Algorithmica (2018) 80:2106–2131
https://doi.org/10.1007/s00453-017-0365-6

Compact Representation of Graphs of Small Clique-Width

Shahin Kamali[1]

Received: 6 July 2016 / Accepted: 10 August 2017 / Published online: 1 September 2017

Abstract The notion of clique-width for graphs offers many research topics and has received considerable attention in the past decade. A graph has clique-width k if it can be represented as an algebraic expression on k labels associated with its vertices. Many computationally hard problems can be solved in polynomial time for graphs of bounded clique-width. Interestingly also, many graph families have bounded clique-width. In this paper, we consider the problem of preprocessing a graph of size n and clique-width k to build space-efficient data structures that support basic graph navigation queries. First, by way of a counting argument, which is of interest in its own right, we prove the space requirement of any representation is $\Omega(kn)$. Then we present a navigation oracle which answers adjacency query in constant time and neighborhood queries in constant time per neighbor. This oracle uses $O(kn)$ space (i.e., $O(kn)$ bits). We also present a degree query which reports the degree of each given vertex in $O(k \log^*(n))$ time using $O(kn \log^*(n))$ bits.

Keywords Clique-width · Navigation oracles · Compact representation

1 Introduction

Graphs of very-large sizes have become increasingly important sources of data in recent years. A natural question is how to represent and store these graphs efficiently. Ideally, a graph is stored in a compressed form and, at the same time, it is possible to run queries about it. A data structure is said to be *compact* if it represents a family of data objects (here, graphs) using space $O(\chi)$, where χ is the information theory

✉ Shahin Kamali
shahin.kamali@umanitoba.ca

1 Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada

lower bound for representing members of such family, i.e., a minimum number of bits required to distinguish members of this family modulo their isomorphism. Meanwhile, a compact data structure should make possible to answer certain queries about the data object without decompressing it. A compact structure is *succinct* if it represent a member of the family in $\chi + o(\chi)$ bits.

Random graphs are highly incompressible: the information theory lower bound for representing a random graph with n vertices and at least m edges is $\Theta(m \log(n^2/m))$, which is quite high and can be matched by different adjacency lists [3,22]. One important observation is that the graphs that appear in practice have certain structures which distinguish them from random graphs. Therefore, one can hope to provide compact data structures to represent real-world graphs. Toward this goal, space-efficient representation of graphs with various combinatorial structures has been the subject of research studies in the past decades (see [9, Chapter 5] for a review). Blelloch and Farzan [4] present a compact representation of separable graphs which supports adjacency, degree and neighborhood queries in constant time. The storage requirement is optimal for all monotone class of these graphs (a monotone class of graphs is defined by a monotone property, i.e., a property that holds for every subgraph of a graph which has that property). Similarly, there are succinct representations for partially ordered set (posets) which answer basic queries in constant time [10,18]. One approach for compact representation of graphs is to consider graphs with certain combinatorial or structural properties, e.g., chromatic number or tree-width, and represent graphs in a space that depends on the size of the graph and its given parameter. For example, there is a succinct representation of graphs of size n and tree-width k which needs $kn + o(kn)$ bits and supports navigation and distance queries in constant time [11].

In this paper, we consider the clique-width parameter for graphs. A graph has clique-width k if it can be denoted by an algebraic expression using k labels (a formal definition can be found in Sect. 2). A complete graph can be constructed with two labels and hence has clique-width two. A complete bipartite graph also has clique-width two. Interestingly, trees have clique-width three and any graph of tree-width k has clique-width at most $3 \cdot 2^{k-1}$ [5]. This implies that graphs of bounded clique-width include a large spectrum of sparse and dense graphs, which include bounded-treewidth graphs and other graph families such as cographs [7], distance-hereditary graphs [13] and partner-limited graphs [21]. Many problems that are generally NP-hard can be solved in polynomial time for graphs of bounded clique-width [6]. For example, every graph property that can be expressed in monadic second-order logic has a linear-time algorithm for graphs of bounded clique-width, assuming the clique-width term is given [6]. This makes graphs of bounded clique-width particularly interesting for further studies.

Implicit representations of graphs of bounded clique-width are previously studied [8,16,20]. In these representations, vertices are assigned labels which encode the structure of the graph. The adjacency of any two vertices can be checked by looking at these labels. More complicated lables enable answering distance queries and in general deciding any graph property expressible in a *fixed* monadic second order logic taking as argument a fixed number of sets of vertices [8]. Unfortunately, in these representations, the total space is not linear for graphs of bounded clique-width and the time complexity of answering queries is not constant either.

In this paper, we consider the problem of preprocessing a graph of size n and clique-width k to build space-efficient encodings to efficiently answer the following queries:

- Adjacency query: given two vertices, indicate whether they are neighbors.
- Neighborhood query: given a vertex, report the set of its neighbors.
- Degree query: given a vertex, report the number of its neighbors.

By way of a counting argument, we show that $\Omega(kn)$ bits are required to represent graphs of size n and clique-width at most k (up to isomorphism). Then we present navigation oracles which can be stored in asymptotically optimal space. These oracles answer adjacency queries in constant time. Reporting the set of neighbors of any given vertex can be done in constant time per neighbor. For the mentioned queries, our oracle has size $O(kn)$. Reporting the degree of each vertex can be done in $O(k\log^*(n))$ time with an oracle of size $O(kn\log^*(n))$. We assume the standard word RAM model where a word is $\Omega(\log(n))$ bits wide. This is a realistic assumption commonly made in word RAM algorithms and succinct data structures [17]. We essentially assume that a word of RAM is wide enough so that a vertex can be distinguished by a label that fits in a word and can be read in constant time. Graphs that we consider are unlabeled, unweighted, and undirected. We note that our oracles still work for directed graphs with the exception that reporting outgoing/ingoing neighbors of a given node might take more than constant time per neighbor.

2 Preliminaries

2.1 Clique-Width Decomposition and Union Trees

We formally define the notion of clique-width of a graph and introduce some basic concepts associated with it.

Definition 1 [15] The clique-width of a graph G is the minimum number of labels needed to construct G using the following operations:

- Creation of a new vertex v with label i (denoted by $i(v)$)
- Disjoint union of two labeled graphs G and H (denoted by $G \oplus H$)
- Joining by an edge every vertex labeled i to every vertex labeled j (denoted by $\eta(i, j)$), where $i \neq j$
- Renaming label i to label j (denoted by $\rho_{i\to j}$)

Throughout, we assume k denotes an upper bound for the clique-width of an input graph. Every graph of clique-width at most k can be defined by an algebraic expression using k labels. As an example, consider graph G of Fig. 1a which has clique-width three. One algebraic expression for G is $P = \eta_{1,3}(\eta_{1,2}(P_1 \oplus P_2))$, where

$$P_1 = \rho_{3\to 1}(\rho_{3\to 2}(\eta_{2,3}(\eta_{1,3}(\eta_{1,2}(1(a) \oplus 2(b)) \oplus 3(c)))) \oplus 3(d)), \text{ and}$$
$$P_2 = \rho_{1\to 3}(\eta_{2,3}(\eta_{2,3}(\eta_{1,3}(\eta_{1,2}(1(e) \oplus 2(f)) \oplus 3(g))) \oplus 3(h)))$$

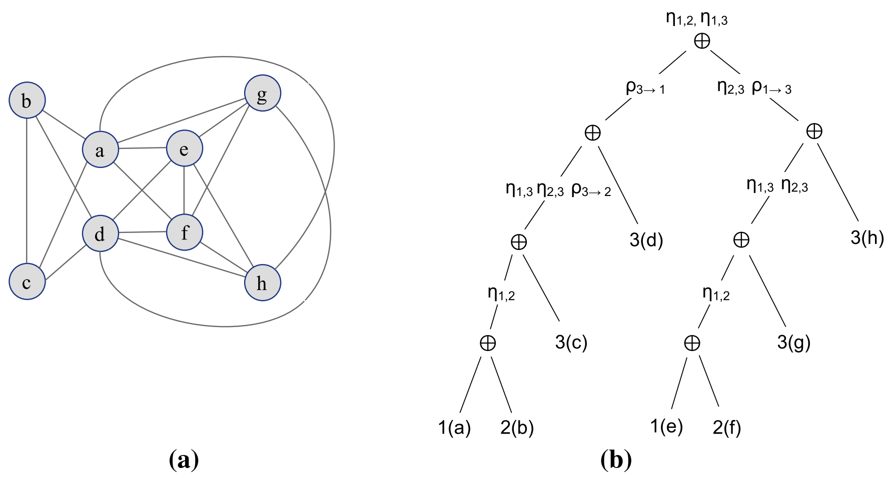

Fig. 1 A graph G of clique-width $k = 3$ and a union tree associated with it. **a** Graph G, **b** a union tree of G

The algebraic expression of a graph can be mapped to a rooted binary tree in which each leaf corresponds to a vertex of the graph and each internal node is associated with a union, join, or rename operation. Note that internal nodes associated with union operations have two children while others have one child. For our purpose, we let each internal vertex x correspond to a union operation, and associate with it a *sequence* $S(x)$ of join and rename operations that are performed before the next union operation. We call this tree a *union* tree of the graph, and define the *width* of such tree to be the number of labels used in the expression. Note that each internal node in a union tree has exactly two children and is associated with a (possibly empty) sequence of rename/join operations that follow the union operation. Figure 1b shows a union tree of the algebraic expression of the graph of Fig. 1a.

Consider a union tree T for a graph G. We say T is *proper* if the following holds for any vertex $a \in T$. Consider the graph G_a formed by the operations in the subtree T_a rooted at a (including the operations in $S(a)$). If T is a proper tree for G, then G_a is the same as the subgraph of G induced by vertices that are created in T_a. As an example, consider the union tree of the graph of Fig. 1. This tree is not proper because in the graph formed by the operations in the left subtree of the root, vertex d is isolated. On the other hand, in the subgraph induced by the vertices created in the same subgraph (a, b, c, d), d is connected to b and c.

Lemma 1 *Any union tree T of a graph can be modified to become a proper union tree of the same graph without increasing its width.*

Proof For a node $x \in T$, let T_x denote the subtree of T rooted at x and let G_x denote the subgraph of G formed by operations in the subtree rooted at x. Assume there are two vertices u and v created in T_x that are connected in G but not in G_x. Let l_u and l_v be labels of u and v after applying all operation in T_x (including the ones in $S(x)$). We add operation η_{l_u,l_v} at the end of the sequence $S(x)$. Clearly, this adds the missing

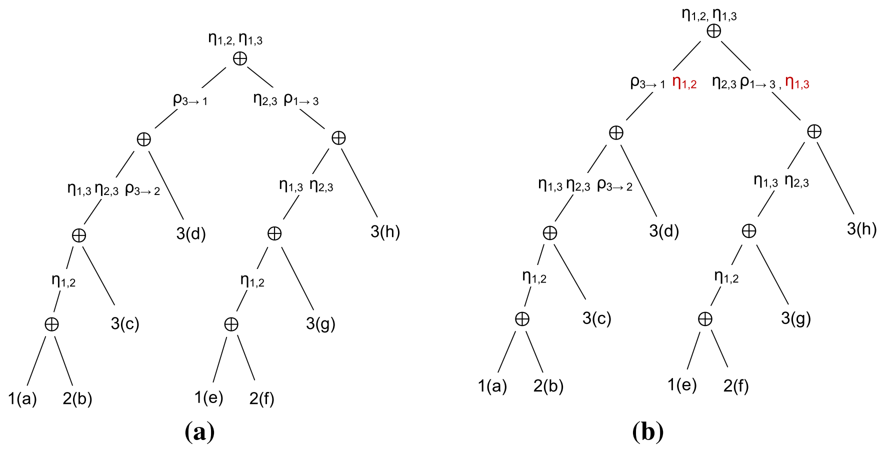

Fig. 2 Converting a union tree to a proper union tree. Let x be the left child of the root in (**a**). Note that d is connected to b and c in G while it is isolated in G_x. Since b, c have label 2 and d has label 1 at the end of $S(a)$, we add a join operation $\eta_{1,2}$ at the end of $S(x)$ in the new tree in (**b**). Similarly, we add a new operation at the end of the sequence associated with the right child of the root so that induced by (e, f, g, h) be the same as the graph created by operations in the right subtree of the root. The added operations are *highlighted* in (**b**). **a** The union tree of Fig. 1, **b** the equivalent proper union tree

edge between u and v. Moreover, this does not change the graph that is eventually formed by the operations in T. This is because all vertices of label l_u are connected to all vertices of label l_v in G. This holds since u and v are connected in G and at some ancestor node of x in T they are joined. As a result, adding the join operation will create edges between vertices that will be eventually connected in G. This way, we update T so that the join operation is applied a bit earlier to some of the neighboring vertices in G. Figure 2 illustrates this for the union tree of Fig. 1. $\square$

2.2 Succinct Structures

Consider a string S of length n over an alphabet Σ of cardinality σ. Query $rank_S(i, c)$ asks for the number of occurrences of c before position i, while $select_S(i, c)$ asks for the index of the ith occurrence of c in S ($c \in \Sigma$). There are succinct data structures for rank/select that store the sequence in $n \log(\sigma) + o(n)$ bits and supports rank/select in constant time, assuming that σ is constant [2, 14].

A balanced parentheses sequence is a sequence in which each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. Such sequence can be created using a formal grammar with production rules $P \rightarrow (P) \mid PP \mid \epsilon$. An ordered tree of n nodes is equivalent to a balanced parenthesis sequence of length $2n$. These trees can be stored in a succinct manner using $2n + o(n)$ bits in a way that many queries are supported in constant time. In particular, we are interested in the following queries:

- $rank(i,$'('$)$, $rank(i,$')'$)$, $select(i,$'('$)$, $select(i,$')'$)$: rank/select operations on parenthesis (as defined before)
- $pre\text{-}rank(i)$: pre-order rank of a given node at index i

- $pre\text{-}select(i)$: selecting a node with a given pre-order rank
- $parent(i)$: parent of a given node
- $lca(i, j)$: least common ancestor of two nodes
- $depth(i)$: distance of a node from root
- $level\text{-}ancestor(i, d)$: ancestor of a node at a given depth
- $child(i, q)$: the qth child of a node i
- $child\text{-}rank(i)$: the number of siblings to the left of node i
- $leaf\text{-}select(i)$: the ith leaf of the tree
- $leaf\text{-}rank(i)$: the number of leaves that appear before node i
- $lmost\text{-}leaf(i)/rmost\text{-}leaf(i)$: the leftmost/rightmost leaf of node i

Lemma 2 *[19] It is possible to store a balanced parenthesis sequence of length* $2n$*, equivalently an ordered tree on n nodes, in* $2n + o(n)$ *bits so that all above operations can be performed in constant time.*

Multiple parenthesis sequences are a generalization of balanced parenthesis sequences in which there are more than one parenthesis types; for example ([]([]())) is a balanced parenthesis sequence with two parenthesis types. In this paper, we are interested only in multiple parenthesis with two types, which are equivalent to ordered trees in which each node is given a label of '0' or '1'. Let $m\text{-}enclose(i, k)$ denote a query which gives the position of the closest matching parenthesis of type k which encloses i. We use the following result:

Lemma 3 *[1] A multiple parenthesis sequence of* $2n$ *parenthesis of two types can be stored using* $4n + o(n)$ *bits to support* $m\text{-}enclose(i, k)$ *in constant time.*

3 Lower Bound

To the best of our knowledge, there is no result that concerns the number of graphs with n vertices and of clique-width at most k. In this section, we present such result and use it to prove a lower bound for the number of bits required for any navigation oracle to represent such graphs. Here, we do not distinguish between isomorphic graphs, and by 'representing' a graph G, we mean representing all graph that are isomorphic to G. It is known that at least $q(n - o(n) - q/2)$ bits are required to represent graphs of size n and tree-width q, modulo their isomorphism [11]. Any of these graphs have a clique-width of at most $3 \cdot 2^{q-1}$ [5]. We conclude that $\Omega(n \log(k))$ bits are required to represent a graph of clique-width at most k. Here we improve this lower bound to $\Omega(kn)$.

We introduce a family of graphs named *layer graphs* and use a counting argument to prove a lower bound for the number of these graphs. Meanwhile, we show that the clique-width of these graphs is bounded and hence deduce a lower bound for the number of graphs of a given clique-width. Layer graphs are formed by smaller components named 'shell graphs'.

Definition 2 A *shell graph* of size m is constructed by connecting a single vertex to all vertices but one endpoint of a path of size $m - 1$. The vertex of degree $m - 1$ is called the *hub*; the vertex of degree 1 is called the *dangling vertex*; all other vertices are called *layer* vertices.

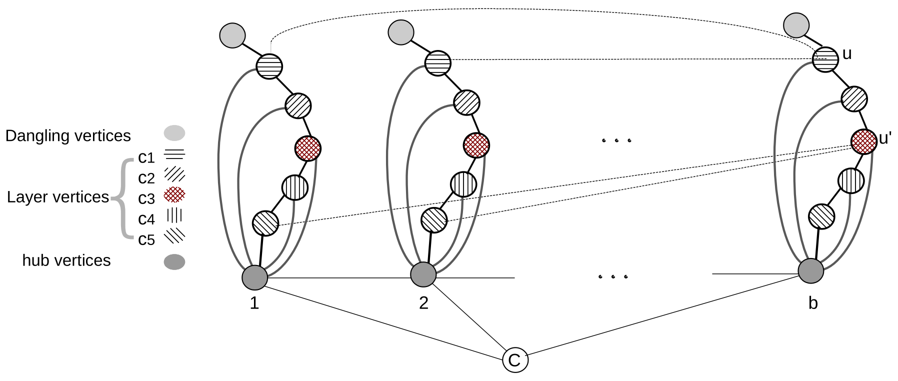

Fig. 3 A layer graph G of width $a = 5$ and length b. The center is vertex C. Shell graphs are ordered from 1 to b. Vertices of these shell graphs are distinguished by their unique *colors*. A layer vertex in a shell graph of order i is connected to either all or none of layer vertices of the same *color* in shell graphs of order $j < i$, e.g., u is connected to all vertices of *color* c_1 and u' is connected to all vertices of *color* c_5 (Color figure online)

Note that no two vertices are isomorphic in a shell graph, i.e., any vertex can be uniquely distinguished from other vertices: the center and dangling vertex can be distinguished by their degree while other vertices are distinguished by their distance from the dangling vertex when the center is removed. As a result, we can assume an implicit coloring of vertices in a shell graph which gives unique colors to all vertices.

Definition 3 A layer graph G_l of width a and length b ($b \geq a$) is constructed by connecting a single vertex, called the *center*, to the hubs of b shell graphs of size $a + 2$. These shell graphs are ordered from 1 to b and there is an edge between hubs of the shell graphs of order i and $i - 1$. In addition to the above structure which is fixed among all layer graphs, there can be edges between layer vertices in shell graphs of different orders which distinguishes them. Let u and v be two layer vertices in the shell graphs of respectively orders i and j so that $j \leq i$. There is an edge between u and v in G_l if and only if there is an edge between u and any layer vertex z of the same color as v in a shell graph of order $j' < i$. In other words, u is connected to all or none of vertices of the same color which belong to shell graphs of order less than i.

Figure 3 illustrates the definition of layer graphs. Note that a layer graph has a ‘fixed’ component formed by b shell graphs of size a and extra edges connecting the center and hubs of these shell graphs. In addition, there is a ‘varying’ component which defines edges between layer vertices of shell graphs of different orders. Because of this component, there are many shell graphs of a given width and length. The following lemma provides a lower bound for the number of layer graphs.

Lemma 4 *There are at least $f(a, b) = 2^{(b-1)a^2-1}$ layer graphs of width a and height b, up to isomorphism.*

Proof Consider layer graphs which share a fixed component formed by b shell graphs of size a and extra edges connecting the center vertex and hubs of these shell graphs. We count the number layer graphs which share this fixed component. In these graphs, each layer vertex u in a shell graph of order $i \geq 2$ is connected to all or none of the layer vertices of the same color and in shell graphs of order less than i. Since there are a colors for layer vertices, there are a possibilities for edges between u and layer vertices in shell graphs of order less than i. This gives a total of a^2 possibilities between vertices of shell graph of order i and those of smaller orders. Summing over all values of $i \geq 2$, the number of possibilities for edges between layer vertices is $(b-1)a^2$. This gives a total number of $2^{(b-1)a^2}$ graphs. So, given a fixed center and an ordering of shell subgraphs, there are at least $2^{(b-1)a^2}$ different layer graphs.

In a layer graph, it is possible to distinguish the center from other vertices as the only vertex of degree at least a (note that $b \geq a$). Moreover, there are two ways to order shell subgraphs (from one endpoint to another endpoint of the path induced by hubs of these graphs). So, the total number of shell graphs of width a and height b is at least $2^{((b-1)a^2)-1}$.

Lemma 5 *Any layer graph G of width a and length b has clique-width at most $a+6$.*

Proof Let G' be a copy of G that excludes the center. We show that G' can be constructed using $a+5$ labels. At the end of the construction for G', there is a label h_p so that all hubs, and only hubs, have label h_p. The center is created using a single label c and a join operation which connects the center to all hubs.

To construct G', we use an inductive procedure. In the ith iteration of the induction, vertices in the shell graph of order i are added to G' $(1 \leq i \leq b)$. In this construction, we use $a+2$ *permanent* labels $1_p, 2_p, \ldots, a_p, h_p, d$. When a vertex receives a permanent label, its label remains unchanged till the end of construction.

At the beginning of iteration i, each layer vertex of color c_q and order $j < i$ has permanent label q_p. Moreover, all hubs and dangling vertices of order less than i have labels h_p and d, respectively. The $a+2$ vertices in the shell graph of order i are created using three *temporary* labels 1_t, 2_t, and h_t. At the beginning of iteration i, the hub is created using label h_t. Other vertices of order i are added one by one, starting from the dangling vertex followed by vertices of colors $c_1, \ldots, c_a$ ordered by their distance from the dangling vertex in their induced path. In this ordering, a new vertex u is created with label 1_t, while the vertex v that is added just before u (in case it exists) has label 2_t; vertices appearing before v have permanent labels, i.e., a dangling vertex has label d and a layer vertex of color x has label x_p. After generating u, two operations η_{1_t,h_t} and $\eta_{1_t,2_t}$ join u with the hub and v. The remaining neighbors of u appear in shell graphs with order less than i. Recall that each layer vertex in a shell graph of order i is connected to all or none of vertices of the same color in shell graphs of order less than i. Therefore, if there is an edge between u and a vertex of color y in a shell graph of order $j < i$, we can use the join operation η_{1_t,y_p} to create edges between u and all vertices of color y which appeared earlier. At this point, edges between u and other added vertices are created. We apply the following rename operations. First, we change the label of v from 2_t to its permanent label. So, if v is a layer vertex of color x, its label is changed from 2_t to x_p (i.e., $\rho_{2_t \to x_p}$); if v is a dangling vertex, its label is changed from 2_t to d (i.e, $\rho_{2_t \to d}$). Next, we change the label of u: if u is the last vertex in the shell graph of order i, we change its label from 1_t to its permanent label a_p (i.e., $\rho_{1_t \to a_p}$). Otherwise, we change the label of u from 1_t to 2_t (i.e., $\rho_{1_t \to 2_t}$). At the end of the iteration, the hub's label is changed from h_t to h_p (i.e., $\rho_{h_t \to h_p}$). This way, all vertices in shell graph of order i receive permanent labels at the end of the ith iteration, and the temporary labels remain unused for the next iteration.

The above procedure uses $a+2$ permanent labels, three temporary labels, and one label for the center. The total number of labels is $a+6$. □

Theorem 1 *For large values of n and any value of $k \geq 9$, there are at least $2^{(k-8)(n-k+3)}$ graphs of size n and of clique-width at most k.*

Proof Consider layer graphs of width $a = k-6$ and height $b = \lfloor (n-1)/(k-4) \rfloor$. These graphs have size $m = ab + 2b + 1$, where $n - k + 4 < m \leq n$. We add $n - m$ isolated vertex to achieve size n. By Lemma 4, the number of such graphs is at least $2^{(b-1)a^2-1}$. For large values of n, we have $4b > a^2$ and we can write $ba^2 - a^2 > ba^2 - 4b = (a-2)(ab+2b) \geq (k-8)(n-k+3) > (k-8)n - k^2$. So, the number of layered graphs of width a and length b is at least $2^{(k-8)(n-k+3)}$. □

Corollary 1 *Consider graphs of size n and clique-width at most $k \geq 9$, where k is a constant independent of n. To represent these graphs at least $(k-8)n - o(n)$ bits are required.*

4 Compact Data Structure

In this section, we present navigation oracles for graphs of size n and clique-width at most k using a space of size $O(kn)$. In Sect. 4.1, we show how to update a union tree of G to a *proper-k-balanced* tree with properties that facilitate encoding the graph. In Sect. 4.2, we encode this proper-k-balanced union tree using succinct structures presented in Sect. 2.2 and show how to answer navigation queries using the encoded structures.

4.1 Proper-*k*-Balanced Union Trees

Let T be an arbitrary union tree of a graph G. We define the *weight* of a node $x \in T$ as the number of leaves in the sub-tree of T rooted at x. We use $|x|$ to denote the the weight of x. For our oracles, we are particularly interested in *k-balanced* union trees. Given a tree T, a node $x \in T$ is said to be *k-balanced* if either $|x| < 3k$ or both children of x have weight at least k. The tree T is k-balanced if all its internal nodes are k-balanced. Note that this definition of k-balanced trees does not imply a logarithmic-depth for them; it basically facilitates partitioning vertices of the graph into groups of 'balanced' sizes in the range $[k, 3k)$ as will be described in Sect. 4.2.

Lemma 6 *Any union tree T of width k for a graph G can be transformed to a k-balanced union tree of width at most $2k$.*

Proof We start from T and modify it to achieve the desired k-balanced tree T' of width at most $2k$. Let $1, 2, \ldots, k$ denotes the labels used in T. In T', there are labels of two *classes*: labels $1, 2, \ldots, k$ which have class 1 and labels $1', 2', \ldots, k'$ which have class 2. We start T' as a copy of T in which all labels have class 1. We process nodes

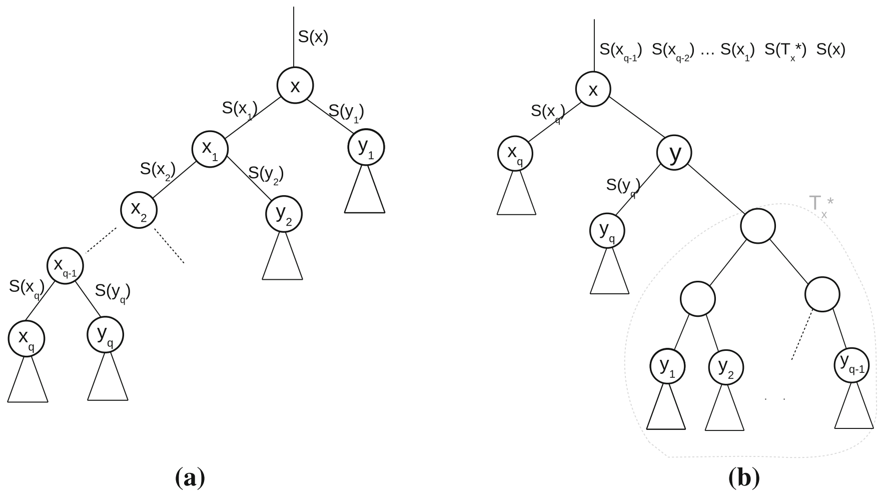

Fig. 4 An illustration of the process to turn node x into a k-balanced node. For the tree in (**a**), we have $|x| \geq 3k$ and $|y_1| < k$. In the new tree in (**b**), vertices created in the subtree T_x^* have distinct labels of class 2. These labels are added in order to avoid operations of $S(x_{q-1}) \dots S(x_1)$ in node x affecting vertices in T_x^* after the union operation at node x. **a** The original tree T, **b** the updated tree T'

in T' in pre-order and update the tree to ensure that visited node is k-balanced (and remains so after any future update). Consider the first node $x \in T'$ in the pre-order traversal of T' which is not k-balanced. W.l.o.g. assume that he subtree on the left of x has weight at least equal to that of the right subtree. We update the tree using the following procedure.

Let $x_1, x_2, \dots, x_m$ be the nodes on the path from x to the left-most leaf in the subtree rooted at x. Let y_i denote the right sibling of x_i ($1 \leq i \leq m$). We form a set R that initially includes vertices in the subtree rooted at y_1. So we have $|R| < k$. We repeatedly add vertices to R until $|R|$ becomes at least k: first we add vertices introduced in the subtree rooted at y_2; this increases $|R|$ by $|y_2|$. In case $|R| < k$, we add vertices introduced in subtree of y_3 to R and continue accordingly. Since T' is a full binary tree and $|x| \geq 3k$, there is $q \leq m$ so that after adding vertices in the subtrees of $y_1, \dots, y_q$, the weight of R becomes at least k.

Provided with R, we update T' in the following manner. The subtree rooted at x_q replaces x_1 as the left child of x. The right child of x becomes a new node y which has the subtree rooted at y_q as one of its children. The other child of y is a complete binary tree T_x^*; leaves of T_x^* are associated with members of R excluding those in the subtree rooted at y_q. Note that there are less than k leaves in T_x^* (since $|R|$ is smaller than k before adding vertices of y_q). We map vertices of R to labels of class 2, i.e., each leaf of T_x^* creates a vertex u with a separate label $L(u)$ of class 2. There is no sequence of join/rename operation on y and nodes of T_x^*. The labels and sequence of join/rename operations for nodes other than x in T' remain the same as T. For node x, the sequence of join/rename operations is $S(x_{q-1})\ S(x_{q-2}) \dots S(x_1)\ S(T_x^*)\ S(x)$. Here $S(T_x^*)$ is a sequence starting with join operations that connect vertices of R with their neighbors in G; since each vertex in R is mapped to a separate label (and only vertices of R have labels of class 2 at node x) a join operation can be associated with each neighboring edge of vertices in R. After these join operations, $S(T_x^*)$ ends with renaming each label $L(u)$ with the label of u in T at node x after applying $S(x)$. Figure 4 provides an illustration of the above process.

Claim 1 The updated tree is a valid decomposition tree for G.

Proof We show that all vertices have the same neighbors in the graphs created by the tree before and after the update. We start with vertices in R, i.e., leaves of the trees rooted at $y_1, \ldots, y_q$. In the updated tree, each of these vertices is created with a separate label of type 2. We know that such vertex is connected to all or none of vertices of the same label in node x_{q-1} of the original tree (this holds by the definition of clique-width decomposition and union trees). Moreover, vertices of the same label at node x_{q-1} of the original tree are of the same label at node x of the new tree; this is because the same sequence $S(x_q) \oplus S(y_q)$ of operations are applied on them when they appear for the first time in both trees. Hence, at the time T_x^* joins vertices of R with vertices of type 1 in the new tree, any vertex $x \in R$ is connected to all or none of vertices of the same label of type 1 in G. In other words, the explicit join operations between vertices of labels of type 1 and 2 in T_x^* connects vertices of R to their neighbors in G created in x_q and y_q. Moreover, the join operations between labels of type 2 explicitly connects each vertex in R with its neighbors in T_x^* (it is possible since there is one label of type 2 for each vertex). So, vertices of R are connected to their neighbors in G in the graph created by the subtree of rooted at node x of the new tree. Note that T_x^* ends by changing the labels of type 2 to their labels at node x of the original tree. Hence, the subsequent operations result in the same graphs in both trees. We conclude that vertices of R have the same neighbors in the graphs associated with the original and new trees. For edges between vertices created at nodes x_q and y_q, we note that the same sequence of operations are applied on these vertices in both trees, namely $(S(x_q) \oplus S(y_q))S(x_{q-1}) \ldots S(x_1)S(x)$. Note that this sequence does not affect vertices in R as they have labels of type 2. We conclude that the updated tree is a valid decomposition tree for G.

Claim 2 After the above update, x is k-balanced in T'.

Proof We prove that both children of x have weight at least k. The right subtree has weight $|R| \geq k$ (by definition of R). For the left subtree, we have $|x| \geq 3k$ (since x was not k-balanced in T), i.e., $|x_q| + |y_q| + (|R| - |y_q|) \geq 3k$; moreover $|R| - |y_q| < k$ (by the way we construct R), and $|x_q| \geq |y_q|$ (by the assumption that the left subtree has larger weight). So we get $|x_q| \geq k$, i.e., the left subtree has weight at least k.

We apply the above procedure on all internal nodes of T' in pre-order to ensure that they are k-balanced. Claim 1 implies that tree the remains a valid decomposition tree after each update, and Claim 2 ensures that the visited node becomes k-balanced after an update. Note that when vertices in a left subtree are transferred to a right subtree, the original labels of vertices in the right subtree are ignored since they are re-introduced with labels of class 2. For example, in Fig. 4a, after processing x, if y is not k-balanced (which happens when $|y| \geq 3k$), in our next update, we must transfer some vertices from y_q to T_x^*. In this case, the labels of vertices in T_x^*, which have type 2, are ignored and new labels of class 2 are used. Regardless, the number of labels of class 2 is no more than k. □

To encode a graph G of clique-width at most k, represented by a union tree of width k, we first apply Lemma 6 to achieve a k-balanced union tree of G of width at most $2k$. Then we apply Lemma 1 to convert this tree into a proper tree. The result is a proper, k-balanced union tree of width at most $2k$. We call this tree a *proper-k-balanced* union tree. Being balanced is useful for storing the tree efficiently, while being proper is essential for answering queries in constant time. We conclude the following theorem.

Theorem 2 *Each graph G of size n and clique-width at most k can be represented by a proper-k-balanced union tree of width at most $2k$.*

In the following sections, in order to represent a graph G, we assume it is described by a proper-k-balanced union tree T, and each vertex in G is represented by its address in T. More precisely, leaves of T are ranked from 1 to n in the pre-order traversal of T, and each vertex is identified by its rank in this order. Therefore, the input to neighborhood and degree queries is a number between 1 and n that identifies a leaf of T. Similarly, the input to an adjacency query is two integers between 1 and n.

4.2 Graph Encoding

In this section, we describe how to store a graph G of width at most $2k$ using $O(kn)$ bits. For that, we introduce an abstract structure called *partition tree* which can be created from the proper-k-balanced tree T of G. In this section, we show that each proper-k-balanced tree can be represented by a partition tree. Later, we will show that a partition tree can be represented by a few succinct components.

Definition 4 A (n, k)-partition tree P is a complete binary tree in which each node is associated with $3k$ labeled 'spots' as well as a 'join graph' of size $3k$. Furthermore, there are n 'vertices' which appear in P in the following manner:

- Each spot of a node of P is a (possibly empty) set of vertices.
- Each vertex in V appears in exactly one spot of one leaf of P.
- Let $V(a)$ denote the set of vertices appearing in all spots of a node $a \in P$. We have:
  - $|V(a)| \geq k$.
  - For the two children b and c of (internal node) a, $V(a) = V(b) \cup V(c)$ and $V(b) \cap V(c) = \Phi$ (empty set).
- If two vertices appear in the same spot in node $a \in P$, they appear in the same spot in the parent of a.

Intuitively, the partition tree P is a copy of T in which groups of leaves are clustered to form a smaller graph. The join graphs in P reflect how labels are joined in T while spots of nodes in P keep track of rename operations in T.

Lemma 7 *Any graph G of size n and clique-width at most k can be represented by a (n, k)-partition tree.*

Proof By Theorem 2, G can be represented by a proper-k-balanced tree of width at most $2k$. We describe how to create the desired partition tree P from T. Intuitively, P is a copy of T where vertices of the same label in a node c of T are placed in the spot of the same label in the projected node c_p of P; join graph of c_p indicate the join operations applied between vertices of different labels in c. In addition, in the partition tree, groups of leaves are clustered to form a smaller graph of size in the range $[k, 3k)$.

Recall that weight $|a|$ of a node $a \in T$ is the number of leaves in the subtree rooted at a. Consider any path from the root to a leaf of T. The weight of nodes in this path is strictly decreasing from the root to the leaf. Consider the first node c of weight less than $3k$ in this path. So, the parent of c has weight at least $3k$ and since T is k-balanced, the weight of c is at least k. This implies that for any path from the root to a leaf of T, there is a node c such that $k \leq |c| < 3k$. In the partition tree P, the vertices created in the subtree rooted at c are presented in a single leaf c_p. Any of these vertices have a label at node c of T. If a vertex has label i in that node, we include it in the spot with label i in c_p ($i \leq 2k$). This way, all vertices are placed in a spot of exactly one leaf of P. Moreover, c_p has a join graph of size $3k$ which precisely indicates whether any two vertices introduced in the tree rooted at c in T are neighbors in G (note that $3k - |c|$ vertices of this graph will be isolated).

After creating the leaves of P as indicated above, the remaining structure of P is copied from T, i.e., any internal node of P is projected to an internal node of T. In what follows, we describe how spots and join graphs of these nodes in P are defined from the join/rename sequences of the projected nodes in T. Let x be an internal node $x \in T$ which has a projection x_p in P. Recall that $S(x)$ ($x \in T$) indicates the sequence of join/rename operations which is applied on vertices created in the subtree rooted at x before the next union operation. Regardless of the order of the operations in $S(x)$, we can represent it by indicating, for each label i, the labels that i is joined with in $S(x)$, and the eventual label after applying $S(x)$. To be more precise, we interpret $S(x)$ in a way that all join operations precede the rename operations (e.g., $\rho_{3\to 1}\ \eta_{1,2}$ becomes $\eta_{1,2}\ \eta_{3,2}\ \rho_{3\to 1}$). In the partition tree, the join operations in $S(x)$ are represented by the join graph in x_p where vertices represent labels in T. There are $3k$ spots in these join graphs, out of which $2k$ spots are associated with the $2k$ labels in T. The remaining k vertices in the join graph are isolated. For any join operation between two labels, there is an edge between the associated vertices in the join graph. This way, we keep track of all join operations in T. Next, assume that a vertex v has label i in a node $y \in T$, and after the sequence $S(y)$ is applied, its label becomes j in the parent of y. To capture this in P, v will appear in the spot j in the parent of y_p. Figure 5 illustrates how a partition tree is constructed from a proper-k-balanced union tree.

It is straightforward to verify that P is indeed a partition tree. First, by the way we selected node $c \in T$ for forming the leaves of P, we know that each vertex appears in exactly one spot of a leaf of P, and for any projected leaf c_p of P, we have $|c_p| \geq k$. Moreover, for any vertex $v \in G$, there is a path from the root to a leaf of P so that v appears in (a spot of) all nodes on this path. Moreover, since spots are associated with labels in T, if two vertices appear in some spot in node $a_p \in P$ (i.e., they have the same label in the associated node $a \in T$), they will appear in the same spot in the parent of a_p (i.e., they will have the same label in the parent of a in T). □

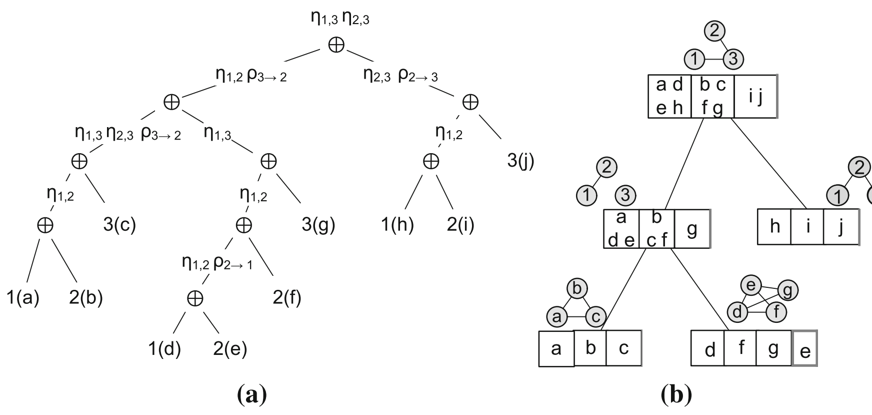

Fig. 5 A proper-k-balanced union tree T of width at most $2k$, where $k = 3$ (**a**) and a partition tree P for T (**b**). Each internal node in P has $3k = 9$ spots and a join graph of the same size. Here, we only depict non-empty spots and non-isolated vertices in the join graphs. **a** A proper-k-balanced union tree T, **b** a partition tree P of T

By the above lemma, to encode a graph G of size n and clique-width at most k, it suffices to encode a (n, k)-partition tree. As mentioned earlier, the partition tree is an abstract structure. In what follows, we describe how to encode a partition tree. First, we bound the number of nodes in a partition tree:

Lemma 8 *The number of nodes in a (n, k)-partition tree is less than $2\lceil n/k\rceil$.*

Proof By definition of partition trees, there are at least k vertices in each node of the tree. Since each vertex appears in exactly one leaf of the tree, there will be at most $\lceil n/k\rceil$ leaves in the tree. Since the tree is a complete binary tree, the number of internal nodes is at most $\lceil n/k\rceil - 1$. □

We store a partition tree P using three succinct components: the 'main tree' which stores the structure of P, the 'join sequence' which stores the join graphs associated with nodes of P, and the 'layer trees' which keep track of vertices in each labeled spot of P. In what follows, we describe these components in details.

– *Main Tree* The main tree of P, denoted by M stores the *structure* of the partition tree. Internal nodes of M form the same tree as the partition tree (in which the spots and join graphs are omitted). For each leaf $x \in P$, there will be $|x|$ leaves in M each representing one of the $|x|$ vertices created in the subtree associated with x in the proper-k-balanced union tree. So, M has n leaves, one associated with every vertex in the graph (see Fig. 6 for an illustration). There are $O(n)$ leaves and $O(n/k)$ internal nodes in M (Lemma 8). As a result, the balanced parenthesis sequence associated with the tree has length $O(n)$ and can be stored in linear space using the structure of Lemma 2. We assume each vertex is *addressed* through its opening parenthesis in the main tree. Recall that the input to navigation queries is the rank of the involved vertices in the pre-order traversal of the proper-k-balanced union tree. Since vertices appear in the same order in the balanced parenthesis sequence,

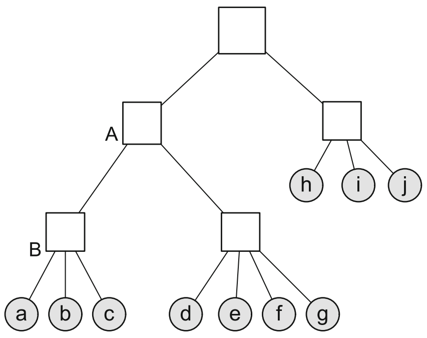

Fig. 6 The main tree M for the partition tree P of Fig. 5b. M stores the binary structure of P; there is a leaf in M for each vertex in the graph

we can use $select_((i)$ on this sequence to find the address of a queried vertex i in constant time.

– *Join Sequence* For each node $x \in P$, we store the join graph of x using an adjacency matrix structure of $O(k^2)$ bits. Be Lemma 8, the partition tree has $O(n/k)$ nodes. The join graph for each node has size $O(k^2)$. So, the total space is $O(kn)$. To be more precise, we use the pre-order traversal of P to sequentially store blocks of fixed size $9k^2$ for any node in P. Note that P is a (n, k)-partition tree and hence the size of the join graph in each node is less than $3k$, which we represent with an adjacency matrix of size $9k^2$. We call the resulting structure the *join sequence*. Since each block has a fixed length, given the pre-order index of an internal node in the main tree, we can find the block associated with the node in constant time.
– *Label Trees* We use label trees to store the spots in which every vertex appears in nodes of P. Recall that each vertex $v \in G$ appears in some labeled spot in every node of a path from a leaf to the root of P. Define the *type* of a vertex as its eventual label in the algebraic expression, i.e., the label of its spot in the root of the partition tree. Note that there are $3k$ possible types for vertices. For each type i, we store a tree, named the *label tree* of type i, that keeps track of labels (spots) of vertices of type i. Each internal node of such tree is associated with a spot in an internal node a in the partition tree and has a *value* that indicates the label of that spot right after the union operation in a. Similarly to the main tree, leaves of label trees are associated with vertices of the graph (see Fig. 7 for an illustration). Note that label trees are not necessarily binary and might have a different structure from that of the main tree. We store each label tree using the balanced parenthesis structure of Lemma 2 to store the structure of the tree and a succinct rank/select structure to store the values of the internal nodes in pre-order. The number of leaves in all label trees is n and the number of internal nodes is equal to the number of spots in the partition tree, i.e., $O(n)$. So, the total size of the structures associated with the label trees is $O(n)$ for storing all trees and $O(n \log(k))$ for storing the values of internal nodes.

Lemma 9 *It is possible to represent a graph G of size n and clique-width at most k using a main tree of size $O(n)$, a join sequence of size $O(kn)$, and $3k$ label trees of total size $O(n \log(k))$.*

Proof By Lemma 7, it is possible to represent G as a (n, k)-partition tree P. As described above, this partition tree can be encoded using a main tree which keeps

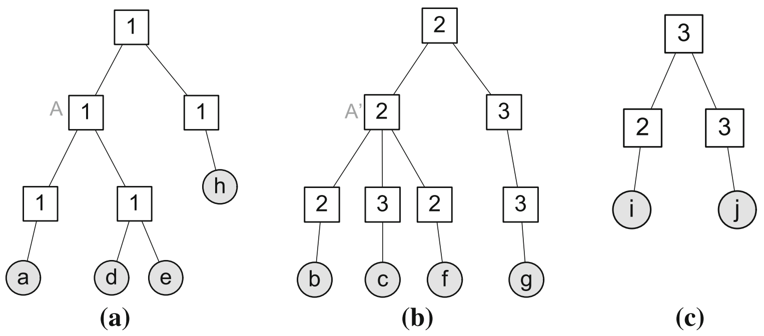

Fig. 7 Label trees of the partition tree P of Fig. 5b. Numbers in an internal node x indicate the label of vertices in the associated spots of P for x. Each vertex in the graph appears as a leaf in exactly one of the label trees. **a** Type 1, **b** type 2, **c** type 3

track of the structure of P, a join sequence which stores join graphs of P, and $3k$ label trees which represent the spots in which vertices of G appear in nodes of P. □

The main tree, join sequence, and label trees are sufficient to represent a partition tree (and consequently the graph associated with it). However, in order to facilitate answering queries, we include two more structures:

– *Type-Lookup Sequences* consider the pre-order traversal of the partition tree (i.e., pre-order traversal of internal nodes in the main tree M). Each node is associated with k spots in the partition tree. We write down the eventual label of these spots (i.e., their labels in the root). In a separate sequence, we write down the type of the leaves of the main tree in the order they appear in the pre-order traversal. The result will be two sequences U_{inner} and U_{leaves}, each of length $O(n)$ over an alphabet of size k, which are associated with the types of internal nodes and leaves, respectively. These two sequences can be stored in $O(n \log(k))$ bits with constant-time support of rank/select operations. Provided with U_{inner} and U_{leaves}, given a leaf or a spot in the main tree, it is possible to *locate* the node associated with it in a label tree in constant time, as will be described later.
– *Skip Trees* for every label tree, we store the same tree but with binary values on nodes which indicate whether there is an edge between vertices in the spot associated with the node and vertices of another spot in the partition tree. There are k skip trees of size $O(n)$ with binary labels on nodes; we store them using multiple balanced parenthesis sequence of Lemma 3 in linear space. Skip trees facilitate reporting neighbors of each vertex in constant time per neighbor.

Table 1 provides a summary of the above structures which together form an oracle of size $O(kn)$ which supports adjacency and neighborhood queries in constant time as will be described.

Recall that, in our navigation queries, vertices are addressed via the index of their opening parenthesis in the main tree M. In addition to M, each vertex u of type u-type is also present at a leaf of the label tree $L_{u\text{-type}}$. In order to answer queries in constant time, we should be able to retrieve address of u in the label tree from its address in the main tree and vice versa:

Table 1 Components that form our compact oracle for supporting adjacency and neighborhood queries

Name	Notation	Succinct representation	Size
*M*ain tree	M	Balanced parenthesis (Lemma 2)	$O(n)$
*J*oin sequence	J	Succinct rank/select seq. (e.g., [2,14])	$O(kn)$
3*k* *L*abel trees	L_1, ..., L_{3k} (structure)	Balanced parenthesis (Lemma 2)	$O(n)))$
	V_1, ..., V_{3k} (values)	Succinct rank/select sequence	$O(n\log(k))$
Type look-*u*p sequences	U_{inner}, U_{leaves}	Succinct rank/select sequence	$O(n\log(k))$
Ski*p* trees	$P_1, \ldots, P_{3k}$	Multiple parenthesis seq. (Lemma 3)	$O(n)$

Lemma 10 *Let u be the index of (the parenthesis representing) a vertex of type u-type in the main tree M. Let u-p be the index of (the parenthesis representing) the same vertex in the label tree $L_{u\text{-type}}$. Provided with u, one can retrieve the pair $(u\text{-type}, u\text{-}p)$ in constant time. Similarly, provided with $(u\text{-type}, u\text{-}p)$, one can retrieve u in constant time.*

Proof Consider we are given index u in the main tree. Using the $leaf\text{-}rank$ operation on the main tree M, we can find the pre-order rank $u\text{-}lr$ of u among all vertices. We retrieve the type of u by checking the type-lookup sequence U_{leaves} at index $u\text{-}lr$. Provided with $u\text{-}type$, we can find its pre-order rank u_{st} among vertices of the same type by applying the $rank$ operation on type-lookup sequence U_{leaves}. Note that $u\text{-}p$ is the address of the u_{st}th leaf in $L_{u\text{-type}}$; so we can retrieve it using $leaf\text{-}select$ operation. Algorithm 1 illustrates the above procedure.

Next, assume we are given $(u\text{-type}, u\text{-}p)$. We use $leaf\text{-}rank$ on tree $L_{u\text{-type}}$ to retrieve u_{st}, which is the pre-order index of u among vertices of the same type. Provided with u_{st} we apply the $select$ operation on the type-lookup table U_{leaves} to retrieve the pre-order $u\text{-}lr$ index of u among all vertices. Provided with $u\text{-}lr$, we use $leaf\text{-}select$ operation on M to retrieve u. Algorithm 2 illustrates this procedure.

By Lemma 2 all above operations on M and $T_{u\text{-}type}$ can be performed in constant time. □

Algorithm 1 project$_{M\rightarrow L}$

```
Input: index $u$ of the parenthesis representing a leaf $u$ (a vertex in $G$) in the main tree $M$
Output: pair $(u\text{-}type, u\text{-}p)$ where $u\text{-}type$ is the type of $u$ and $u\text{-}p$ is the projection of $u$ in $L_{u\text{-}type}$, i.e., the
        parenthesis representing the leaf associated with $u$ in $L_{u\text{-}type}$.
$u\text{-}lr \leftarrow leaf\text{-}rank_M(u)$ /* $u\text{-}lr$ is the index of $u$ among leaves of the main tree $M$ */
$u\text{-}type \leftarrow U_{leaves}[u\text{-}lr]$ /* querying the type-lookup table to find the type of $u$. */
$u\text{-}st \leftarrow rank_{U_{leaves}}(u\text{-}lr, u\text{-}type)$ /* querying the type-lookup sequence to set $u\text{-}st$, which is the index of $u$ among
  leaves of the $s$ame $t$ype. */
$u\text{-}p \leftarrow leaf\text{-}select_{L_{u\text{-type}}}$ /* returning the $st$th leaf in the label tree of $u$ */
return $(u\text{-type}, u\text{-}p)$
```

4.3 Adjacency Queries

Given two vertices u and v, we describe how to report whether there is an edge between them in constant time. First, we illustrate a big picture based on the partition tree and then describe the details based on the succinct structures that form our oracle. First, we find the lowest common ancestor of u and v in P. Let lca denote such ancestor; lca represents the first time that u and v appear in the same graph when building the graph according to the algebraic expression. Recall that the algebraic expression associated with the partition tree is proper-k-balanced and hence proper (Lemma 1). Therefore, if there is an edge between u and v, the spots associated with them are joined in lca. In other words, it suffices to check whether there is an edge between the vertices representing labels of u and v in the joined graph stored for lca. We show that this can be done in constant time. In what follows, we describe the above procedure in more details.

Algorithm 2 project$_{L\to M}$

```
Input: pair $(u\text{-}type, u\text{-}p)$ where $u\text{-}type$ is the type a vertex $u$ and $u\text{-}p$ is the projection of $u$ in $L_{u\text{-}type}$,
    i.e., the parenthesis representing the leaf associated with $u$ in $L_{u\text{-}type}$.
Output: index of $u$ in the main tree $M$
$u\text{-}st \leftarrow leaf\text{-}rank_{L_{u\text{-type}}}(u\text{-}p)$ /* $u\text{-}st$ is the index of $u$ among leaves of the same type. */
$u\text{-}lr \leftarrow select_{U_{\text{leaves}}}(ust, u\text{-}type)$ /* Quering lookup table to find the $st$th leaf of type $u\text{-}st$; this gives $u\text{-}lr$, which is
    the index of $u$ among leaves of the main tree $M$ */
$u \leftarrow leaf\text{-}select_M(u\text{-}lr)$
return $u$
```

Lemma 11 *Given any two vertices u and v, we can report whether there is an edge between u and v in constant time.*

Proof Note that u and v are given with their addresses (the indices of their opening parentheses) in the main tree M. First, we use the $parent$ operation on M to see if u and v have the same parent. If they do, we just need to check the join graph associated with their common parent p; this graph is stored in the join sequence J. We find the pre-order index of p among internal nodes of M using $pre\text{-}rank$ and $leaf\text{-}rank$ operations. We use this index to find the block associated with p in the join sequence J. Note that each internal node in the partition tree is associated with a join graph of size $3k$ represented by a block of fixed size $9k^2$. The entry associated with indices of u and v among children of p in this block indicates whether they are neighbors. We use $child\text{-}rank$ operation on M to find such entry in constant time.

Next, assume u and v do not have a common parent. First, we find the lowest common ancestor lca of u and v in the main tree M. Next, we find the depth d of lca in the main tree. On the side, we find the projections and types of u and v in their respective label trees using $project_{M\to L}$ operation of Lemma 11. Using $level\text{-}ancestor$ and $pre\text{-}rank$ queries in the label trees, we find the labels of u and v at depth d of their respective label trees; the results will be the spot at which these vertices are located at node lca of the partition tree. Provided with labels of the two vertices, checking the neighborhood becomes equivalent to checking an entry in the table associated with lca in the join sequence J, which can be done in constant time (again, because of the fixed size $3k$ of the blocks associated with join graphs in J).

By Lemmas 2, 11, all above operations take constant time. Algorithm 3 illustrates the adjacency query as described above. □

Algorithm 3 Adjacency Query

```
Input  : Integers $u$ and $v$ which are indices of two vertices in the main tree $M$
Output: '1' if $u$ and $v$ are neighbors and '0' otherwise
if $parent_M(u) = parent_M(v)$ then
    /* check the join block associated with the common parent of $u$ and $v$ */
    $p \leftarrow parent_M(u)$
        /* $p$ is the index of the common parent in $M$ */
    $p\text{-}rank \leftarrow pre\text{-}rank_M(p) - leaf\text{-}rank_M(p)$
        /* $p\text{-}rank$ is the pre-order rank of $p$ among internal nodes of $M$ */
    $add \leftarrow p\text{-}rank \cdot 9k^2$
        /* $add$ is the address of the block associated with $p$ in the join sequence $J$ */
    $u\text{-}chr \leftarrow child\text{-}rank_M(u)$; $v\text{-}chr \rightarrow child\text{-}rank_M(v)$
        /* $u\text{-}chr$ and $v\text{-}chr$ are ranks of $u$ and $v$ among children of their common parent */
    $result \leftarrow J[add + u\text{-}chr \cdot 3k + v\text{-}chr]$
        /* probing the index which indicates whether $u$ and $v$ are joined in parent $p$ */
else
    $lca \leftarrow lca_M(u, v)$
        /* $lca$ is the index of the least common ancestor of $u$ and $v$ in $M$ */
    $d \rightarrow depth_M(lca)$
        /* $d$ is the depth of the lca in the main tree $M$ */
    $(u\text{-}type, u^*) \leftarrow project_{M \to L}(u)$ ; $(v\text{-}type, v^*) \leftarrow project_{M \to L}(v)$ /* setting the label and projected
        nodes of $u$ and $v$ in their respective label tree */
    $T_u \leftarrow L_{u\text{-}type}$; $T_v \leftarrow L_{v\text{-}type}$
        /* $T_u$ and $T_v$ are the label trees associated with types of $u$ and $v$ */
    $u\text{-}pr \leftarrow pre\text{-}rank_{T_u}(level\text{-}ancestor_{T_u}(u, d))$; $v\text{-}pr \leftarrow pre\text{-}rank_{T_v}(level\text{-}ancestor_{T_v}(v, d))$
        /* $u\text{-}la$ and $v\text{-}la$ are the pre-order indices of ancestors of $u$ and $v$ at level $d$ of their respective label trees */
    $u\text{-}spot = V_{u\text{-}type}[u\text{-}pr]$; $v\text{-}spot = V_{v\text{-}type}[v\text{-}pr]$
        /* $u\text{-}spot$ and $v\text{-}spot$ are the values stored in label trees of $u$ and $v$ at level $d$, i.e., their spot at level $d$ of the partition
        tree. */
    $add \leftarrow (pre\text{-}rank_M(lca) - leaf{-}rank_M(lca)) \cdot 9k^2$
        /* the address of the block associated with $lca$ in the join sequence */
    $result \leftarrow J[add + u\text{-}spot \cdot 3k + v\text{-}spot]$
        /* probing the index which indicates whether $u$ and $v$ are joined in their lca. */
end
return result
```

Example 1 Assume we need to find whether there is an edge between vertices a and g in the graph represented by the proper-k-balanced union tree of Fig. 5. The lowest common ancestor lca of the two vertices in the main tree is the left child of the root in Fig. 6. Note that lca has depth $d = 2$ in the main tree. Reading from the label lookup sequence U_{leaves}, we see that a has type 1 and g has type 2. Using the $level\text{-}ancestor$ query on label trees of types 1 and 2, we see that a and g respectively have values 1 and 3 at depth d of their label trees (see Fig. 7). This implies that they are at spots with labels 1 and 3 of lca. Using the entry associated with lca in the join table, we realize that the two labels 1 and 3 are not connected. Consequently, there is no edge between a and g.

4.4 Neighborhood Queries

Given a vertex u, we describe how to report the set of its neighbors in constant time per neighbor. Recall that u is addressed via its position in the main tree. We visit ancestors of u in that tree one by one. We use the skip-tree (of the same type as u) to skip ancestors in which there is no join operation between the spot of u and another spot. For each visited ancestor, we visit spots that u is joined to. The vertices in that spot are connected to u and we should report them. In doing so, we skip vertices that are reported earlier; it is possible since the reported vertices form a consecutive block among members of the visited spot. In the following lemma, we illustrate the above procedure in details.

Lemma 12 *Given any vertex u, we can report neighbors of u in constant time per neighbor.*

Proof First, we report 'close neighbors' of u, which are the neighbors with the same parent as u in the partition tree. Let p denote the parent of u in the main tree. We locate the block B associated with the join graph of p in the join sequence J. For that, we use $pre\text{-}rank$ and $leaf\text{-}rank$ operations to find the pre-order index of p among internal nodes of M. Since the join blocks of internal nodes have fixed size $9k^2$, we can use this index to locate B in constant time. Next, we locate the address at which the row associated with u starts in B. For that, we need the index of u among children of p, which can be found using $child\text{-}rank$ operation on M. This way, we find the address of the row associated with u in in the join block (matrix) of p in the join sequence J. Note that this row has length $3k$. Using the *select* operation in this row, we report the siblings of u which are connected to it in constant time per sibling. For that, we use $child\text{-}rank$ operation to locate those children of p which have non-zero entries in the row associated with u. Algorithm 4 illustrates the details in reporting close neighbors of u.

Algorithm 4 Neighborhood Query - Reporting Close Neighbors

```
Input  : Integer $u$ which is the index of a vertex in the main tree $M$
Output: Neighbors of $u$ which have the same parent as $u$ in the partition tree.
$p \leftarrow parent_M(u)$ /* the parent of $u$ in $M$ */
$start\text{-}add \leftarrow ((pre\text{-}rank_M(p) - leaf\text{-}rank_M(p)) \cdot 9k^2 + child\text{-}rank_M(u) \cdot 3k$ /* starting address of the
    row associated with $u$ in the join graph of $p$ */
$next\text{-}address \leftarrow select_J(start\text{-}add, \text{`1'})$
    /* the next non-zero entry in the join sequence associated with an edge in the join graph */
while $next\text{-}address < start\text{-}add + 3k$ do
    /* In this loop, we iterate over leaves which are joined with $u$ at their common parent in the partition tree */
    $i \leftarrow next\text{-}index - start\text{-}address$ /* the next neighbor of $i$ among children of their common parent */
    $v \leftarrow child\text{-}select_M(p, i)$ /* the entry associated with $(u, v)$ in the join graph of the common parent of $u$ and $v$ is
        1; so we report $v$ */
    report $v$ as a neighbor of $u$
        $next\text{-}index \leftarrow select_J(next\text{-}index + 1, \text{`1'})$ /* iterating to the next non-zero entry */
end
```

Next, we describe how to report 'far neighbors' of u, i.e., those with different parent than u in the partition tree. First, we use Lemma 11 to find the type of u, the label tree L^* of this type, and the index of its projected vertex u^* of u in L^*.

Our procedure for reporting far neighbors has three nested loops: in the first (outermost) loop, we locate 'critical ancestors' of u, which are those ancestors for which the spot of u is joined with at least one other spot. Note that u has some non-reported neighbors in critical ancestors. We use the *enclose* operation on the skip tree of the same type as u to locate (the depths of) critical ancestors in constant time per ancestor. Note that skip trees encode different parenthesis-types for critical and non-critical ancestors and hence we can locate the critical ancestors using Lemma 3. Given a critical ancestor A of u at depth d of the partition tree, we locate the spot of u in A: we use $level\text{-}ancestor$ query to find the ancestor of u^* at depth d of L^*; the value of that ancestor indicates the spot $u\text{-}spot$ of u at A. Provided with $u\text{-}spot$, in the second nested loop, we visit all 'critical spots' of A. A critical spot is a spot that is joined with $u\text{-}spot$ at A. Critical spots can be located in a similar way that we report close neighbors: we locate the row associated with $u\text{-}spot$ in the join sequence and use the $select$ operation to locate the non-zero entries in that row. Each non-zero entry indicates a critical spot for $u\text{-}spot$. For each critical spot cs, we can read the lookup sequence U_{inner} to read the type $cs\text{-}type$ of vertices in that spot. For that, we use $pre\text{-}rank$ and $leaf\text{-}rank$ operations to find the number of spots in the internal nodes that precede A in the main tree; note that each internal node has a fixed number of $3k$ spots, and we can find the rank $cs\text{-}rank$ of cs among all spots in the partition tree in constant time. The value stored in U_{inner} at index $cs\text{-}rank$ indicates the type $cs\text{-}type$ of spot cs. We use $cs\text{-}type$ to locate a node tn, called 'target node', in the label tree $L_{cs\text{-}type}$ which is associated with the spot cs. For that, we use the $rank$ operation on U_{inner} to find the index of cs among spots of the same type, and $pre\text{-}select$ operation to find the node tn of that rank in the tree $L_{cs\text{-}typ}$. Leaves of the subtree rooted at tn are all connected to u (they are joined at the node associated with A in the main/partition tree). In the third (innermost) nested loop, we go through these leaves to report vertices which are not already reported. Using, $lmost\text{-}child$ and $rmost\text{-}child$ operations on $T_{cs\text{-}type}$, we locate the range of these leaves and visit them one by one using the $leaf\text{-}select$ operation. Note that neighbors that are already reported are among the leaves of the tree rooted at the ancestor of u at depth $d+1$ in the main tree. So, we can set a 'skip-range' in the main tree which includes indices of these leaves (and no other leaf) in the main tree. When we visit a leaf of $L_{cs\text{-}type}$ which is projected to the skip-range in the main tree, we skip to the next node whose projection is not in the skip-range. This can be done using $rank$ operation on U_{inner} (to find the number of vertices of type $cs\text{-}type$ that precede the right-boundary of the skip-range) and $leaf\text{-}select$ operation (to find the next vertex of type $cs\text{-}type$ which is out of the skip-range).

In a nutshell, to report far vertices of u, we visit critical ancestors of u in the main tree one by one (first nested loop). In each critical ancestor, we visit critical spots one by one (second nested loop). For a given critical spot, we report neighbors which are outside of the skip-range one by one (third nested loop). Be definition, all critical nodes include at least one critical spot, and all critical spots include at least one non-reported neighbor of u. All operations take constant time. Hence, neighbors of u are reported in constant time per neighbor. Algorithm 5 illustrates the above procedure in details. □

Algorithm 5 Neighborhood Query - Reporting Far Neighbors

```
Input  : Integer $u$ which is the index of a vertex in the main tree $M$
Output: Indices of neighbors of $u$ in the main tree $M$
$(u\text{-}type, u^*) \leftarrow project_{M\rightarrow L}(u)$ /* the label and projected node of $u$ in its label tree */
$L^* \leftarrow L_{u\text{-}type}$; $P^* \leftarrow P_{u\text{-}type}$ /* $L^*$ and $P^*$ are respectively the label tree and the skip-tree associated with the type
    of $u$ */
```

```
$next\text{-}node \leftarrow u^*$
do
    /* We iterate on the path from $u$ to the root of the partition tree; in each iteration, we report neighbors joined with $u$ in an ancestor on this path. */
    $d \leftarrow depth_{P*}(m\text{-}enclose_{P*}(u^*, `1'))$ /* The depth of the next ancestor of $u$ represented by a parenthesis of type ‘1’ in $P^*$; $u$ is connected to at least one vertex at the node associated with this ancestor in the partition tree. */
    $A \leftarrow level\text{-}ancestor_M(u, d)$ /* $A$ is a critical ancestor of $u$ in the main tree, i.e., it is joined to at least one vertex at this node of the partition tree */
    $u\text{-}spot \leftarrow V_{u\text{-}type}[pre\text{-}rank_{L*}(level\text{-}ancestor_{L*}(u^*, d))]$ /* $u\text{-}spot$ is the label (spot) of $u$ at depth $d$ of the partition tree */
    $start\text{-}add \leftarrow (pre\text{-}rank_M(A) - leaf\text{-}rank_M(A)) \cdot 9k^2 + u\text{-}spot \cdot 3k$;
      $end\text{-}add \leftarrow start\text{-}add + 3k$ /* [$start\text{-}add$,$end\text{-}address$] is the range of entries in the row associated with $u\text{-}spot$ in the join sequence $J$. */
    $B \leftarrow level\text{-}ancestor_M(u, d+1))$/* neighbors in the subtree rooted at this node are already reported */
    $lch \leftarrow leaf\text{-}rank_M(lmost\text{-}leaf_M(B)$
      $rch \leftarrow leaf\text{-}rank_M(rmost\text{-}leaf_M(B)$
      $skip\text{-}range \leftarrow [pre\text{-}rank_M(lmost\text{-}leaf_M(B), pre\text{-}rank_M(rmost\text{-}leaf_M(B)]$ /* vertices with leaf-rank in the $skip - range$ should be skipped since neighbors of $u$ in this range are reported */
    $next\text{-}address \leftarrow select_J(start\text{-}add, `1')$ /* the next non-zero entry in the join sequence associated with an edge in the join graph */
    while $next\text{-}address < start\text{-}add + 3k$ do
        /* In this loop, we iterate over the spots which are joined with the spot of $u$ among spots of $A$ of the partition tree */
        $cs \leftarrow next\text{-}index - start\text{-}address$ /* $cs$ is the index of a critical spot at node $A$; $u$ is connected to vertices at this spot of $A$. */
        $cs\text{-}rank \leftarrow (pre\text{-}rank_M(A) - leaf\text{-}rank_M(A)) \cdot 3k + cs$
          /* $cs\text{-}rank$ is the rank of the spot of $cs$ among all spots (of internal nodes) in the partition tree (we find the number of internal nodes preceding $A$ in pre-order; each of them has $3k$ spots.) */
        $cs\text{-}type \leftarrow U_{inner}[cs\text{-}rank]$ /* querying the label-lookup sequence for the type of $cs$ (vertices in $cs$ have this label in the root of partition tree) */
        $T^s \leftarrow L_{cs\text{-}type}$ /* the label tree associated with $cs\text{-}type$ */
        $cs\text{-}rank\text{-}same \leftarrow rank_{U_{inner}}(cs\text{-}rank, cs\text{-}type)$ /* $cs\text{-}rank\text{-}same$ is the rank of the spot of $cs$ among spots of the same type in the partition tree */
        $tn \leftarrow pre\text{-}select_{T^s}(cs\text{-}rank\text{-}same)$ /* $tn$ is the ‘target node’ in tree $T^s$; all leaves of the tree rooted at $tn$ in are neighbors to $u$ */
        $begin\text{-}leaf\text{-}rank \leftarrow leaf\text{-}rank_{T^s}(lmost\text{-}child_{T^s}(tn))$
          $end\text{-}leaf\text{-}rank \leftarrow leaf\text{-}rank_{T^s}(rmost\text{-}child_{T^s}(tn))$ /* the ranks of the left-most and right-most children of $tn$ in $T^s$ among leaves of $T^s$; leaves in this range are neighbors of $u$ */
        $q \leftarrow begin\text{-}leaf\text{-}rank$ /* counter */
        while $q < end\text{-}leaf\text{-}rank$ do
            /* we loop through all children of $tn$, skipping once to avoid reporting neighbors that are reported in previous iterations of the outer loop */
            $v \leftarrow project_{L \to M}(cs\text{-}type, leaf\text{-}select_{T^s}(q))$ /* $v$ is a neighbor of $u$; if it is reported earlier, we should skip it and other reported vertices */
            if $pre\text{-}rank_M(v) \in skip\text{-}range$ then
                /* find the next node to be reported: */
                $q \leftarrow leaf\text{-}select_{T^s}(rank_{U\text{-}inner}(rch, cs\text{-}type) + 1)$ /* the next vertex of type $cs\text{-}type$ which is not in the skip range */
            else
                report $v$ as a neighbor of $u$
                $q \leftarrow q + 1$
            end
        end
    end
    $next\text{-}index \leftarrow select_J(next_index + 1, `1')$
while $next\text{-}node \neq Null$;
return result
```

Example 2 Assume we want to report neighbors of vertex a in the graph represented by the proper-k-balanced union tree of Fig. 5. First, we probe the join sequence J to report the close neighbors of a, i.e., b and c. Next, we locate a in the label/skip tree of type 1 (Fig. 7). Using the $m\text{-}enclose$ operation on the skip-tree, we locate the first ancestor in which a is joined to some other vertices. This ancestor would be the left child of root in Fig. 7a, and has depth $d = 2$ in the label/skip tree. Locating the ancestor A of a at depth 2 in the main tree and checking the non-zero entries of the join table associated with the row of u in A's block in the join sequence J, we realize that a is connected (joined) to vertices at spot $cp = 2$ of A. Checking the lookup sequence U_{leaves}, we figure these vertices have type $cs\text{-}type = 2$. Let A' be the vertex associated with spot 2 of A in the label tree T^s of type 2 (the left child of the root in Fig. 7a). Note that all leaves in the tree rooted at A', namely b, c and f, are joined to a. Let $B \in M$ be the ancestor of a which is a child of A (the left child of A in Fig. 6). When reporting children of A' in T^s, we skip reporting vertices which are created in the subtree of M rooted at B, namely, we skip reporting b and c as they are reported earlier.

From the discussions above, we conclude the following theorem:

Theorem 3 *Given a graph of size n and constant clique-width in the form of an algebraic expression on at most k labels where k is a constant with respect to n, an oracle is constructed to answer adjacency queries in constant time. The neighborhood queries are also supported in constant time, i.e., the neighbors of a given vertex are reported in constant time per neighbor. The space requirement of the oracle is* $O(kn)$ *bits.*

4.5 Degree Queries

For the degree queries, we apply a more explicit representation. We decompose the partition tree into $\Theta(n/\log(n))$ subtrees of size $2\lceil\log(n)\rceil$, using the following lemma with $L = \lceil\log(n)\rceil$ (see Fig. 8 for an illustration of the lemma):

Lemma 13 *[12] A tree with n nodes can be decomposed into* $O(n/L)$ *subtrees of size at most L, where L is an arbitrary integer. These are pairwise disjoint aside from the subtree roots, i.e., if a node x is present in two subtrees, then x is the root of both subtrees. Furthermore, aside from edges stemming from the component root nodes, there is at most one edge per component leaving a node of a component to its child in another component.*

For each subtree in the above partitioning, there will be at most two nodes which are connected to nodes in other subtrees, one being the root of the subtree and potentially another node that we refer to as the *lower hub* of the subtree. Neighbors of a vertex v can be partitioned into three groups: group 1 are those created in the same subtree as v in the partition tree, group 2 are those that will be disconnected from the partition tree if we remove the lower hub, and group 3 are the rest of vertices. In what follows, we show how to report the number of neighbors of any vertex in groups 2 and 3

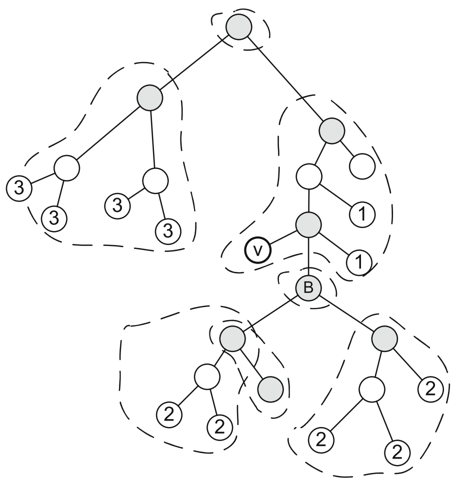

Fig. 8 Decomposition of a tree into subtree for value $L = 4$. Subtrees have size at most $2L = 8$. Note that the left child of B is present in two subtrees and hence is the root of both subtrees. The *highlighted nodes* are those which are connected to other subtrees. The *numbers* on leaves indicate the group of vertices in the leaf with respect to node v

provided with as structure of size $O(kn)$ and in time $O(k)$. Later we bootstrap the same decomposition to report neighbors in group 1.

Consider the node B in another subtree of the partition tree which is adjacent to the lower hub of a subtree T. Vertices introduced in the subtree rooted at B are partitioned into k spots, each representing a label in the algebraic expression. This implies that, these vertices have the same neighborhood with respect to vertices introduced in T (because vertices of T are joined to the tree 'later'). In other words, each vertex v in T is connected to all or none of vertices in each spot of B. For each vertex v, we store a bitstring of size k which indicates whether v is connected to vertices in spots of B. We store this sequence using a rank/select structure. This requires $O(kn)$ bits for all vertices. Moreover, for each lower hub, we explicitly store the number of vertices in each spot in the neighboring node B in $O(\log(n))$ bits; since there are $O(n/\log(n))$ lower hubs, that requires $O(kn)$ bits. To report the number of neighbors of each vertex among vertices of group 2, we check the bitstring associated with the vertex and for each bit with value 1 (for each spot connected to v) add up the number of vertices in the spot to the answer. Note that we can find the bits with value 1 using *select* operation in constant time. There are k spots and hence it takes $O(k)$ time.

We report the number of replicas in group 3 in a similar manner of group 2. We note that vertices in the same spot as a vertex v in the root of the subtree have the same neighborhood among vertices which are introduced later as non-descendants of the root of the subtree. For each spot in the root, we store the number of these neighboring vertices in $O(\log(n))$ bits; this sums up to $O(kn)$ for all $O(n/\log(n))$ trees. For each vertex, we store its index (spot) in the root; that requires $O(n\log(k))$ bits for all vertices. To report the number of neighbors of v in group 3, we check its index in the root and report the stored number for that spot in constant time.

It remains to report the number of neighbors in group 1. If we decompose each tree of size $L_1 = \lceil\log(n)\rceil$ into $O(L_1/L_2)$ subtrees of size $L_2 = \lceil\log(L_1)\rceil$, the neighbors of each vertex in group 1 can be partitioned in groups 1_1, 1_2 and 1_3, defined in a similar fashion as above. Neighbors in group 1_2 and 1_3 can be reported with an additional structure defined in a similar way with size $O(kn)$ and in time $O(k)$. To report neighbors in group 1_1, we recursively decompose trees of size L_i into $O(L_i/L_{i+1})$ trees of size $L_{i+1} = \lceil \log(L_i) \rceil$. When the depth of recursion becomes $\Theta(\log^* n)$, the number of vertices in L_i will be a constant, and the number of neighbors of vertices within each subtree can be explicitly stored in $\Theta(n)$. The total size of structures used for reporting neighbors in each level of recursion is $O(kn)$, which sums up to $O(kn \log^* n)$ for all levels. Similarly, reporting neighbors in each level requires $O(k)$ which sums up in total time complexity of $\Theta(k \log^* n)$.

Theorem 4 *Given a graph of size n and clique-width at most k, an oracle is constructed to answer degree queries in $O(k \log^* n)$ time. The space requirement of the oracle is $O(kn \log^* n)$ bits.*

5 Concluding Remarks

We presented a compact data structure for graphs of clique-width at most k that answers adjacency and neighborhood queries in constant time using $O(kn)$ space. Our structure supports degree query in time $O(k \log^* n)$ using $O(kn \log^* n)$ bits. We also proved an information theory lower bound of $kn - o(n)$ for the number of bits required to represent graphs of size n and clique-width at most k up to isomorphism. Presenting a succinct structure that represents graphs in $kn + o(kn)$ bits or improving this lower bound remains an open question. Removing the $\log^* n$ factor for the structure used for the degree queries is another question to investigate. We note that ideas and constructions used in this paper are expected to be useful in presenting compact structures based on other graph width parameters, and leave this as a topic for future research.

Acknowledgements The author would like to thank the anonymous reviewers for their careful reading of the paper and for their valuable comments and suggestions which resulted in substantial improvement of the paper.

References

1. Barbay, J., Aleardi, L.C., He, M., Munro, J.I.: Succinct representation of labeled graphs. Algorithmica **62**(1–2), 224–257 (2012)
2. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: Proceedings of 21st International Symposium on Algorithms and Computation (ISAAC), pp. 315–326. (2010)
3. Blandford, D.K., Blelloch, G.E., Kash, I.A.: Compact representations of separable graphs. In: Proceedings of 14th Symposium on Discrete Algorithms (SODA), pp. 679–688. (2003)
4. Blelloch, G.E., Farzan, A.: Succinct representations of separable graphs. In: Proceedings of 21st Combinatorial Pattern Matching Conference (CPM), pp. 138–150. (2010)
5. Corneil, D.G., Rotics, U.: On the relationship between clique-width and treewidth. SIAM J. Comput. **34**(4), 825–847 (2005)
6. Courcelle, B., Makowsky, J.A., Rotics, U.: Linear time solvable optimization problems on graphs of bounded clique-width. Theory Comput. Syst. **33**(2), 125–150 (2000)
7. Courcelle, B., Olariu, S.: Upper bounds to the clique width of graphs. Discrete Appl. Math. **101**(1–3), 77–114 (2000)
8. Courcelle, B., Vanicat, R.: Query efficient implementation of graphs of bounded clique-width. Discrete Appl. Math. **131**(1), 129–150 (2003)

9. Farzan, A.: Succinct representation of trees and graphs. PhD thesis, School of Computer Science, University of Waterloo, (2009)
10. Farzan, A., Fischer, J.: Compact representation of posets. In: Proceedings of 22nd International Symposium on Algorithms and Computation (ISAAC), pp. 302–311. (2011)
11. Farzan, A., Kamali, S.: Compact navigation and distance oracles for graphs with small treewidth. Algorithmica **69**(1), 92–116 (2014)
12. Farzan, A., Munro, J.I.: A uniform paradigm to succinctly encode various families of trees. Algorithmica **68**(1), 16–40 (2014)
13. Golumbic, M.C., Rotics, U.: On the clique-width of some perfect graph classes. Int. J. Found. Comput. Sci. **11**(3), 423–443 (2000)
14. Golynski, A., Munro, J.I., Srinivasa Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proceedings of 17th Symposium on Discrete Algorithms (SODA), pp. 368–373. (2006)
15. Kaminski, M., Lozin, V.V., Milanic, M.: Recent developments on graphs of bounded clique-width. Discrete Appl. Math. **157**(12), 2747–2761 (2009)
16. Meer, K., Rautenbach, D.: On the OBDD size for graphs of bounded tree- and clique-width. Discrete Math. **309**(4), 843–851 (2009)
17. Munro, J.I.: Succinct data structures. Electr. Notes Theor. Comput. Sci. **91**, 3 (2004)
18. Munro, J.I., Nicholson, P.K.: Succinct posets. Algorithmica **76**(2), 445–473 (2016)
19. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. ACM Trans. Algorithms **10**(3), 16:1–16:39 (2014)
20. Spinrad, J.P.: Efficient Graph Representations. The Fields Institute for Research in Mathematical Sciences, Toronto (2003)
21. Vanherpe, J.-M.: Clique-width of partner-limited graphs. Discrete Math. **276**(1–3), 363–374 (2004)
22. Witten, I.H., Moffat, A., Bell, Timothy C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann Publishers Inc., Burlington (1999)