path intersections in T

In this section, first we will show that a heavy sub-path $\pi \in \Pi$ partitions the nodes of T into four ranges of pre-order labels. Paths intersecting π are characterised based on these ranges. Adjacency and neighbourhood queries for π are implemented using orthogonal range search queries that use these ranges.

Successor of a heavy sub-path. For $\pi_i, \pi_j \in \Pi$, if there exists nodes $u_1 \in V(\pi_i)$ and $u_2 \in V(\pi_j)$ such that $\{u_1, u_2\}$ is a light edge in T then we say that π_i and π_j are *light edge separable* heavy sub-paths; an extension of the notion of light edge separable heavy paths from Section 2.2. Let $1 \leq i < j \leq k, \pi_i, \pi_j \in \Pi$. π_j is called the *successor* of π_i if $\pi_i \prec \pi_j$ and they are light edge separable. We define a mapping $\mathtt{succ} : \Pi \times \{1, 2\} \rightarrow \Pi \cup \{\mathtt{NULL}\}$ from a heavy sub-path of P to its successors defined as follows.

1. Successors for π_1: We have the following sub-cases depending on the number of heavy sub-paths of P.

   (a) $k = 1 : \pi_1$ has no successors, that is, $\mathtt{succ}(\pi_1, 1) = \mathtt{succ}(\pi_1, 2) = \mathtt{NULL}$.

   (b) $k > 1 :$ There are two cases:

      i. π_1 has both its successors that is $\mathtt{succ}(\pi_1, 1) = \pi_2 \neq \mathtt{NULL}$ and $\mathtt{succ}(\pi_1, 2) = \pi_z \neq \mathtt{NULL}$ for $z \in [3, k]$. This happens when P can be divided into two sub-paths $P^1 = (e, l)$ and $P^2 = (e, r)$ such that $e \neq l \neq r$.

      ii. π_1 has only one successor. Let $\pi_2 = (a_2, b_2)$ and $a_1 \neq b_1$. There are two sub-cases.

         A. $\mathtt{succ}(\pi_1, 1) = \mathtt{NULL}$ and $\mathtt{succ}(\pi_1, 2) = \pi_2$. This happens when $\mathtt{parent}(a_2) = a_1$.

B. $\texttt{succ}(\pi_1, 1) = \pi_2$ and $\texttt{succ}(\pi_1, 2) = \texttt{NULL}$. This happens when $\texttt{parent}(a_2) = b_1$.

When $a_1 = b_1$, we define $\texttt{succ}(\pi_1, 1) = \pi_2$ and $\texttt{succ}(\pi_1, 2) = \texttt{NULL}$.

2. Successors for $\pi_i, i \neq 1$: $\texttt{succ}(\pi_i, 1) = \pi_{i+1}$ if $\pi_{i+1} \in \Pi$ and π_i, π_{i+1} are light edge separable else $\texttt{succ}(\pi_i, 1) = \texttt{NULL}$. For all $i \neq 1, \texttt{succ}(\pi_i, 2) = \texttt{NULL}$.

Note that if $\texttt{succ}(\pi_1, 2) = \pi_z \neq \texttt{NULL}$ for $z \in [3, k]$ then $\texttt{succ}(\pi_{z-1}, 1) = \texttt{succ}(\pi_{z-1}, 2) = \texttt{NULL}$. Also, $\texttt{succ}(\pi_k, 1) = \texttt{succ}(\pi_k, 2) = \texttt{NULL}$.

Interval ranges associated with $\pi_i \in \Pi, 1 \leq i \leq k$. For $u \in V(T), \texttt{rmost\_leaf}(u)+1$ is the node that is visited immediately after traversing the nodes in sub-tree rooted at u in the pre-order traversal of T. We associate four ranges of nodes of T with π_i. The four ranges associated with π_i denoted by $R_j(i), 1 \leq j \leq 4$, are as follows.

i. $R_1(i)$: Range of nodes visited before a_i in the pre-order traversal of T. If $i = 1$ and $a_1 > 1$ then $R_1(1) = [1, a_1 - 1]$ else $R_1(1) = \phi$.

ii. $R_2(i)$: Heavy sub-path π_i, $[a_i, b_i]$.

iii. $R_3(i)$: There are two cases depending on existence of $\texttt{succ}(\pi_i, 1)$.

(a) If $\texttt{succ}(\pi_i, 1) \neq \texttt{NULL}$ then $R_3(i) = R_3^1(i) \cup R_3^2(i)$ where

i. $R_3^1(i)$: Range of nodes visited after b_i and before the nodes of $T_{a_{i+1}}$, $R_3^1(i) = [b_i + 1, a_{i+1} - 1]$.

ii. $R_3^2(i)$: Range of nodes visited after visiting the nodes of $T_{a_{i+1}}$ and before the right-most leaf of T_{b_i}. If $\texttt{rmost\_leaf}(a_{i+1}) \neq \texttt{rmost\_leaf}(b_i)$ then $R_3^2(i) = [\texttt{rmost\_leaf}(a_{i+1}) + 1, \texttt{rmost\_leaf}(b_i)]$ else $R_3^2(i) = \phi$.

(b) If $\texttt{succ}(\pi_i, 1) = \texttt{NULL}$ and $b_i \neq \texttt{rmost\_leaf}(b_i)$ then $R_3(i) = [b_i+1, \texttt{rmost\_leaf}(b_i)]$ else $R_3(i) = \phi$. Note that $b_i \neq \texttt{rmost\_leaf}(b_i)$ means that b_i is not a leaf.

iv. $R_4(i)$: There are two cases depending on $\texttt{succ}(\pi_i, 2)$.

(a) If $\texttt{succ}(\pi_i, 2) \neq \texttt{NULL}$ then by definition $i = 1$ and $R_4(1) = R_4^1(1) \cup R_4^2(1)$ where

i. $R_4^1(1)$: Range of nodes visited after nodes in T_{b_1} and before nodes of T_{a_z}. If $\texttt{rmost\_leaf}(b_1)+1 \neq a_z$ then $R_4^1(1) = [\texttt{rmost\_leaf}(b_1)+1, a_z - 1]$ else $R_4^1(1) = \phi$. Note that $\texttt{rmost\_leaf}(b_1) + 1 \neq a_z$ means b_1 is not a leaf.

ii. $R_4^2(1)$: Range of nodes visited after nodes in T_{a_z} and before the right-most node of T_{a_1}. If $\texttt{rmost\_leaf}(a_z) \neq \texttt{rmost\_leaf}(a_1)$ then $R_4^2(1) = [\texttt{rmost\_leaf}(a_z) + 1, \texttt{rmost\_leaf}(a_1)]$ else $R_4^2(1) = \phi$.

(b) If $\texttt{succ}(\pi_i, 2) = \texttt{NULL}$ and $\texttt{rmost\_leaf}(a_i) \neq b_i$ then $R_4(i) = [\texttt{rmost\_leaf}(b_i)+1, \texttt{rmost\_leaf}(a_i)]$ else $R_4(i) = \phi$.

Remark. For each $1 \le i \le k$, $1 \le j \le 4$, $R_j(i) \subseteq [1, n]$ are intervals, and $R_1(i) \cup R_2(i) \cup R_3(i) \cup R_4(i) = V(T_{a_i}) \cup [1, a_i - 1]$. Further, the range of nodes greater than `rmost_leaf`(a_i) is not relevant, as it follows from Lemma 16 that the starting point of a path intersecting with P should be in one of these ranges. See Figure 3 for a pictorial representation of the ranges and Table 2 and 3 summarise the ranges.

Figure 3: The regions generated by (a) π_1, and (b) π_i, $2 \le i \le n$. For $j \in \{1, 2, t, i, i+1\}$, `rmost_leaf`$(a_j) = d_j$ and `rmost_leaf`$(b_j) = c_j$.

Characterising intersection of a path Q with a heavy sub-path. Let $Q = (s, t)$ be a path in $\mathcal{P}$. Let the sequence of nodes in Q be $(s = u_1, u_2, \ldots, u_z = t), 1 \le z \le n$. Let y be the first node in the sequence such that $y \in V(P)$. From Lemma 7, it follows that Π partitions the vertices of P, and thus y belongs to a unique $\pi \in \Pi$.

Convention: For ease of presentation, in the rest of this section, $Q = (s, t)$ is a path in $\mathcal{P}$, and y denotes the first vertex in $(s = u_1, u_2, \ldots, u_z = t), 1 \le z \le n$ which is in P.

For path P and path $Q \in \mathcal{P}$ we define the many-to-one function $\alpha : \mathcal{P} \times \mathcal{P} \to \Pi \cup \{\texttt{NULL}\}$ as follows.

$$\alpha(P, Q) = \begin{cases} \texttt{NULL}, & \text{if } V(P) \cap V(Q) = \phi \\ \pi \in \Pi, & \text{if } y \in V(\pi) \end{cases}$$

As a consequence of the definition of $\alpha(P, Q)$ we have the following lemma.

Lemma 18. *For $1 \le i \le k$, $\alpha(P, Q) = \pi_i$ if and only if exactly one of the following is true.*

1. $i = 1$ and $s \in R_1(1)$ and $t \in V(T_{a_1})$.
2. $s \in R_2(i)$
3. $s \in R_3(i)$ and $\texttt{lca}(s, t) \le b_i$
4. $s \in R_4(i)$ and $\texttt{lca}(s, t) < b_i$

Proof. Let $\alpha(P, Q) = \pi_i$ then the position of the starting node of Q has three possibilities relative to $\pi_i = (a_i, b_i)$. They are as follows:

1. $s < a_i$: This can happen only when $i = 1$. By Proposition 14, $a_1 \in V(Q)$ and $y = a_1$. In this case, $s \in R_1(1)$ and $t \in V(T_{a_1})$. For $i \ne 1$, any path Q with $l \in R_1(i)$

and $t \in T_{a_i}$ will have to pass through b_{i-1}. This implies $\alpha(P, Q) \neq \pi_i$ and thus a contradiction.

2. $a_i \leq s \leq b_i$: By Lemma 16, $s = y \in [a_i, b_i]$ that is $s \in R_2(i)$.
3. $s > b_i$: In this case, $s \neq y$ and $y \in [a_i, b_i]$. Depending on the regions of π_i as described above we have two possibilities as shown below:

   (a) $s \in R_3(i)$ and $\texttt{lca}(s, t) \leq b_i$. By Proposition 14, $b_i \in V(Q)$ and $y = b_i$.

   (b) $s \in R_4(i)$ and $\texttt{lca}(s, t) < b_i$. In this case, $y \in [a_i, b_i - 1]$.

Exactly one of the conditions is satisfied as the ranges $R_j(i)$ are non-overlapping and paths start in any one of the ranges. On the other hand, if there exists an i such that any one of the four conditions as given below is true, then we show that $\alpha(P, Q) = \pi_i$.

1. $s \in R_1(1)$ and $t \in V(T_{a_1})$: In this case, $y = a_1$ and since $y \in V(\pi_1)$, $\alpha(P, Q) = \pi_1$.
2. $s \in R_2(i)$: In this case, $y = s$ and since $y \in V(\pi_i)$, $\alpha(P, Q) = \pi_i$.
3. $s \in R_3(i)$ and $\texttt{lca}(s, t) \leq b_i$: Since $l \in R_3(i)$ and $\texttt{lca}(l, r) \leq b_i$, $y = b_i$. Since $y \in V(\pi_i)$, $\alpha(P, Q) = \pi_i$.
4. $s \in R_4(i)$ and $\texttt{lca}(s, t) < b_i$: Since $s \in R_4(i)$ and $\texttt{lca}(s, t) < b_i$, $y \in [a_i, b_i - 1]$. Since $y \in V(\pi_i)$, $\alpha(P, Q) = \pi_i$.

$\square$

Function $\texttt{check}\alpha$. This is a useful function that returns true if $\alpha(P, Q) = \pi_i, \pi_i \in \Pi$, based on conditions of Lemma 18.

Lemma 19. *For path $P \in \mathcal{P}$, given as input the index i of a heavy sub-path $\pi_i \in \Pi$, successors of $\pi_1 \in \Pi$ and another path $Q \in \mathcal{P}$, there exists a function $\texttt{check}\alpha(i, Q, \Pi, \texttt{succ}(1, 1), \texttt{succ}(1, 2))$ that checks $\alpha(P, Q) = \pi_i$ in constant time.*

Proof. The check can be done in the following manner.

1. Compute the interval ranges $R_j, 1 \leq j \leq 4$, of π_i using its end points and its successor stored in Π. If $i = 1$ then we can get the successors $\texttt{succ}(1, 1)$ and $\texttt{succ}(1, 2)$ from the input. If $i \neq 1$ then it can be obtained from Π as follows. For $i \neq 1$, it is π_{i+1} unless $\pi_{i+1} = \texttt{succ}(1, 2)$ or $i = k$ where k is the number of heavy sub-paths in Π. Since there are only four ranges and from Lemma 10, $\texttt{rmost\_leaf}$ takes constant time, the ranges can be computed in constant time.
2. Check if $\alpha(P, Q) = \pi_i$ based on Lemma 18. It takes constant time as the four checks are based on comparisons and from Lemma 10, $\texttt{lca}(s, t)$ can be computed in constant time. If $\alpha(P, Q) = \pi_i$ then $\texttt{check}\alpha$ returns true.

Thus, $\texttt{check}\alpha$ checks $\alpha(P, Q) = \pi_i$ in constant time. $\square$

For $\pi \in \Pi$, let $\beta(\pi) = \{Q \mid Q \in \mathcal{P} \text{ and } \alpha(P, Q) = \pi\}$. We have the following lemma.

Lemma 20. *For all distinct $\pi, \pi' \in \Pi$, $\beta(\pi) \cap \beta(\pi') = \phi$.*

Proof. For each $\pi \in \Pi$, $\beta(\pi)$ is the pre-image of π under the function α. Since α is a function, it follows that if $\pi \neq \pi'$, $\beta(\pi) \cap \beta(\pi') = \phi$. $\square$

Let the *neighbourhood of a path* be the set of all paths that have non-empty intersection with it. We have the following theorem regarding neighbourhood.

Lemma 21. *Let $N(P)$ denote the neighbourhood of P. $N(P) = \biguplus_{\pi \in \Pi} \beta(\pi)$.*

Proof. For a path $Q \in N(P)$, $\alpha(P, Q) \neq NULL$, and thus Q is an element of $\beta(\pi)$ for some $\pi \in Pi$. By Lemma 20, for each pair of distinct π, π', $\beta(\pi) \cap \beta(\pi') = \phi$. Thus $N(P) = \biguplus_{\pi \in \Pi} \beta(\pi)$. ☐

4 The Succinct Data Structure

In this section, we present the construction of the succinct representation followed by the implementation of the queries. The input to our construction procedure is $(T, \mathcal{P})$ obtained from the path graph G using Gavril's method [11] where T is the clique tree and $\mathcal{P} = \{P_1, \ldots, P_n\}$ is the set of paths in it such that the paths correspond to vertices of G and have a non-empty intersection of their vertex sets if and only if the corresponding vertices are adjacent. The construction procedure starts by preprocessing the clique tree as explained in the previous section followed by storing it and the paths in a space efficient manner. We demonstrate a polynomial time construction mechanism without worrying about the most optimal way.

Figure 4: Succinct representation for the path graph G of Figure 1. Top left diagram shows the paths as points in a grid. Bottom left shows the path aliases stored in the wavelet tree. The right side shows the l_i and r_i values of paths, the balanced parentheses representation of the pre-processed clique tree along with the F and J data structures.

4.1 Construction of the Succinct Data Structure

Our succinct data structure for path graphs has two main parts - the clique tree T and the paths $\mathcal{P} = \{P_1, \ldots, P_n\}$ in it. The construction uses other compact data

structures [19] which are of the types: ordinal tree, bit vector, wavelet tree, and array of sorted integers. In the next two sections we will explain the construction and storage of the clique tree and the paths in it.

4.1.1 Storing the Clique Tree

In this section we explain how the clique tree T and its BP representation is stored succinctly.

Clique tree T. By Remark 4, the clique tree has at most n nodes and is an ordinal tree. It is stored using $2n + o(n)$ bits using the data structure of Lemma 10.

Bit-vector BP. The balanced parentheses representation of T is stored using the data structure of Lemma 11 in bit-vector BP using $2n + o(n)$ bits. In BP the open and close parenthesis are represented by bit 1 and 0, respectively. For every node v in T there exists two indices i and j in BP where $i < j$ such that $BP[i] = 1$ and $BP[j] = 0$. For some $1 \le i \le 2n$, if $BP[i] = 1$ and $BP[i-1] = 0$ then they represent the open and close parenthesis of nodes $v, u \in V(T)$ that have a common parent w. Since T is ordinal, in the order of children of w, u comes immediately before v and it is called v's *previous sibling*. The following three methods are supported by BP:

1. `getPreorder(i)`: For $1 \le i \le 2n$ such that $BP[i] = 1$, returns the pre-order label of the node which has its open parenthesis at i in BP. It is implemented by $\texttt{rank}(BP, 1, i)$ for $i \ne 0$ and for $i = 0$ it returns 1.

2. `getIndex(v)`: Returns the index of the open parenthesis of $v \in V(T)$ in BP. It is implemented by $\texttt{select}(BP, 1, v)$.

3. `getHPStartNode(v)`: Returns the start node of heavy path π that contains $v \in V(T)$ in constant time. If v is not the first child, that is, it is adjacent to its parent by a light edge, then v itself is returned else the method returns $\texttt{getPreorder}(\texttt{select}(BP, 0, \texttt{rank}(BP, 0, \texttt{getIndex}(v))) + 1)$.

Lemma 22. *For $v \in V(T)$, $\texttt{getHPStartNode}(v)$ returns in constant time the starting node of heavy path π that contains v.*

Proof. We need to show that the method `getHPStartNode` as implemented above indeed obtains the start node of π in constant time. As we use constant time methods of Lemma 11, `getHPStartNode` also completes in constant time. To show that `getHPStartNode` returns the starting node of π we consider the two cases depending on v:

1. *When v is the root node of T i.e. $v = 1$:* `getIndex(v)` returns 1 when $v = 1$ and $\texttt{select}(BP, 0, \texttt{rank}(BP, 0, 1))$ returns 0. Further, `getPreorder` on input 1 returns 1. Thus, `getHPStartNode` returns v when input v is the root node, as it is the start node of π.

2. *When v is not the root node of T i.e. $v \ne 1$:* Let $BP[i]$ be the open parenthesis of v and x denote the starting node of π. Also, let $BP[j]$ be the closing parenthesis of the previous sibling of x in BP. Since $BP[j + 1]$ is the open parenthesis of x,

the length of path from v to x is $l = i - j - 1$. The base case is when $l = 0$ that is when v is the starting node of π. In this case, `getHPStartNode` returns v itself. When $l > 0$, `getIndex`(v) returns the position i of the open parenthesis of v in BP. `select`$(BP, 0, $`rank`$(BP, 0, i))$ returns j, the index of the closing parenthesis of the previous sibling of x. `getPreorder`$(j + 1)$ thus returns the start node of π correctly.

$\square$

4.1.2 Storing the Paths $P_1, \ldots, P_n$

To store path $P_i, 1 \leq i \leq n$ we need to store its starting node l_i and its ending node r_i in a space efficient way. Let $M = (M_1, \ldots, M_n)$ and $N = (N_1, \ldots, N_n)$ be the sequence of starting and ending nodes of paths sorted in non-decreasing order, respectively. For $1 \leq i \leq n$, $M[i]$ is the starting node of path P_i. On the other hand, for $1 \leq i \leq n$, $N[i]$ is the $i-$th ending node in the non-decreasing sorted order of ending nodes.

Bit-vectors F and J. M and N are stored in data structures F and J, respectively, using the data structure of Lemma 12 taking $2n + o(n)$ bits each.

Proposition 23. *For $1 \leq i \leq n$,* `accessNS`(F, i) *returns $M[i]$ stored in F in constant time.*

Proposition 24. *For $1 \leq i \leq n$,* `accessNS`(J, i) *returns $N[i]$ stored in J in constant time.*

F supports the following useful function too:

- `getPathCount`(d): Returns the number of paths that start at node $d \in V(T)$. When `select`$(F, 1, d)$ is well defined and $F[$`select`$(F, 1, d) + 1] = 0$, the count is obtained using the expression `rank`$(F, 0, $`select`$(F, 1, d+1)) - $`rank`$(F, 0, $`select`$(F, 1, d))$. In all other cases the function returns 0.

Lemma 25. *For $x \in V(T)$, method* `getPathCount`(x) *returns $|\{P_i : P_i \in \mathcal{P}, l_i = x\}|$ where l_i is the starting node of path P_i in constant time.*

Proof. Let input $x \in [n]$ be a valid l_i value of some path in $\mathcal{P}$, that is, `select`$(F, 1, d)$ is well defined and $F[$`select`$(F, 1, d) + 1] = 0$. If the l_i value x is repeating in F then there will be a contiguous sequence of two or more 0's between the x-th 1 and the $x+1-$th 1. Let n_1 be the number of 0's before the $x+1-$st 1. It can be obtained using the expression `rank`$(F, 0, $`select`$(F, 1, d+1))$. Let n_2 be the number of 0's before the $x-$th 1. n_2 can be obtained using the expression `rank`$(F, 0, $`select`$(F, 1, d))$. The number of times x is repeating is $n_1 - n_2$. As per Lemma 11 all these operations can be done in constant time. $\square$

Next, we need to associate the path P_i with its starting and ending nodes stored in F and J. Starting node of P_i is available directly from F using `accessNS`(F, i) whereas to get the ending node we need to associate it with its ending node's position in J. This association is established using a wavelet tree as described below.

Wavelet tree S. For each path $P_i, 1 \le i \le n$ we assign the tuple (f_i, j_i) where f_i and j_i are indices of the l_i and r_i values in M and N respectively. Since paths are numbered based on the non-decreasing order of their starting nodes, $i = f_i$. In other words, (f_i, j_i) acts as an alias for path $P_i = (l_i, r_i)$ and they have the following property.

Lemma 26. *Let $\mathcal{P}' = \{(f_1, j_1), \dots, (f_n, j_n)\}$ be the set of aliases of paths in $\mathcal{P}$. The following are true:*

1. $1 \le k \ne l \le n, f_k \ne f_l$ *and* $j_k \ne j_l$
2. $\mathcal{P}'$ *can be stored using the wavelet tree S using $n \log n + o(n \log n)$ bits of space such that S supports the following method:*

   (a) $\mathtt{accessWT}(S, i)$: *For $i \in [n]$, returns j_i in $O(\log n)$ time where $N[j_i]$ is the ending node of path P_i.*

   (b) $\mathtt{searchWT}(S, [i_1, i_2], [j_1, j_2])$: *For $i_1, i_2, j_1, j_2 \in [n]$, returns $\{i \mid P_i \in \mathcal{P}', f_i \in [i_1, i_2]$ and $j_i \in [j_1, j_2]\}$ in $O(\log n)$ time per path.*

   (c) $\mathtt{countWT}(S, [i_1, i_2], [j_1, j_2])$: *For $i_1, i_2, j_1, j_2 \in [n]$, returns $|\{i \mid P_i \in \mathcal{P}', f_i \in [i_1, i_2]$ and $j_i \in [j_1, j_2]\}|$ in $O(\log n)$ time.*

Proof. The proof is as follows:

1. $P_i, 1 \le i \le n$ has its starting and ending nodes stored at unique indices f_i and j_i in M and N, respectively. This ensures that for $1 \le k \ne l \le n, f_k \ne f_l$ and $j_k \ne j_l$.
2. For path $P_i, 1 \le i \le n$, the wavelet tree of Lemma 13 stores (f_i, j_i), where $M[i]$ and $N[j_i]$ are the starting and ending nodes of P_i. $\mathtt{accessWT}$, $\mathtt{searchWT}$, and $\mathtt{countWT}$ functions can be directly delegated to the $\mathtt{access}$, $\mathtt{search}$, and $\mathtt{count}$ functions of the wavelet tree of Lemma 13. The time complexities also follow from Lemma 13.

$\square$

Function $\mathtt{pathep}$. Given a path index $i, 1 \le i \le n$, we can now obtain its l_i and r_i values using the method $\mathtt{pathep}$. The method takes i as input and returns (l_i, r_i) in $O(\log n)$ time as follows.

1. $l_i = \mathtt{accessNS}(F, i)$.

2. $r_i = \mathtt{accessNS}(J, \mathtt{accessWT}(S, i))$.

Lemma 27. *For $1 \le i \le n$, $\mathtt{pathep}(i)$ returns (l_i, r_i) of P_i in $O(\log n)$ time.*

Proof. First we show that $\mathtt{pathep}(i)$ returns the l_i value of path i correctly. The l_i value of i is the number of 1's before the $i-$th 0 in F which is obtained by $\mathtt{accessNS}(F, i)$. Now, we show that the correct r_i value is returned by $\mathtt{pathep}(i)$. To get the r_i value which is stored in J we have to get the index j of path i in J. This can be obtained by querying S. We obtain the r_i value from J by $\mathtt{accessNS}(J, \mathtt{accessWT}(S, i))$. Since $\mathtt{accessNS}$ takes constant time as per Lemma 12 and $\mathtt{accessWT}$ takes $O(\log n)$ time as per Lemma 26, the total time taken is $O(\log n)$ time. $\square$

Function $\texttt{maprange}^F/\texttt{maprange}^J$. Given range $[l, l']$ of starting nodes of paths as input, $\texttt{maprange}^F$ outputs the range $[j, j']$ where j is the first index in M such that $M[j] \geq l$ and j' is the last index in M such that $M[j'] \leq l'$.

1. j is obtained using the expression $\texttt{rank}(F, 0, \texttt{select}(F, 1, l)) + 1$ that returns the index in M of the first occurrence of l or a value greater than l but less than or equal to l'.

2. To obtain j' we use the following steps:

   (a) If $M[\texttt{rank}(F, 1, \texttt{select}(F, 1, l') + 1)] = l'$ then return
   $\texttt{rank}(F, 0, \texttt{select}(F, 1, l') + 1) + \texttt{getPathCount}(l') - 1$. In other words, if l' is present in M then j' is the index of the last l' in M. To account for the repeating l' we add to the the first occurrence of l' in M one less than the number of times the l' value repeats.

   (b) If $M[\texttt{rank}(F, 1, \texttt{select}(F, 1, l') + 1)] \neq l'$ then return
   $\texttt{rank}(F, 0, \texttt{select}(F, 1, l'))$. If l' is not present then $M[j']$ is a value that is less than l' but greater than or equal to l.

   $M[z], z \in [n]$ can be obtained using $\texttt{accessNS}(F, z)$.

Lemma 28. *Given a range $[l, l']$ of starting nodes where $l, l' \in [n]$, $\texttt{maprange}^F(l, l')$ returns the range $[j, j']$ in constant time where j and j' are the smallest and largest indices in M such that $M[j] \geq l$ and $M[j'] \leq l'$.*

Proof. First we will show that j is computed correctly by the expression $\texttt{rank}(F, 0, \texttt{select}(F, 1, l)) + 1$. In the unary encoding in F, $\texttt{select}(F, 1, l)$ identifies the position i of the l−th 1. If l is present in F then $F[i + 1]$ is a 0 else its a 1. If $F[i + 1] = 0$ then $\texttt{rank}(F, 0, i) + 1$ returns the index j of l in M. On the other hand, if $F[i + 1] = 1$ then let k be the smallest number such that $F[i + k] = 0$. In this case, $\texttt{rank}(F, 0, i) + 1$ returns the smallest index j in M of $l'' > l$. Next, we show that j' is returned correctly. If l' is present in M then $\texttt{rank}(F, 0, \texttt{select}(F, 1, l') + 1)$ returns the index j'' of the first l' in M. The largest index in M of l' is obtained by adding $\texttt{getPathCount}(l') - 1$ to j''. On the other hand, if l' is not in M then $\texttt{rank}(F, 0, \texttt{select}(F, 1, l'))$ returns the largest index j in M of $l'' < l'$. By Lemma 11, $\texttt{rank}$ and $\texttt{select}$ can be completed in constant time. Also, by Lemma 12, $\texttt{accessNS}$ takes constant time. Thus, $\texttt{maprange}^F$ completes in constant time. $\square$

We have a similar function, $\texttt{maprange}^J$ for mapping the range $[r, r']$ of ending nodes of paths to range $[j, j']$ such that j is the smallest index in N such that $N[j] \geq r$ and j' is the largest index in N such that $N[j'] \leq r'$.

Lemma 29. *Given a range $[r, r']$ of ending nodes where $r, r' \in [n]$, $\texttt{maprange}^J(r, r')$ returns the range $[j, j']$ in constant time where j and j' are the smallest and largest indices in N such that $N[j] \geq r$ and $N[j'] \leq r'$.*

Bit-vector D. Bit vector D of size n stores for each path a 1 if the path intersects with more than $\log n$ other paths else a 0. It supports the following function.

- `isLargeDegree`(i): Returns true if $D[i] = 1$ else false in constant time.

Lemma 30. *There exists an $n \log n + o(n \log n)$-bit succinct data structure for path graphs.*

Proof. The space taken by the components of the succinct data structure for path graphs are as follows:

1. The clique tree T and its BP representation takes $O(n)$ bits of space. This follows from Lemma 10 and 11.
2. To store the end points of paths in $\mathcal{P}$ we have bit vectors F, J. From Lemma 12 this also takes $O(n)$ bits.
3. The wavelet tree stores the indices of paths in M and N and from Lemma 26 takes $n \log n + o(n \log n)$ bits.
4. To improve the degree query we have the n bit vector D.

The space complexity of the succinct representation is dominated by the space required for wavelet tree S. Thus, our representation takes $n \log n + o(n \log n)$ bits. This representation is succinct as it uses the permitted storage for succinct representation of interval graphs [4] that is a proper sub-class of path graphs [3]. $\qquad \square$

4.2 Adjacency and Neighbourhood Queries

In this section, we will present efficient implementations of adjacency and neighbourhood queries using the succinct representation as constructed in Section 4.1. In this section, as a consequence of Lemma 30, the succinct representation for path graph G is denote as $(T, \mathcal{P})$. Adjacency query, as will be shown in Lemma 33, takes two path indices $i, j \in [n]$ and the succinct representation $(T, \mathcal{P})$ as input and returns true if the paths P_i and P_j have a non-empty intersection. The neighbourhood query, as will be shown in Lemma 34, takes a single path index $i \in [n]$ and the succinct representation $(T, \mathcal{P})$ as input and returns the list of paths that have non-empty intersection with of the path P_i. The implementation of the queries depend on the following:

1. Computing paths $P = (l, r)$ and $Q = (s, t)$ corresponding to i and j, respectively using `pathep` and $p = \texttt{lca}(l, r)$ in $O(\log n)$ time.

2. Computing $\Pi, k, \texttt{succ}(1, 1)$ and $\texttt{succ}(1, 2)$. From Lemma 31 that follows, Algorithm 1 can do this in $O(\log n)$ time.

3. Computing $\beta(\pi)$ for $\pi \in \Pi$. From Lemma 32 that follows, $\beta(\pi)$ can be computed in $O(d \log n)$ time where d is the number of paths returned by $\beta(\pi)$.

Lemma 31. *Given a path $P = (l, r)$, $\texttt{compute}\Pi(l, r)$ computes Π, k, $\texttt{succ}(1, 1)$, and $\texttt{succ}(1, 2)$ for it in $O(\log n)$ time.*

Proof. First we show that $\texttt{compute}\Pi$ of Algorithm 1 computes the heavy sub-paths of P as in Lemma 8. Function $\texttt{compute}\Pi$ depends on the function $\texttt{compute}\Pi\_\texttt{Helper}$ to compute the heavy sub-paths. Paths are of two types depending on whether the lca is same as its starting node. Based on this distinction different steps are executed in the function $\texttt{compute}\Pi$; see Line 5 of Algorithm 1.

Algorithm 1: Given path $P = (l, r)$ as input, function $\texttt{compute}\Pi$ computes Π, k, $\texttt{succ}(1,1)$, and $\texttt{succ}(1,2)$. We assume that $\texttt{parent}(v) = 0$ when v is the root of the tree.

---

1 Function $\texttt{compute}\Pi(l, r)$:

2 $p \leftarrow \texttt{lca}(l, r)$

3 $k \leftarrow 0$

4 $\Pi = \Pi' = \texttt{succ}(1,1) = \texttt{succ}(1,2) = \texttt{NULL}$

5   **if** $l \neq p$ **then**

6 $\texttt{compute}\Pi\_\texttt{Helper}(p, l, \Pi, k)$

7 $\texttt{compute}\Pi\_\texttt{Helper}(p, r, \Pi', k)$

8   Add second entries of Π and Π' as $\texttt{succ}(1,1)$ and $\texttt{succ}(1,2)$ respectively

9   Concatenate Π' to Π preserving the order $\prec$

10   **else**

11 $\texttt{compute}\Pi\_\texttt{Helper}(l, r, \Pi, k)$

12   **if** *first entry in* Π *has equal starting and ending nodes* **then**

13   If starting node of the first entry in Π is the parent of starting node of the second entry then $\texttt{succ}(1,1)$ is the second entry in Π and $\texttt{succ}(1,2) = \texttt{NULL}$

14   **else**

15   If ending node of the first entry in Π is the parent of the starting node of the second entry then $\texttt{succ}(1,1)$ is the second entry in Π and $\texttt{succ}(1,2) = \texttt{NULL}$

16 Function $\texttt{compute}\Pi\_\texttt{Helper}(l, r, \Pi, k)$:

17   **if** $l > r$ **then**

18   **return**

19

20 $p = l$

21 $u \leftarrow \texttt{getHPStartNode}(r)$

22   Increment k

23   **if** $u >= p$ **then**

24   Add (u, r) to beginning of Π

25 $\texttt{compute}\Pi\_\texttt{Helper}(p, \boldsymbol{parent}(u), \Pi, k)$

26   **else**

27   Add (p, r) to beginning of Π

---

1. Type 1 paths: If lca of P is not equal to l then the heavy sub-paths that comprise the sub-path from p to l are computed first. This is followed by computing the heavy sub-paths that comprise the sub-path from p to r. This is done using the function $\texttt{compute}\Pi\_\texttt{Helper}$ as shown in Line 6 and 7 of Algorithm 1. $\texttt{compute}\Pi\_\texttt{Helper}(p, l, \Pi, k)$ computes the heavy sub-paths recursively till π_1; see

Line 6 of Algorithm 1. Starting at l, the starting node of the heavy sub-path to which it belongs is obtained by using `getHPStartNode`; see Line 21 of Algorithm 1. The set of heavy sub-paths are computed in this manner till p is reached; see Line 23 to 25 of Algorithm 1. Similar steps are performed for `computeΠ_Helper`(p, r, Π, k); see Line 7 of Algorithm 1. This gives us the end points of the heavy sub-paths of P.

2. Type 2 paths: If lca of P is equal to l then the heavy sub-paths comprising the only sub-path from $l = p$ to r is computed using the function `computeΠ_Helper` as shown in Line 11 of Algorithm 1. Heavy sub-paths for type 1 paths are also computed just as heavy sub-paths for type 1; see Line 11 to 15 in Algorithm 1.

It takes $O(\log n)$ time to compute heavy sub-paths as there are $O(\log n)$ light edges (or heavy sub-paths) as per Lemma 8 and as per Lemma 22, `getHPStartNode` takes constant time. From Lemma 10, `lca` and `parent` also take constant time. Since function `computeΠ` calls `computeΠ_Helper` only a constant number of times, the complexity of the `computeΠ` function is also $O(\log n)$. □

From Lemma 21, we know that the neighbourhood query depends on computing $\beta(\pi)$ for all $\pi \in \Pi$. Next, we show that $\beta(\pi)$ can be computed in $O(d_\pi \log n)$ time where d_π is $|\beta(\pi) = \{Q | Q \in \mathcal{P} \text{ and } \alpha(P, Q) = \pi\}|$. By an abuse of terminology, d_π is called the *degree* of π.

Lemma 32. *Given index i of $\pi_i \in \Pi$, there exists a function* `compute`$\beta(i, \Pi, \text{succ}(1, 1), \text{succ}(1, 2))$ *that returns* $\beta(\pi_i) = \{Q | Q \in \mathcal{P} \text{ and } \alpha(P, Q) = \pi_i\}$ *in* $O(d_{\pi_i} \log n)$ *time where* d_{π_i} *is the degree of* π_i.

Proof. First we will show that there exists a function `compute`$\beta(i, \Pi, \text{succ}(1, 1), \text{succ}(1, 2))$ that computes $\beta(\pi_i)$ correctly. The high level steps of function `compute`β are as follows.

1. Compute the interval ranges $R_j, 1 \le j \le 4$, of π_i using its end points and its successor stored in Π. If $i = 1$ then the successors are directly available in the input else it can be obtained from Π as follows. For $i \ne 1$, it is π_{i+1} unless $\pi_{i+1} = \text{succ}(1, 2)$ or $i = k$ where k is the number of heavy sub-paths in Π.

2. The next step is to identify all $Q \in \mathcal{P}$ that satisfy $\alpha(P, Q) = \pi_i$. The ranges of starting and ending nodes of such paths can be obtained from the conditions of Lemma 18. Using these ranges the paths can be retrieved by issuing orthogonal range search queries on wavelet tree S of Lemma 26. The ranges corresponding to first two conditions of Lemma 18 can be directly obtained. For the last two conditions we use Lemma 17.

3. `searchWT` from Lemma 26 is used to perform the orthogonal range search on wavelet tree S.

As there are only four interval ranges for π_i and from Lemma 10, `rmost_leaf` takes constant time, the interval ranges of π_i can be computed in constant time. From these interval ranges the ranges for orthogonal range search can be obtained using Lemma 18. This can be done in constant time as from Lemma 10, `lca` takes constant time. `searchWT` takes $O(d \log n)$ time per range query where d is the number of paths in $\mathcal{P}$ with starting and ending nodes in the input range. There is no over counting of

paths between range search queries as no path satisfies more than one condition due to Lemma 18. Since there are only four orthogonal range queries to be issued for any heavy sub-path, $\texttt{compute}\beta$ completes in $O(d_{\pi_i} \log n)$ time. $\qquad\square$

Adjacency query in $O(\log n)$ time. Given indices of paths $i, j \in [n]$ and $(T, \mathcal{P})$ as input, adjacency query returns $\texttt{true}$ if paths corresponding to i and j, namely P and Q, have a non-empty intersection in T. Adjacency of paths with indices i and j can be checked as shown in Algorithm 2. We have the following lemma.

Algorithm 2: Given two path indices $i, j \in [n]$, the function $\texttt{adjacency}$ checks if the paths corresponding to them have a non-empty intersection.

---

1 Function $\texttt{adjacency}(i, j)$:
2   Obtain paths $P = \texttt{pathep}(i)$ and $Q = \texttt{pathep}(j)$. Let $P = (l, r)$ and $Q = (s, t)$.
3   Initialize $k \leftarrow 0$ and Π to empty
4 $(\Pi, k, \texttt{succ}(1, 1), \texttt{succ}(1, 2)) \leftarrow \texttt{compute}\Pi(l, r)$
5   For each $1 \leq i \leq k$ return true if $\texttt{check}\alpha(i, Q, \Pi, \texttt{succ}(1, 1), \texttt{succ}(1, 2))$ of Lemma 19 returns true

---

Lemma 33. *Given two path indices $i, j \in [n]$ and $(T, \mathcal{P})$ as input, the function $\texttt{adjacency}(i, j)$ checks if paths corresponding to i and j have a non-empty intersection in $O(\log n)$ time.*

Proof. By definition, if $\alpha(P, Q) = \pi$ for $\pi \in \Pi$ then Q and P are adjacent. The existence of such a heavy sub-path can be tested as shown in Line 5 of Algorithm 2. Paths $P = (l, r)$ and $Q = (s, t)$ corresponding to i and j, respectively, can be obtained in $O(\log n)$ time using $\texttt{pathep}$ due to Lemma 27. By Lemma 31, Π, k, and the successors of π_1 can be computed in $O(\log n)$ time. For each heavy sub-path $\pi \in \Pi$, the conditions of Lemma 18 can be checked in constant time using $\texttt{check}\alpha$ of Lemma 19. Also, from Lemma 10, $\texttt{rmost\_leaf}$ can be computed in constant time. Since by Lemma 8, Π contains at most $O(\log n)$ heavy sub-paths, the total time taken is $O(\log n)$. $\qquad\square$

Neighbourhood query. Given a path index $i \in [n]$ and $(T, \mathcal{P})$, the neighbourhood query returns the neighbours of path P corresponding to index i; see Lemma 21 for definition of neighbours of a path. Let $N(P)$ be initialized to empty. $N(P)$ can be obtained as shown in Algorithm 3. We call $|N(P)|$ the *degree* of P. We have the following lemma.

Lemma 34. *Given path index $i \in [n]$ of path $P \in \mathcal{P}$ and $(T, \mathcal{P})$ as input, the function $\texttt{neighbourhood}(i)$ returns the set of neighbours of P in $O(d_P \log n)$ time where d_P is the degree of P.*

Proof. From Lemma 21, the neighbours of P are the paths in $\biguplus_{i=1}^{k} \beta(\pi_i)$. The end points of P can be obtained in $O(\log n)$ time using $\texttt{pathep}$ due to Lemma 27. From

---
Algorithm 3: Given path index i, the function `neighbourhood` enumerates the paths that have non-empty intersection with P.
---
1 Function `neighbourhood`(i):

  2   Obtain end points (l, r) of P using `pathep`(i)

  3   Initialize $k \leftarrow 0$ and Π to empty

  4   Compute $(\Pi, k, \text{succ}(1, 1), \text{succ}(1, 2))$ for P using the `computeII`(l, r) function

  5   For each $\pi \in \Pi$ add `compute`$\beta(\pi)$ of Lemma 32 to $N(P)$.

Lemma 31, we know that `computeII` takes $O(\log n)$ time and from Lemma 32, we know that `compute`β takes $O(d_\pi \log n)$ time for each $\pi \in \Pi$ where d_π is the number of paths that α maps to π. The time taken by `neighbourhood` is sum of the time taken by `pathep`, `computeII` and at most k iterations of `compute`β. Since by Lemma 21, we know that none of the neighbours are over-counted the total time taken is $O(d_P \log n)$ where $d_P = \sum_{i=1}^{k} d_{\pi_i}$ where d_P is the degree of path P. $\qquad\square$

Degree query. Degree of path P can be obtained by two different methods depending on the degree of the path. We use a bit vector D as described in Section 4.1. We have two methods for computing degree of P with index i depending on `isLargeDegree`(i).

    1. `isLargeDegree`(i) is true: We modify Algorithm 3 for `neighbourhood` to return the count of the orthogonal range search instead of the paths by using `countWT` of Lemma 26 instead of `searchWT`.

    2. `isLargeDegree`(i) is false: We run the Algorithm 3 for `neighbourhood` without modification and count the number of paths returned.

Lemma 35. *Given path index $i \in [n]$ of path $P \in \mathcal{P}$ and $(T, \mathcal{P})$ as input, the function* `degree`(i) *returns the degree of P in* $\min\{O(\log^2 n), O(d_P \log n)\}$ *time where d_P is the degree of P.*

Proof. As described above, two different methods are used depending on whether `isLargeDegree`(i) is true or not. Thus, we have the following two cases:

1. `isLargeDegree`(i) is true: `countWT` of Lemma 26 takes $O(\log n)$ time. Since there are $O(\log n)$ heavy sub-paths as per Lemma 8, the total time is $O(\log^2 n)$.
2. `isLargeDegree`(i) is false: By Lemma 34, Algorithm 3 takes $O(d_P \log n)$ time. Thus, degree also takes $O(d_P \log n)$ time.

Since we run only one of the two depending on which is better, the time taken by degree query is $\min\{O(\log^2 n), O(d_P \log n)\}$. $\qquad\square$

Proof of Theorem 1. Lemma 30 shows that there exists a succinct representation for path graphs that takes $n \log n + o(n \log n)$ bits. Given this representation as input, Lemma 33 shows that adjacency between vertices can be checked in $O(\log n)$ time. Similarly, given this representation as input Lemma 34 and 35 show that for vertex $u \in V(G)$ with degree d_u, neighbourhood and degree queries are supported in $O(d_u \log n)$ and $\min\{O(\log^2 n), O(d_u \log n)\}$ time, respectively. Hence, Theorem 1. $\square$

5 The Space-Efficient Data Structure

We present an $O(n \log^2 n)$-bit space-efficient representation for path graphs that supports faster adjacency and degree queries in comparison to the succinct representation presented in Section 4. The approach we take is to represent a path graph using the succinct data structure for interval graphs due to Acan et al. [4]. To represent the path graph using the interval graph representation in [4] we end up having multiple *copies* of each vertex, and the adjacency between vertices could be witnessed in different interval graphs in our transformation. Our data structure stores these interval graphs using the representation of [4], along with an additional table to keep track of the copies of the vertices and edges. This transformation has an interesting contrast to the succinct data structure in Section 4; there the path graph is represented using the clique tree and the adjacency queries are transformed to range queries.

The path graph G is presented as $(T, \mathcal{P})$, where T is a clique tree of G and $\mathcal{P} = \{P_1, \ldots, P_n\}$ is the set of paths in T. Consider the heavy path tree $\mathcal{T}$ of T. Let $\mathcal{H}$ denote the set of heavy paths of T.

Convention: Let M denote $|V(T)|$. It follows from Remark 4 that M and $|\mathcal{H}|$ are at most n. P_v denotes the path in $\mathcal{P}$ corresponding to vertex $v \in V(G)$. For a node $w \in \mathcal{T}$, we use H_w to denote the heavy path in T associated with the node w. The level number of a node in $\mathcal{T}$ is one more than the number of edges on the path to it from the root; thus the level number of the root is 1. K denotes the number of levels in $\mathcal{T}$ and level l consists of the heavy paths which are at that level in $\mathcal{T}$. From Lemma 6, $\mathcal{T}$ has at most $\lceil \log n \rceil$ levels and each path P in $\mathcal{T}$ has at most $2\lceil \log n \rceil$ edges.

Lemma 36. *For any $P, Q \in \mathcal{P}$, there exists a node v in $\mathcal{T}$ such that $\Phi^{-1}(v)$ has a non-empty intersection with the path $P \cap Q$.*

Proof. From Lemma 7, we know that nodes of T are partitioned among the nodes of $\mathcal{T}$. This implies nodes of $P \cap Q$ belong to some $v \in V(\mathcal{T})$. Thus, for some $v \in V(\mathcal{T})$, $\Phi^{-1}(v)$ intersects with the path $P \cap Q$. $\square$

Lemma 37. *For u and v in $V(G)$, P_u and P_v have a non-empty intersection in T if and only if one of the following is true:*

1. *there is a light edge $\{w, w'\}$ in $\mathcal{T}$ such that P_u and P_v both intersect H_w and $H_{w'}$*

2. *there is exactly a node w in $\mathcal{T}$ such that $P_u \cap H_w$ and $P_v \cap H_w$ have a non-empty intersection.*

Proof. If $V(P_u) \cap V(P_v) \neq \phi$ then there are two possibilities:

1. there exist $t_1, t_2 \in V(P_u) \cap V(P_v)$ and $t_1 \in V(H_w)$ and $t_2 \in V(H_{w'})$ such that there exists a light edge $\{w, w'\}$ in $\mathcal{T}$.
2. there exists only one $H_w \in \mathcal{H}$ that contains all nodes in $V(P_u) \cap V(P_v)$ for some $w \in V(\mathcal{T})$. In this case, $P_u \cap H_w$ and $P_v \cap H_w$ have a non-empty intersection.

Conversely, if P_u and P_v both intersect heavy paths H_w and $H_{w'}$ where $w, w' \in V(\mathcal{T})$ then P_u and P_v share the light edge $\{w, w'\}$. Thus, they intersect in T. If $P_u \cap H_w$ and $P_v \cap H_w$ intersect then by definition P_u and P_v intersect in T. □

Interval graph associated with heavy path H. For a heavy path H associated with a node in $\mathcal{T}$ of level number l, G_H is a graph whose vertices are defined as follows: for each $1 \leq i \leq n$, if $P_i \cap H \neq \phi$ then there is a vertex corresponding to $P_i \cap H$ in $V(G_H)$. Two vertices are adjacent in G_H if the corresponding paths have a non-empty intersection, otherwise they are not adjacent.

Lemma 38. *Let $H \in \mathcal{H}$. Then G_H is an interval graph.*

Proof. The vertices of G_H correspond to paths in $\mathcal{P}$ that have non-empty intersection with H. Thus, it follows that each vertex of G_H corresponds to a sub-path of H, which is equivalently an interval in the set $\{1, 2, \ldots, |V(H)|\}$. Thus, G_H is an interval graph. □

Lemma 39. *Let P_u and P_v be paths in T. $P_u \cap P_v \neq \phi$ if and only if there exist a heavy path $H \in \mathcal{T}$ such that in the interval graph G_H, the vertices corresponding to $P_u \cap H$ and $P_v \cap H$ are adjacent.*

Proof. The proof follows directly from Lemma 37. □

It follows from the above lemma that each edge in G has a representative in the interval graph associated with at least one of the heavy paths in $\mathcal{T}$. Thus, it is natural to group all the interval graphs into the levels associated with the heavy paths in $\mathcal{T}$.

Interval graph associated with a level l in $\mathcal{T}$. Let S_l be the set of nodes in $\mathcal{T}$ at level l. For each l, define $U_l = \{G_{H_w} \mid w \in S_l\}$. In other words, U_l is the collection of interval graphs associated with each heavy path at level l. Thus, the vertex set and edge set of U_l is the union of vertex sets and edge sets of G_{H_w} for all $w \in S_l$. Clearly, U_l is an interval graph. We next show that the number of vertices in U_l is at most twice the number of vertices in G, that is, at most twice the number of paths in $\mathcal{P}$.

Lemma 40. *Let P_v be a path in $\mathcal{P}$ and l be a level number in $\mathcal{T}$. There exists at most two nodes w and w' at level l of $\mathcal{T}$ such that G_{H_w} and $G_{H_{w'}}$ have a vertex each corresponding to the paths $P_v \cap H_w$ and $P_v \cap H_{w'}$. Therefore, the number of vertices in the interval graph U_l is at most $2n$.*

Proof. We know from Lemma 8 that the nodes of P_v are partitioned into heavy sub-paths, each of which is contained in a heavy path. Since each heavy path corresponds to a node in $\mathcal{T}$, it follows that P_v naturally defines a path P in $\mathcal{T}$. From Proposition 9, it follows that P has at most two nodes in level l. Thus, for each level l, P_v has a non-empty intersection with at most two heavy paths whose nodes are at level l in $\mathcal{T}$. Consequently, U_l has at most $2n$ vertices. $\qquad\square$

In Section 5.1, we present the data structure to store the set of interval graphs $\{U_l \mid 1 \leq l \leq K\}$ and additional tables to respond to the adjacency and neighborhood queries.

5.1 Construction of the Space Efficient Data Structure

The main goal of this section is to prove the space complexity part of Theorem 2. Given the $(T, \mathcal{P})$ representation for a path graph G with n vertices, the space-efficient data structure is constructed by the following steps:

1. Compute the heavy path decomposition of clique tree T and the heavy path tree $\mathcal{T}$ from T as guaranteed in Section 2.2.

2. Construct the ordinal clique tree T as explained in Section 3.

3. Store the set of interval graphs $\{U_l \mid 1 \leq l \leq K\}$ using [4] and a table called PIT that stores, for every level $1 \leq l \leq K$, the labels of vertices in G_l corresponding to paths in $\mathcal{P}$; by Lemma 40 there are two labels per path at a level l.

The construction of the data structure takes polynomial time and implementation details are left out. The components of the space-efficient representation are as follows.

Array, F. This is a one dimensional array of length M. $F[a] = i$ where i is the index, in $\prec_{\mathcal{H}}$, of the heavy path which contains $a \in V(T)$. So to store the heavy paths to which all nodes of T belong we need $O(n \log n)$ bits. The following function is supported by F.

- `getHeavyPath(a)`: Returns the heavy path number of $H \in \mathcal{H}$ to which $a \in V(T)$ belongs in constant time.

Array, L. This is a one dimensional array of length $|\mathcal{H}|$. $L[i] = l$ where l is the level in $\mathcal{T}$ to which heavy path $H_i \in \mathcal{H}$ belongs. Each entry in the array uses $O(\log \log n)$ bits, since from Lemma 6, there are $O(\log n)$ levels in $\mathcal{T}$. Since $|\mathcal{H}| \leq n$, the total space taken by L is $O(n \log \log n)$ bits. The following function is supported.

- `getLevel(j)`: Returns in constant time the level to which heavy path $H_j \in \mathcal{H}$, for $j \in [n]$, belongs in $\mathcal{T}$.

Array, E. This is a $K \times 2n$ two dimensional array. Each row corresponds to a level of $\mathcal{T}$ and column corresponds to a vertex in $U_l, 1 \leq l \leq K$; by Lemma 40, U_l has at most $2n$ vertices. $E[l][v] = i$ where i is the index of the path $P_i \in \mathcal{P}$ that has non-empty intersection with a heavy path H at level l and v is the vertex in G_l corresponding to $P_i \cap H$. For vertex labels that are not present in U_l, $E[l][v]$ stores 0. The total space taken by E is $O(n \log^2 n)$ as there are $2n \log n$ entries and each entry takes $\log n$ bits.

- `getPathIndex(l, v)`: Returns the path index stored at $E[l][v]$ in constant time given $1 \leq l \leq K$ and $1 \leq v \leq 2n$ as input.

The Path Intersection Table (PIT). PIT is an $n \times K$ two dimensional array of records with rows corresponding to paths of $\mathcal{P}$ and columns to the levels of $\mathcal{T}$. For path index i and level l, each record consists of a bit and two vertex labels $v, v' \in V(G_l)$ such that v denotes $P_i \cap H_w$ and v' denotes $P_i \cap H_{w'}$ for some heavy paths H_w and $H_{w'}$ corresponding to nodes $w, w' \in V(\mathcal{T})$ at level l. The entries in PIT are as follows:

1. $PIT[i][l] = 1$, if the path P_i has non-empty intersection with some heavy path at level l else $PIT[i][l] = 0$. Storing this information takes $n \log n$ bits.

2. $PIT[i][l]$ stores the labels of the two vertices in interval graphs G_{H_w} and $G_{H_{w'}}$ corresponding to $P_i \cap H_w$ and $P_i \cap H_{w'}$ where $w, w' \in V(\mathcal{T}$ at level l. If P_i does not belong to the level l then we store `NULL`. If P_i belongs to the level l but to only one interval graph, say G_{H_w}, then $PIT[i][l]$ stores the label of the vertex in G_{H_w}. Each entry of the PIT takes at most $2 \log n$ bits, since by Lemma 40 there are at most two labels for a path per level.

PIT has $n \log n$ entries and each entry takes $O(\log n)$ bits. Thus, total space needed is $O(n \log^2 n)$. PIT is constructed as follows. Note that as per Proposition 15, each heavy path in $\mathcal{H}$ is an interval.

1. Initialize an array of counters $c[l] \leftarrow 0$ for each level $1 \leq l \leq K$.

2. For each $i \in [n]$ perform the following steps.

   (a) For each $a \in V(P_i)$ and the heavy path $j \leftarrow$ `getHeavyPath`(a) such that a is the first vertex of P_i that is also in H_j, do the following steps.

      i. $l \leftarrow$ `getLevel`(j).
      ii. Store new vertex label $c[l] \leftarrow c[l] + 1$ corresponding to path $P_i \cap H_j$ at level l at $PIT[i][l]$.

The following functions are supported.

1. `isPresent`(i, l): Returns true if $PIT[i][l] = 1$ in constant time for path index $i \in [n]$ and the level $1 \leq l \leq K$.

2. `getVertices`(i, l): Returns the vertex labels stored at $PIT[i][l]$ in constant time for path index $i \in [n]$ and level $1 \leq l \leq K$. The vertices are ordered based on the total order of the heavy paths that define them.

The Interval Graph Table (IT). IT is a one dimensional array of length K. As a consequence of Lemma 38, $U_l, 1 \leq l \leq K$, is an interval graph. For level l, $IT[l]$ stores U_l using the method of Acan et al. [4]. Thus, the total space taken by IT is $O(n \log^2 n)$. IT can be constructed in polynomial time as follows. Populate $IT[l]$, for each level l of $\mathcal{T}$, using the following steps.

1. Let S_l be the set of heavy paths at level l. Obtain S_l from L and sort it in non-decreasing order.

2. Let UNUSED=0 and USED=1. For each $w \in S_l$ and path $P_i \in \mathcal{P}, i \in [n]$, do the following after initializing the bit vector B of length $2n$ to UNUSED.

   (a) If $P_i \cap H_w \neq \phi$ then add the vertex returned by getVertices(i, l) that is marked UNUSED in B to $V(G_{H_w})$. Once a vertex corresponding to a path in PIT is added to the interval graph it is marked as USED in B.

   (b) For every vertex u added to $V(G_{H_w})$, add u into a temporary array TEMP$[w]$ along with $V(P_i \cap H_w)$.

3. For every $w \in S_l$, add edges to interval graph G_{H_w} as follows.

   (a) For all pairs of vertex labels u and v in TEMP$[w]$ where u corresponds to $P_i \cap H_w$ and v corresponds to $P_j \cap H_w$, add edge $\{u, v\}$ to $E(G_{H_w})$ if $V(P_i \cap H_w) \cap V(P_j \cap H_w) \neq \phi$.

4. Finally, we get $U_l = \{G_{H_w} \mid w \in S_l\}$.

U_l thus obtained can now be stored using the data structure of [4]. The following functions are supported by IT.

1. adjacentIG(u, v, l): Returns true if $u, v \in V(U_l)$ are adjacent in constant time. This adjacency check is delegated to the interval graph representation of [4]. [4] supports constant time adjacency query.

2. neighbourhoodIG(u, l): Returns the neighbours of vertex $u \in V(U_l)$ in $O(d_u)$ time where d_u is the degree of vertex u. The query is delegated to the interval graph representation of [4]. [4] returns neighbours in constant time per neighbour.

Array R. This is an $n \times 2$ two dimensional array. For $P_i \in \mathcal{P}$, $R[i][1] = a_i$ and $R[i][2] = b_i$ where a_i and b_i are the lowest and highest levels to which heavy paths $H_w, H_{w'} \in \mathcal{H}$ belong in $\mathcal{T}$ such that $H_w \cap P_i \neq \phi$ and $H_{w'} \cap P_i \neq \phi$. Since P_i is a path, it has non-empty intersection with some heavy path at all the levels in the range $[a_i, b_i]$. We say, P_i spans the levels from a_i to b_i and denote this range by an interval $I_i = [a_i, b_i], 1 \leq a_i \leq b_i \leq K$. Each row of R consists of two values, each taking $O(\log \log n)$ bits since $K \leq \log n$. R takes a total space of $O(n \log \log n)$ bits. The following functions are supported by R:

1. getEndPoints(i): Returns the end points of I_i for path $P_i \in \mathcal{P}$ in constant time for path index $i \in [n]$.

2. getMinLevel(i, j): Returns the left end point of $I_i \cap I_j$ in constant time for path indices $i, j \in [n]$ if $I_i \cap I_j \neq \phi$ else returns 0. The function returns:

   (a) if $b_i < a_j$ or $b_j < a_i$ then 0

(b) else if $a_i \leq b_j$ then a_i

(c) else if $a_j \leq b_i$ then a_j

Array A. This is a one dimensional array of length M. $A[a], 1 \leq a \leq M$, stores the list of paths that have their lca at node $a \in V(T)$. A path have only one lca and it takes $\log n$ bits to store this information as $M \leq n$. For n paths it takes $O(n \log n)$ bits. The following function is supported.

- getPathsLCA(a): Returns paths in $\mathcal{P}$ with lca at node $a \in V(T)$ in constant time.

Heavy path tree $\mathcal{T}$. The heavy path tree of clique tree T is stored using the method of Lemma 10 in $\mathcal{T}$. Since T is an ordinal tree, $\mathcal{T}$ is also ordinal. $\mathcal{T}$ takes $2n + o(n)$ bits and supports all the methods of ordinal trees as supported by the data structure of Lemma 10.

Array H. This is a one dimensional array of length $|\mathcal{H}|$; see Figure 5 for an example. Let $w \in V(\mathcal{T})$ have n_w children. Contents of H are as follows.

- $H[w]$ stores a one dimensional array C of length n_w with a location for each of the children of w.

- $H[w] = $ NULL if w does not have any children.

Since $\mathcal{T}$ is an ordinal tree, the children of a node are ordered. For the $i-$th child of w, denoted c, with n_c children, $C[i]$ stores a one dimensional array D of length $n_c + 1$. Contents of D are as follows.

- $D[j][1] = d, 1 \leq j \leq n_c$, where d is the $j-$th child of c and $D[j][2]$ contains a list that stores the paths that contain edges $\{d, c\}$ and $\{c, w\}$ where w, c, d belong to consecutive levels $l_1 < l_2 < l_3$, respectively, in $\mathcal{T}$.

- $D[n_c + 1][1] = $ NULL and $D[n_c + 1][2]$ stores the list of paths that contain only $\{c, w\}$ and no light edge incident on c in the sub-tree rooted at c.

- If c does not have a child then $D[1][1] = $ NULL and $D[1][2]$ contains the list of paths that contain light edge $\{c, w\}$.

C and D are of size $O(n \log n)$ bits as they store entries for edges of $\mathcal{T}$ which, as a consequence of Remark 4, is at most $n - 1$. Thus, H, C, and D take a total of $O(n \log n)$ bits. The following function is supported.

- getDistinctPaths(w_1, w_2, w_3): Returns, in constant time, the list of paths that contain light edge $\{w_1, w_2\}$ but not $\{w_2, w_3\}$ where $\{w_1, w_2\}, \{w_2, w_3\} \in E(\mathcal{T})$ and w_1, w_2, w_3 lie on consecutive levels $l_1 < l_2 < l_3$, respectively, in $\mathcal{T}$.

Lemma 41. *Let $\mathcal{T}$ be the ordinal heavy path tree and $\{w_1, w_2\}, \{w_2, w_3\} \in E(\mathcal{T})$ be two light edges such that w_1, w_2, w_3 lie on consecutive levels $l_1 < l_2 < l_3$, respectively, in $\mathcal{T}$. There exists a function that returns the list of paths that contain $\{w_1, w_2\}$ but not $\{w_2, w_3\}$ in constant time per path returned.*

Figure 5: A part of an example heavy path tree $\mathcal{T}$, is shown on the left side and array H is shown on the right side. $H[w]$ contains C with three entries corresponding to its children $\{a_1, a_2, a_3\}$. The array D corresponding to child a_2 at $C[2]$ is also shown. The first entry in the list stored at D corresponds to c_1 and is associated with a list containing only one entry, P_1. This means P_1 contains light edge $\{c_1, a_2\}$ and $\{a_2, w\}$. The last entry in D is NULL which implies there is no path that starts at a_2 and contains light edge $\{a_2, w\}$. Notice that the D corresponding to child c_2 of a_2, contains P_2 as the last entry.

Proof. The function `getDistinctPaths`(w_1, w_2, w_3) is implemented as follows.

1. $r \leftarrow$ `child_rank`(w_2). `child_rank` is a function supported by ordinal tree $\mathcal{T}$ that returns the number of siblings to the left of w_2. It takes constant time as per Lemma 10.

2. Obtain array D from $C[r+1]$ stored in $H[w_1]$. Let L' denote the list obtained by concatenating the lists stored at D except the list corresponding to w_3.

3. Return L'.

   `child_rank` takes constant time as per Lemma 10. Concatenating each list into one takes constant time per list concatenated. As each list contains at least one neighbour, the time taken is $O(1)$ per path returned. □

Lemma 42. *There exists an $O(n \log^2 n)$-bit space-efficient data structure for path graphs.*

Proof. The space taken by the components of the space-efficient data structure for path graphs are as follows:

1. Array F that contains the heavy paths to which each node of T belongs takes $O(n \log n)$ bits.

2. Array L stores the level to which each heavy path belongs taking $O(n \log \log n)$ bits. R stores the ranges of levels in $\mathcal{T}$ that a path spans taking $O(n \log \log n)$ bits.
3. For every level l, the path index corresponding to each of the vertex labels in U_l is stored in array E using $O(n \log^2 n)$ bits. PIT stores the levels to which paths in $\mathcal{P}$ belong. For each level l, the vertex labels in U_l corresponding to a path in $\mathcal{P}$ is stored using $O(n \log^2 n)$ bits. For each level l, IT stores the interval graph U_l taking $O(n \log^2 n)$ bits using the representation of [4].

Thus, the entire space-efficient data structure uses $O(n \log^2 n)$ bits. $\qquad \square$

5.2   Adjacency and Neighbourhood Queries

Next, we present the algorithms for the adjacency, neighborhood and degree queries and their time complexities. We have the following useful lemmata that we will use in the implementation of the queries.

Lemma 43. *Consider paths with indices $i, j \in [n]$ such that $[l_1, l_2]$ is the maximal range of levels with $PIT[i][l] = PIT[j][l] = 1$ for all $l \in [l_1, l_2]$. If paths P_i and P_j do not intersect in U_{l_1} then they do not intersect at any level $l > l_1$.*

Proof. If paths $P_i, P_j \in \mathcal{P}$ do not intersect in U_{l_1} then there are two possibilities:

1. They intersect two different heavy paths at level l_1 in the heavy path tree. In this case, they will not intersect in any level greater than l_1 as they are contained in two different branches of the heavy path tree.
2. They intersect the same heavy path at level l_1 but different heavy paths at levels greater than l_1. Thus, at any level $l > l_1$ they are in different branches of the heavy path tree and so will not intersect.

Hence, the lemma. $\qquad \square$

Lemma 44. *Let $P = (l, r)$ and $Q = (s, t)$ be two paths in $\mathcal{P}$ with sequence of heavy sub-paths Π_P and Π_Q, respectively. Also, let $V(P) \cap V(Q) \neq \phi$ such that there does not exist a light edge e such that $e \in E(P) \cap E(Q)$. The following are true.*

1. *There exists exactly one $\pi \in \Pi_P, \pi' \in \Pi_Q$ and heavy path $H = (h_1, h_2)$ such that $V(\pi) \cap V(\pi') \cap V(H) \neq \phi$.*
2. *Further, either $E(P) \subseteq E(T_{h_1})$ or $E(Q) \subseteq E(T_{h_1})$ where T_{h_1} is the sub-tree rooted at h_1.*
3. *The lowest numbered node of π is either the $\mathtt{lca}(l, r)$ or it is h_1 such that light edge $\{h_1, \mathtt{parent}(h_1)\} \in E(P)$.*
4. *Either, $\mathtt{lca}(l, r) \in V(\pi)$ or $\mathtt{lca}(s, t) \in V(\pi)$.*

Proof. The proof is as follows:

1. $V(P) \cap V(Q)$ is contained in some $H \in \mathcal{H}$, since by Lemma 7, the nodes of T are partitioned among the heavy paths. There is exactly one such H, as $V(P) \cap V(Q)$ does not have pair of nodes u, v such that $\{u, v\}$ is a light edge in T.
2. We consider two cases here.

(a) h_1 is the root of T: In this case, trivially $E(P) \subseteq E(T_{h_1})$ and $E(Q) \subseteq E(T_{h_1})$ since $T_{h_1} = T$.

(b) h_1 is not the root of T: If both P and Q do not contain light edge $\{\texttt{parent}(h_1), h_1\}$, then $E(P) \subseteq E(T_{h_1})$ and $E(Q) \subseteq E(T_{h_1})$. Else, since P and Q do not share a light edge, either $\{\texttt{parent}(h_1), h_1\} \in E(P)$ or $\{\texttt{parent}(h_1), h_1\} \in E(Q)$. Without loss of generality, let it be an element of $E(P)$. Then, $E(Q) \subseteq E(T_{h_1})$. Thus, if P and Q do not share a light edge, at least one of the paths must be contained inside T_{h_1}.

3. Based on the earlier proved statement, we have two cases:

   (a) $E(P) \subseteq E(T_{h_1})$: In this case, $\texttt{lca}(l, r) \in V(\pi)$ and is the lowest numbered node in π.

   (b) $E(P) \not\subseteq E(T_{h_1})$: In this case, $h_1 \in V(\pi)$ and is the lowest numbered node in π.

4. There are two possibilities based on the lowest numbered node in π.

   (a) If the lowest numbered vertex of π is the $\texttt{lca}(l, r)$ then the statement follows trivially.

   (b) If the lowest numbered vertex of π is h_1 such that light edge $\{h_1, \texttt{parent}(h_1)\} \in E(P)$ then $\texttt{lca}(s, t) \in V(\pi) \cap V(\pi')$; since $E(Q) \subseteq E(T_{h_1})$. Hence, the result.

   $\square$

Lemma 45. *For every $a \in V(T)$ there exists $P = (l, r)$ in $\mathcal{P}$ such that $a = \texttt{lca}(l, r)$.*

Proof. Every $a \in V(T)$ corresponds to a maximal clique of G. We categorise maximal cliques of G in the following manner.

1. Maximal clique $C \in \mathcal{C}$ contains a simplicial vertex $v \in V(G)$: Let $P_v = (a, a)$ be the path corresponding to v where $a \in V(T)$ is the node corresponding to C. Then, $\texttt{lca}(a, a) = a$ and the statment follows.

2. Maximal clique $C \in \mathcal{C}$ does not contain a simplicial vertex: Since C is a maximal clique, $V(C) \not\subseteq V(C')$ for $C' \in \mathcal{C}$ and $C \neq C'$. Let $a \in V(T)$ be the node corresponding to C. If all the vertices of C correspond to paths that contain $\texttt{parent}(a)$ then $C \subseteq C'$ where C' is the maximal clique corresponding to $\texttt{parent}(a)$. Thus, at least one of the following must be true:

   (a) there is a path containing a that starts at a descendant of a and ends at another descendant of a, or

   (b) there is a path that starts at a and ends at a descendant of a.

   Let that path be $P = (l, r)$. Then, $\texttt{lca}(l, r) = a$.

   $\square$

Adjacency query. The adjacency query of Algorithm 4 takes the index i, j of paths $P, Q \in \mathcal{P}$ and the space-efficient representation constructed in Section 5.1 as input and checks if $P_i, P_j \in \mathcal{P}$ have a non-empty intersection.

Algorithm 4: For path graph $(T, \mathcal{P})$ and two paths $P_i, P_j \in \mathcal{P}$, the function checks if $V(P_i) \cap V(P_j) \neq \phi$.

1 Function adjacency(i, j):
2 $l = \texttt{getMinLevel}(i, j)$
3   **if** $l \neq 0$ **then**
4 $\{u_1, u_2\} \leftarrow \texttt{getVertices}(i, l)$
5 $\{v_1, v_2\} \leftarrow \texttt{getVertices}(j, l)$
6   **if** *for any pair* $\{a, b\} \in \{\{u_1, v_1\}, \{u_1, v_2\}, \{u_2, v_1\}, \{u_2, v_2\}\}$
              $\texttt{adjacentIG}(a, b, l)$ *is true* **then**
7   **return** true

8   **return** false

Lemma 46. *Given path indices* $i, j \in [n]$ *and the space-efficient representation as input, the function* adjacency(i, j) *checks if paths corresponding to i and j have a non-empty intersection in constant time.*

Proof. Due to Lemma 43 it is only required to check if P_i and P_j intersect in level $\texttt{getMinLevel}(i, j)$. If $\texttt{getMinLevel}(i, j) \neq 0$ then in Line 6 of Algorithm 4 we check if any one of the four pairs of vertex labels paths P_i and P_j in interval graph U_l are adjacent. The vertex labels for paths P_i and P_j in U_l are obtained using the function $\texttt{getVertices}(i, l)$ and $\texttt{getVertices}(j, l)$, respectively in Lines 4 and 5. The adjacency check in the interval graph is done using the function $\texttt{adjacenctIG}$ in Line 6. Since $\texttt{getMinLevel}$, $\texttt{getVertices}$ and $\texttt{adjacenctIG}$ are constant time functions adjacency check can be completed in constant time. $\square$

Neighbourhood query. The neighbourhood query can be implemented as shown in Algorithm 5. It takes the path index and the space-efficient representation constructed in Section 5.1 as input and lists all the paths that have a non-empty intersection with the input path.

Lemma 47. *Given the space-efficient data structure for G and the index i of path* $P \in \mathcal{P}$ *as input,* neighbourhood(i) *returns the neighbours of P in* $O(d)$ *time where d is the degree of P.*

Proof. We will prove that Algorithm 5 enumerates neighbours of P at least once and at most a constant number of times. It follows from Lemma 37 that intersecting paths are of two types, namely, ones with no common light edge and ones with at least one light edge. We have the following cases.

1. Neighbours that share no light edge with P: Let Π be the set of heavy sub-paths of P. From Lemma 44, neighbours with no common edges with P are characterised by $\texttt{lca}(s, t) \in V(\pi)$ and/or $\texttt{lca}(l, r) \in V(\pi)$ where $\pi \in \Pi$. In Lines 5 to 10 and Lines 15 to 20 of Algorithm 5, the paths with lca in any node $u \in V(\pi)$ are added to N_i using the function $\texttt{getPathsLCA}$. Further, in Lines 11 to 14, neighbours of P, for instance, Q such that $V(P) \cap V(Q) \cap V(H) \neq \phi$ such that $H \in \mathcal{H}$ and $\texttt{lca}(l, r) \in V(H)$, are

---

Algorithm 5: For space-efficient representation of path graph $(T, \mathcal{P})$ and an input path $P_i \in \mathcal{P}$, the function returns its neighbours.

---

1 Function neighbourhood(i):

2   Set N_i, E_1, E_2 to NULL

3    $[l, r] \leftarrow$ getEndPoints(i)

4    $p \leftarrow$ lca(l, r)

5   **while** $l \neq p$ **do**

6   Add getPathsLCA(l) to N_i

7       $c \leftarrow$ parent(l)

8   **if** getHeavyPath(l) $\neq$ getHeavyPath(c) **then**

9   Add $\{c, l\}$ to end of E_1

10      $l \leftarrow c$

11    $h \leftarrow$ getHeavyPath(l)

12    $L \leftarrow$ getLevel(h)

13    $\{v_1, v_2\} \leftarrow$ getVertices(i, L)

14   Add neighbourhoodIG(v_1, L) to N_i

15   **while** $r \neq p$ **do**

16   Add getPathsLCA(r) to N_i

17       $c \leftarrow$ parent(r)

18   **if** getHeavyPath(r) $\neq$ getHeavyPath(c) **then**

19   Add $\{c, r\}$ to end of E_2

20      $r \leftarrow c$

21   Concatenate E_2 to the end of E_1 and assign it to E

22    $e' \leftarrow$ NULL

23   **foreach** $e = (w_1^e, w_2^e)$ *in* E **do**

24   **if** $e' =$ NULL **then**

25   Add getDistinctPaths(w_1^e, w_2^e, NULL) to N_i

26   **else**

27   Add getDistinctPaths($w_1^e, w_2^e, w_2^{e'}$) to N_i

28      $e' \leftarrow e$

---

added to N_i. Thus, neighbours that share no light edge with P will be counted at least once. A path Q that has lca in $V(\pi)$ for a $\pi \in \Pi$ such that lca(l, r) $\in V(\pi)$ will be counted at most twice.

2. Neighbours that share at least one light edge with P: Let $p =$ lca(l, r). In Lines 23 to 28 of Algorithm 5, the light edges that are encountered as we traverse from l to p and r to p, respectively, are considered. getDistinctPaths is used to add paths that contain these light edges to N_i. getDistinctPaths do not repeat paths that are counted on light edges already visited as P is traversed from l to p. Also, getDistinctPaths do not repeat paths that are counted on light edges already visited

as P is traversed from r to p. Thus, every neighbour sharing a light edge with P is counted exactly once.

Some neighbours of P can share a light edge with it and also satisfy, for some $\pi \in \Pi$, $\mathtt{lca}(s, t) \in V(\pi)$ or $\mathtt{lca}(l, r) \in V(\pi)$. In this case too, they will be over-counted at most a constant number of times.

Functions $\mathtt{getEndPoints}$, $\mathtt{lca}$, $\mathtt{getPathsLCA}$, $\mathtt{getHeavyPath}$, $\mathtt{getLevel}$, $\mathtt{neighbourhoodIG}$, and $\mathtt{getDistinctPaths}$ are constant time functions. Loops at Line 5 and 15 repeat a maximum of $O(d)$ times since as per Lemma 45, each node in $V(P)$ is a maximal clique that contributes at least one distinct neighbour. By the same argument, the loop at Line 23 repeats $O(d)$ times as the number of edges in P is $O(d)$. Hence, the time complexity of the neighbourhood query is $O(d)$. □

Degree query. The degree of each vertex can be stored using $n \log n$ bits and the degree query can be solved in constant time.

Proof of Theorem 2. Lemma 42 shows that there exists an $O(n \log^2 n)$ bit space-efficient data structure for path graphs. Given this representation as input Lemma 46 shows that adjacency between vertices can be checked in constant time. Similarly, using this representation, Lemma 47 shows an $O(d)$ neighbourhood query. Also, degree query is satisfied in constant time by accessing it from an array. Thus, we conclude Theorem 2. □

6 Conclusion

In this work, we designed efficient data structures for path graphs. In the future, we believe some of the following directions would be interesting to explore regarding path graphs.

1. The best implementations of BFS and DFS are of significant interest as many other problems for path graphs use them as subroutines. In the work by Acan et al. [4], for interval graphs we can see that the representation permits very efficient BFS and DFS algorithms. Can we perform BFS/DFS efficiently on path graphs assuming our representation?

2. Can we show time/space trade-off lower bounds for our data structures? More specifically, can we prove tight space lower bound of redundancy with respect to query time?

3. Are there other graph classes amenable to our techniques for designing succinct data structures?

Reference

[1] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, p. 114–122, 1981.

[2] V. Mäkinen and G. Navarro, "Rank and select revisited and extended," *Theoretical Computer Science*, vol. 387, no. 3, pp. 332–347, 2007.

[3] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, North-Holland Publishing Co., NLD, 2004.

[4] H. Acan, S. Chakraborty, S. Jo, and S. R. Satti, "Succinct data structures for families of interval graphs," *WADS*, vol. 11646, 2019.

[5] S. Chakraborty and K. Sadakane, "Indexing graph search trees and applications," in *44th MFCS*, 2019, pp. 67:1–67:14.

[6] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses and static trees," *SIAM J. Comput.*, vol. 31, no. 3, pp. 762–776, 2001.

[7] L. C. Aleardi, O. Devillers, and G. Schaeffer, "Succinct representations of planar maps," *Theor. Comput. Sci.*, vol. 408, no. 2-3, pp. 174–187, 2008.

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

[9] A. Farzan and J. I. Munro, "Succinct encoding of arbitrary graphs," *Theor. Comput. Sci.*, vol. 513, pp. 38–52, 2013.

[10] J. I. Munro and K. Wu, "Succinct data structures for chordal graphs," in *ISAAC*, 2018, vol. 123 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 67:1–67:12.

[11] F. Gavril, "A recognition algorithm for the intersection graphs of paths in trees," 1978.

[12] C. L. Monma and V. K.-W. Wei, "Intersection graphs of paths in a tree," *J. Comb. Theory, Ser. B*, vol. 41, no. 2, pp. 141–181, 1986.

[13] Reinhard Diestel, *Graph Theory*, Springer Publishing Company, Incorporated, 5th edition, 2017.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd edition, 2009.

[15] R. Grossi and G. Ottaviano, "Fast compressed tries through path decompositions," *ACM J. Exp. Algorithmics*, vol. 19, Jan. 2015.

[16] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter, "On searching compressed string collections cache-obliviously," 2008, PODS '08, p. 181–190, ACM.

[17] G. Navarro and K. Sadakane, "Fully functional static and dynamic succinct trees," *ACM Trans. Algorithms*, vol. 10, no. 3, May 2014.

[18] R. Raman, V. Raman, and S. R. Satti, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," *ACM Trans. Algorithms*, vol. 3, no. 4, pp. 43, 2007.

[19] G. Navarro, *Compact Data Structures - A Practical Approach*, Cambridge University Press, 2016.

[20] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao, "Rank/select operations on large alphabets: A tool for text indexing," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, USA, 2006, SODA '06, p. 368–373, Society for Industrial and Applied Mathematics.

Data Structure	Query	Functionality	To store	Ref
Ordinal trees	$\mathtt{lca}(u,v)$	returns lowest common ancestor of nodes u and v	the clique tree in Sections 4 and 5	[17]
	$\mathtt{parent}(u)$	returns parent of node u		[17]
	$\mathtt{first\_child}(u)$	returns first child of node u		[17]
	$\mathtt{rmost\_child}(u)$	returns rightmost leaf of the sub-tree rooted at u		[17]
	$\mathtt{child\_rank}(u)$	returns the number of siblings to the left of u		[17]
Bit string	$\mathtt{rank}(B,b,i)$	returns the number of bit b's up to and including position i on bit vector B from left	for instance, the BP representation of clique tree	[18]
	$\mathtt{select}(B,b,i)$	returns the position of the $i-$th bit b in the bit vector B from left		[18]
Increasing number sequence	$\mathtt{accessNS}(B,i)$	returns the $i-$th number in the sequence B	the starting nodes of paths in clique tree in Section 4	Section 2.8 of [19]
Wavelet tree	$\mathtt{access}(S,c)$	returns the $y-$coordinate of the point with $x-$coordinate value c stored in wavelet tree S	the paths as points in a two dimensional grid in Section 4	[2]
	$\mathtt{select}(S,[i,i'],[j,j'])$	returns the points in the input range $[i,i'] \times [j,j']$		[2]
	$\mathtt{count}(S,[i,i'],[j,j'])$	returns the number of points in the input range $[i,i'] \times [j,j']$		[2]

Table 1: Summary of the data structures used. Note that $b \in \{0,1\}$.

		Condition	Range
$i = 1$	R_1	$a > 1$	$R_1(1) = [1, a_1 - 1]$
		$a = 1$	$R_1(1) = \texttt{NULL}$
	R_2		$R_2(1) = [a_1, b_1]$
	R_3	$\texttt{succ}(\pi_1, 1) \neq \texttt{NULL}$	$R_3(1) = R_3^1(1) \cup R_3^2(1)$ 1. $R_3^1(1)$: $R_3^1(1) = [b_1 + 1, a_2 - 1]$. 2. $R_3^2(1)$: If $\texttt{rmost\_leaf}(a_2) \neq \texttt{rmost\_leaf}(b_1)$ then $R_3^2(1) = [\texttt{rmost\_leaf}(a_2) + 1, \texttt{rmost\_leaf}(b_1)]$ else $R_3^2(1) = \texttt{NULL}$.
		$\texttt{succ}(\pi_1, 1) = \texttt{NULL}$	If $\texttt{rmost\_leaf}(a_2) \neq \texttt{rmost\_leaf}(b_1)$ then $R_3^2(1) = [\texttt{rmost\_leaf}(a_2) + 1, \texttt{rmost\_leaf}(b_1)]$ else $R_3^2(1) = \texttt{NULL}$
	R_4	$\texttt{succ}(\pi_1, 2) \neq \texttt{NULL}$	$R_4(1) = R_4^1(1) \cup R_4^2(1)$ 1. $R_4^1(1)$: If $\texttt{rmost\_leaf}(b_1) + 1 \neq a_t$ then $R_4^1(1) = [\texttt{rmost\_leaf}(b_1) + 1, a_t - 1]$ else $R_4^1(1) = \texttt{NULL}$. 2. $R_4^2(1)$: If $\texttt{rmost\_leaf}(a_t) \neq \texttt{rmost\_leaf}(a_1)$ then $R_4^2(1) = [\texttt{rmost\_leaf}(a_t) + 1, \texttt{rmost\_leaf}(a_1)]$ else $R_4^2(1) = \texttt{NULL}$.
		$\texttt{succ}(\pi_1, 2) = \texttt{NULL}$	If $\texttt{rmost\_leaf}(a_1) \neq b_1$ then $R_4(1) = [\texttt{rmost\_leaf}(b_1) + 1, \texttt{rmost\_leaf}(a_1)]$ else $R_4(1) = \texttt{NULL}$

Table 2: Ranges for heavy sub-path $\pi_1 \in \Pi$.

		Condition	Range
$i \neq 1$	R_1		$R_1(i) = \texttt{NULL}$
	R_2		$R_2(i) = [a_i, b_i]$
	R_3	$\texttt{succ}(\pi_i, 1) \neq \texttt{NULL}$	$R_3(i) = R_3^1(i) \cup R_3^2(i)$ 1. $R_3^1(i)$: $R_3^1(i) = [b_i + 1, a_{i+1} - 1]$. 2. $R_3^2(i)$: If $\texttt{rmost\_leaf}(a_{i+1}) \neq \texttt{rmost\_leaf}(b_i)$ then $R_3^2(i) = [\texttt{rmost\_leaf}(a_{i+1}) + 1, \texttt{rmost\_leaf}(b_i)]$ else $R_3^2(i) = \texttt{NULL}$.
		$\texttt{succ}(\pi_i, 1) = \texttt{NULL}$	If $\texttt{rmost\_leaf}(b_i) \neq \texttt{rmost\_leaf}(b_i)$ then $R_3(i) = [b_i + 1, \texttt{rmost\_leaf}(b_i)]$ else $R_3(i) = \texttt{NULL}$
	R_4	$\texttt{rmost\_leaf}(a_i) \neq b_i$	$R_4(i) = [\texttt{rmost\_leaf}(b_i) + 1, \texttt{rmost\_leaf}(a_i)]$
		$\texttt{rmost\_leaf}(a_i) = b_i$	$R_4(i) = \texttt{NULL}$

Table 3: Ranges for heavy sub-path $\pi_i \in \Pi$. Note that for $i \neq 1$, $\texttt{succ}(\pi_i, 2) = \texttt{NULL}$.