ADJACENCY LABELLING FOR PLANAR GRAPHS (AND BEYOND)

Vida Dujmović,[♀] *Louis Esperet*,[♁] *Cyril Gavoille*,[♂] *Gwenaël Joret*,[♃] *Piotr Micek*,[♄] *and Pat Morin*[♅]

Abstract. We show that there exists an adjacency labelling scheme for planar graphs where each vertex of an n-vertex planar graph G is assigned a $(1+o(1))\log_2 n$-bit label and the labels of two vertices u and v are sufficient to determine if uv is an edge of G. This is optimal up to the lower order term and is the first such asymptotically optimal result. An alternative, but equivalent, interpretation of this result is that, for every positive integer n, there exists a graph U_n with $n^{1+o(1)}$ vertices such that every n-vertex planar graph is an induced subgraph of U_n. These results generalize to a number of other graph classes, including bounded genus graphs, apex-minor-free graphs, bounded-degree graphs from minor closed families, and k-planar graphs.

[♀]School of Computer Science and Electrical Engineering, University of Ottawa, Canada. This research was partially supported by NSERC.

[♁]Laboratoire G-SCOP (CNRS, Univ. Grenoble Alpes), Grenoble, France. Partially supported by the French ANR Projects ANR-16-CE40-0009-01 (GATO) and ANR-18-CE40-0032 (GrR).

[♂]LaBRI, University of Bordeaux, France. This research was partially supported by the French ANR projects ANR-16-CE40-0023 (DESCARTES) and ANR-17-CE40-0015 (DISTANCIA).

[♃]Département d'Informatique, Université Libre de Bruxelles, Brussels, Belgium. Research supported by an ARC grant from the Wallonia-Brussels Federation of Belgium and by a grant from the National Fund for Scientific Research (FNRS).

[♄]Theoretical Computer Science Department, Faculty of Mathematics and Computer Science, Jagiellonian University, Kraków, Poland. This research was partially supported by the Polish National Science Center grant (BEETHOVEN; UMO-2018/31/G/ST1/03718).

[♅]School of Computer Science, Carleton University, Canada. This research was partially supported by NSERC.

Contents

1 Introduction

A family $\mathcal{G}$ of graphs has an *$f(n)$-bit adjacency labelling scheme* if there exists a function $A : (\{0,1\}^*)^2 \to \{0,1\}$ such that for every n-vertex graph $G \in \mathcal{G}$ there exists $\ell : V(G) \to \{0,1\}^*$ such that $|\ell(v)| \leqslant f(n)$ for each vertex v of G and such that, for every two vertices v, w of G,

$$A(\ell(v), \ell(w)) = \begin{cases} 0 & \text{if } vw \notin E(G); \\ 1 & \text{if } vw \in E(G). \end{cases}$$

Let $\log x := \log_2 x$ denote the binary logarithm of x. In this paper we prove the following result:

Theorem 1. *The family of planar graphs has a $(1 + o(1)) \log n$-bit adjacency labelling scheme.*

Theorem 1 is optimal up to the lower order term, which is $\mathcal{O}\left(\sqrt{\log n \log\log n}\right)$ in our proof. An alternative, but equivalent, interpretation of Theorem 1 is that, for every integer $n \geqslant 1$, there exists a graph U_n with $n^{1+o(1)}$ vertices such that every n-vertex planar graph is isomorphic to some vertex-induced subgraph of U_n.[1]

Note that the proof of Theorem 1 is constructive: it gives an algorithm producing the labels in $\mathcal{O}(n \log n)$ time.

1.1 Previous Work

The current paper is the latest in a series of results dating back to Kannan, Naor, and Rudich [19, 20] and Muller [24] who defined adjacency labelling schemes[2] and described $\mathcal{O}(\log n)$-bit adjacency labelling schemes for several classes of graphs, including planar graphs. Since this initial work, adjacency labelling schemes and, more generally, informative labelling schemes have remained a very active area of research [2, 8, 1, 5, 7, 6, 9, 3].

Here we review results most relevant to the current work, namely results on planar graphs and their supporting results on trees and bounded-treewidth graphs. First, a superficial review: Planar graphs have been shown to have $(c + o(1)) \log n$-bit adjacency labelling schemes for successive values of $c = 6, 4, 3, 2, \frac{4}{3}$ and finally Theorem 1 gives the optimal[3] result $c = 1$. We now give details of these results.

Muller's scheme for planar graphs [24] is based on the fact that planar graphs are 5-degenerate. This scheme orients the edges of the graph so that each vertex has 5 outgoing edges, assigns each vertex v an arbitrary $\lceil \log n \rceil$-bit identifier, and assigns a label to v consisting of v's identifier and the identifiers of the targets of v's outgoing edges. In this way, each vertex v is assigned a label of length at most $6\lceil \log n \rceil$. Kannan, Naor, and Rudich

[1] There is a small technicality that the equivalence between adjacency labelling schemes and universal graphs requires that $\ell : V(G) \to \{0,1\}^*$ be injective. The labelling schemes we discuss satisfy this requirement. For more details about the connection between labelling schemes and universal graphs, the reader is directed to Spinrad's monograph [27, Section 2.1].

[2] There are some small technical differences between the two definitions that have to do with the complexity of computing $\ell(\cdot)$ as a function of G and $A(\cdot, \cdot)$.

[3] It is easy to see that, in any adjacency labelling scheme for any n-vertex graph G in which no two vertices have the same neighbourhood, all labels must be distinct, so some label must have length at least $\lceil \log n \rceil$.

[20] use a similar approach that makes use of the fact that planar graphs have arboricity 3 (so their edges can be partitioned into three forests [25]) to devise an adjacency labelling scheme for planar graphs whose labels have length at most $4\lceil \log n \rceil$.

A number of $(1+o(1))\log n$-bit adjacency labelling schemes for forests have been devised [12, 9, 4], culminating with a recent $(\log n + \mathcal{O}(1))$-bit adjacency labelling scheme [4] for forests. Combined with the fact that planar graphs have arboricity 3, these schemes imply $(3+o(1))\log n$-bit adjacency labelling schemes for planar graphs.

A further improvement, also based on the idea of partitioning the edges of a planar graph into simpler graphs was obtained by Gavoille and Labourel [18]. Generalizing the results for forests, they describe a $(1+o(1))\log n$-bit adjacency labelling scheme for n-vertex graphs of bounded treewidth. As is well known, the edges of a planar graph can be partitioned into two sets, each of which induces a bounded treewidth graph. This results in a $(2+o(1))\log n$-bit adjacency labelling scheme for planar graphs.

Very recently, Bonamy, Gavoille, and Pilipczuk [10] described a $(4/3+o(1))\log n$-bit adjacency labelling scheme for planar graphs based on a recent *graph product structure theorem* of Dujmović et al. [15]. This product structure theorem states that any planar graph is a subgraph of a strong product $H \boxtimes P$ where H is a bounded-treewidth graph and P is a path. See Figure 1. It is helpful to think of $H \boxtimes P$ as a graph whose vertices can be partitioned into $h := |V(P)|$ *rows* $H_1, \ldots, H_h$, each of which induces a copy of H and with vertical and diagonal edges joining corresponding and adjacent vertices between consecutive rows.

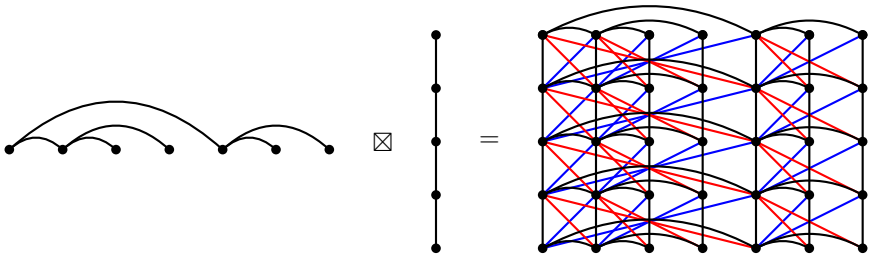

Figure 1: The strong product $H \boxtimes P$ of a tree H and a path P.

The product structure theorem quickly leads to a $(1+o(1))\log(mh)$-bit labelling scheme where $m := |V(H)|$ and $h := |V(P)|$ by using a $(1+o(1))\log m$-bit labelling scheme for H (a bounded treewidth graph), a $\lceil \log h \rceil$-bit labelling scheme for P (a path), and a constant number of bits to locally encode the subgraph of $H \boxtimes P$ (of constant arboricity). However, for an n-vertex graph G that is a subgraph of $H \boxtimes P$ in the worst case m and h are each $\Omega(n)$, so this offers no immediate improvement over the existing $(2+o(1))\log n$-bit scheme.

Bonamy, Gavoille, and Pilipczuk improve upon this by cutting P (and hence G) into subpaths of length $n^{1/3}$ in such a way that this corresponds to removing $\mathcal{O}(n^{2/3})$ vertices of G that have a neighbourhood of size $\mathcal{O}(n^{2/3})$. The resulting (cut) graph is a subgraph of $H' \boxtimes P'$ where H' has bounded treewidth, $|H'| \leqslant n$, and P' is a path of length $n^{1/3}$ so it has a labelling scheme in which each vertex has a label of length $(1+o(1))\log(|H'| \cdot |P'|) \leqslant$

$(4/3 + o(1))\log n$. A slight modification of this scheme allows for the $\mathcal{O}(n^{2/3})$ *boundary* vertices adjacent to the cuts to have shorter labels, of length only $(2/3 + o(1))\log n$. The cut vertices and the boundary vertices induce a bounded-treewidth graph of size $\mathcal{O}(n^{2/3})$. The vertices in this graph receive secondary labels of length $(2/3 + o(1))\log n$. In this way, every vertex receives a label of length at most $(4/3 + o(1))\log n$.

1.2 New Results

The adjacency labelling scheme described in the current paper is also based on the product structure theorem for planar graphs, but it avoids cutting the path P, and thus avoids boundary vertices that take part in two different labelling schemes. Instead, it uses a weighted labelling scheme on the rows $H_1, \dots, H_h$ of $H \boxtimes P$ in which vertices that belong to H_i receive a label of length $(1 + o(1))\log n - \log W_i$ where W_i is related to the number of vertices of G contained in H_i and H_{i-1}. The vertices of G in row i participate in a secondary labelling scheme for the subgraph of G contained in H_i and H_{i-1} and the labels in this scheme have length $\log W_i + o(\log n)$. Thus every vertex receives two labels, one of length $(1 + o(1))\log n - \log W_i$ and another of length $\log W_i + o(\log n)$ for a total label length of $(1 + o(1))\log n$.

The key new technique that allows all of this to work is that the labelling schemes of the rows $H_1, \dots, H_h$ are not independent. All of these labelling schemes are based on a single balanced binary search tree T that undergoes insertions and deletions resulting in a sequence of related binary search trees $T_1, \dots, T_h$ where each T_i represents all vertices of G in H_i and H_{i-1} and the label assigned to a vertex of H_i is essentially based on a path from the root of T_i to some vertex of T_i. By carefully maintaining the binary search tree T, the trees T_{i-1} and T_i are similar enough so that the label for v in H_i can be obtained, with $o(\log n)$ additional bits from the label for v in H_{i-1}.

The product structure theorem has been generalized to a number of additional graph families including bounded-genus graphs, apex-minor free graphs, bounded-degree graphs from minor-closed families, k-planar graphs, powers of bounded-degree bounded genus graphs, and k-nearest neighbour graphs of points in $\mathbb{R}^2$ [15, 16]. As a side-effect of designing a labelling scheme to work directly on subgraphs of a strong product $H \boxtimes P$, where H has bounded treewidth and P is a path, we obtain $(1 + o(1))\log n$-bit labelling schemes for all of these graph families. All of these results are optimal up to the lower order term.

A graph is *apex* if it has a vertex whose removal leaves a planar graph. A graph is *k-planar* if it has a drawing in the plane in which each edge is involved in at most k crossings. Such graphs provide a natural generalisation of planar graphs, and have been extensively studied [21]. The definition of k-planar graphs naturally generalises for other surfaces. A graph G is *(g,k)-planar* if it has a drawing in some surface of Euler genus at most g in which each edge of G is involved in at most k crossings. Note that already 1-planar graphs are not minor closed. The generalization of Theorem 1 provided by known product structure theorems is summarized in the following result:

Theorem 2. *For every fixed integer $t \geqslant 1$, the family of all graphs G such that G is a subgraph of $H \boxtimes P$ for some graph H of treewidth t and some path P has a $(1 + o(1))\log n$-bit adjacency labelling scheme. This includes the following graph classes:*

1. *graphs of bounded genus and, more generally, apex-minor free graphs;*
2. *bounded degree graphs that exclude a fixed graph as a minor; and*
3. *k-planar graphs and, more generally,* (g,k)*-planar graphs.*

The case of graphs of bounded degree from minor-closed classes (point 2 in Theorem 2) is particularly interesting since, prior to the current work, the best known bound for adjacency labelling schemes in planar graphs of bounded degree was the same as for general planar graphs, i.e., $(4/3+o(1))\log n$. On the other hand, our Theorem 2 now gives an asymptotically optimal bound of $(1+o(1))\log n$ for graphs of bounded degree from any proper minor-closed class.

1.3 Outline

The remainder of the paper is organized as follows. Section 2 reviews some preliminary definitions and easy results. Section 3 describes a new type of balanced binary search tree that has the specific properties needed for our application. Section 4 solves a special case, where G is an n-vertex subgraph of $P_1 \boxtimes P_2$ where P_1 and P_2 are both paths. We include it to highlight the generic idea behind our adjacency labelling scheme. Section 5 solves the general case in which G is an n-vertex subgraph of $H \boxtimes P$ where H has bounded treewidth and P is a path. Section 6 concludes with a discussion of the computational complexity of assigning labels and testing adjacency and presents directions for future work.

2 Preliminaries

All graphs we consider are finite and simple. The vertex and edge sets of a graph G are denoted by $V(G)$ and $E(G)$, respectively. The *size* of a graph G is denoted by $|G| := |V(G)|$.

For any graph G and any vertex $v \in V(G)$, let $N_G(v) := \{w \in V(G) : vw \in E(G)\}$ and $N_G[v] := N_G(v) \cup \{v\}$ denote the open neighbourhood and closed neighbourhood of v in G, respectively.

2.1 Prefix-Free Codes

For a string $s = s_1,\dots,s_k$, we use $|s| := k$ to denote the length of s. A string $s_1,\dots,s_k$ is a *prefix* of a string $t_1,\dots,t_\ell$ if $k \leqslant \ell$ and $s_1,\dots,s_k = t_1,\dots,t_k$. A *prefix-free code* $c : X \to \{0,1\}^*$ is a one-to-one function in which $c(x)$ is not a prefix of $c(y)$ for any two distinct $x, y \in X$. Let $\mathbb{N}$ denote the set of non-negative integers. The following is a classic observation[4] of Elias from 1975.

Lemma 3 (Elias [17])**.** *There exists a prefix-free code* $\gamma : \mathbb{N} \to \{0,1\}^*$ *such that, for each* $i \in \mathbb{N}$, $|\gamma(i)| \leqslant 2\lfloor \log(i+1) \rfloor + 1$.

In the remainder of the paper, γ (which we call an *Elias encoding*) will be used extensively, without referring systematically to Lemma 3.

[4] Observing that if the binary writing of an integer $i \geqslant 1$ is w, where w is a word of $\lfloor \log i \rfloor$ bits, then the code $0^{|w|}, 1, w$, i.e., the length of w in unary followed by 1 and then i in binary, is prefix-free.

2.2 Labelling Schemes Based on Binary Trees

A *binary tree* T is a rooted tree in which each node except the root is either the *left* or *right* child of its parent and each node has at most one left and at most one right child. For any node x in T, $P_T(x)$ denotes the path from the root of T to x. The *length* of a path P is the number of edges in P, i.e., $|P|-1$. The *depth*, $d_T(x)$ of x is the length of $P_T(x)$. The *height* of T is $h(T) := \max_{x\in V(T)} d_T(x)$. A *perfectly balanced* binary tree is any binary tree T of height $h(T) = \lfloor \log |T| \rfloor$.

A binary tree is *full* if each non-leaf node has exactly two children. For a binary tree T, we let T^+ denote the full binary tree obtained by attaching to each node x of T $2-c_x$ leaves where $c_x \in \{0,1,2\}$ is the number of children of x. We call the leaves of T^+ the *external nodes* of T. (Note that none of these external nodes are in T.)

A node a in T is a *T-ancestor* of a node x in T if $a \in V(P_T(x))$. If a is a T-ancestor of x then x is a *T-descendant* of a. (Note that a node is a T-ancestor and T-descendant of itself.) For a subset of nodes $X \subseteq V(T)$, the *lowest common T-ancestor* of X is the maximum-depth node $a \in V(T)$ such that a is a T-ancestor of x for each $x \in X$.

Let $P_T(x_r) = x_0, \ldots, x_r$ be a path from the root x_0 of T to some node x_r (possibly $r = 0$). Then the *signature* of x_r in T, denoted $\sigma_T(x_r)$ is a binary string $b_1, \ldots, b_r$ where $b_i = 0$ if and only if x_i is the left child of x_{i-1}. Note that the signature of the root of T is the empty string.

A *binary search tree* T is a binary tree whose node set $V(T)$ consists of distinct real numbers and that has the *binary search tree property*: For each node x in T, $z < x$ for each node z in x's left subtree and $z > x$ for each node z in x's right subtree. For any $x \in \mathbb{R} \setminus V(T)$, the *search path* $P_T(x)$ in T is the unique root-to-leaf path $v_0, \ldots, v_r$ in T^+ such that adding x as a (left or right, as appropriate) child of v_{r-1} in T would result in a binary search tree T' with $V(T') = V(T) \cup \{x\}$.

The following observation allows us to compare values in a binary search tree just given their signatures in the tree.

Observation 4. *For any binary search tree T and any nodes x, y in T, we have $x < y$ if and only if $\sigma_T(x)$ is lexicographically less than $\sigma_T(y)$.*

Let $\mathbb{R}^+$ denote the set of positive real numbers. The following is a folklore result about biased binary search trees, but we sketch a proof here for completeness.

Lemma 5. *For any finite $S \subset \mathbb{R}$ and any function $w : S \to \mathbb{R}^+$, there exists a binary search tree T with $V(T) = S$ such that, for each $y \in S$, $d_T(y) \leqslant \log(W/w(y))$, where $W := \sum_{y\in S} w(y)$.*

Proof. The proof is by induction on $|S|$. The base case $|S| = 0$ is vacuously true. For any $x \in \mathbb{R}$, let $S_{<x} := \{y \in S : y < x\}$ and $S_{>x} := \{y \in S : y > x\}$. For $|S| \geqslant 1$, choose the root of T to be the unique node $y_0 \in S$ such that $\sum_{z\in S_{<y_0}} w(z) \leqslant W/2$ and $\sum_{z\in S_{>y_0}} < W/2$. Apply induction on $S_{<y_0}$ and $S_{>y_0}$ to obtain the left and right subtrees of T, respectively.

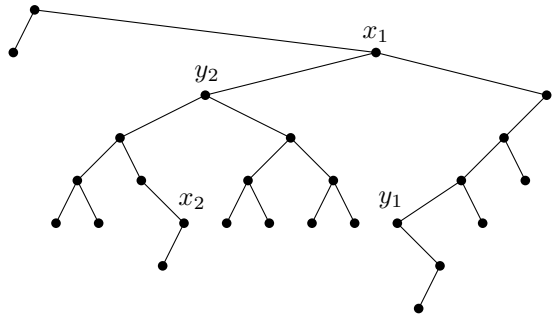

Figure 2: An illustration of Observation 6: (1) $\sigma_T(y_1) = 11000$ and $\sigma_T(x_1) = 1\cancel{1000}$ (2) $\sigma_T(y_2) = 10\cancel{011}$ and $\sigma_T(x_2) = 10011$.

Then $d_T(y_0) = 0 = \log 1 \leqslant \log(W/w(y_0))$. For each $y \in S_{<y_0}$,

$$d_T(y) \leqslant 1 + \log\left(\frac{\sum_{z \in S_{<y_0}} w(z)}{w(y)}\right) \leqslant 1 + \log\left(\frac{W/2}{w(y)}\right) = \log\left(\frac{W}{w(y)}\right),$$

and the same argument shows that $d_T(y) < \log(W/w(y))$ for each $y \in S_{>y_0}$. □

The following fact about binary search trees is useful, for example, in the deletion algorithms for several types of balanced binary search trees [22, Section 6.2.3], see Figure 2:

Observation 6. *Let T be a binary search tree and let x, y be nodes in T such that $x < y$ and there is no node z in T such that $x < z < y$, i.e., x and y are consecutive in the sorted order of $V(T)$. Then*

1. *(if y has no left child) $\sigma_T(x)$ is obtained from $\sigma_T(y)$ by removing all trailing 0's and the last 1; or*
2. *(if y has a left child) $\sigma_T(x)$ is obtained from $\sigma_T(y)$ by appending a 0 followed by $d_T(y) - d_T(x) - 1$ 1's.*

Therefore, there exists a function $D : (\{0,1\}^*)^2 \to \{0,1\}^*$ such that, for every binary search tree T and for every two consecutive nodes x, y in the sorted order of $V(T)$, there exists $\delta_T(y) \in \{0,1\}^*$ with $|\delta_T(y)| = \mathcal{O}(\log h(T))$ such that $D(\sigma_T(y), \delta_T(y)) = \sigma_T(x)$.

The bitstring $\delta_T(y)$ from Observation 6 is obtained as follows: It consists of a first bit indicating whether y has a left child in T or not and, in case y does have a left child, an Elias encoding $\gamma(s)$ of the value $s := d_T(y) - d_T(x) - 1$. More precisely, $\delta_T(y) = 0$ or $\delta_T(y) = 1, \gamma(s)$.

Putting some of the preceding results together we obtain the following useful coding result.

Lemma 7. *There exists a function $A : (\{0,1\}^*)^2 \to \{-1, 0, 1, \perp\}$ such that, for any $h \in \mathbb{N}$, and any $w : \{1, \ldots, h\} \to \mathbb{R}^+$ there is a prefix-free code $\alpha : \{1, \ldots, h\} \to \{0,1\}^*$ such that*

1. *for each* $i \in \{1,\dots,h\}$, $|\alpha(i)| = \log W - \log w(i) + \mathcal{O}(\log\log h)$, *where* $W := \sum_{j=1}^{h} w(j)$;
2. *for any* $i, j \in \{1,\dots,h\}$,

$$A(\alpha(i),\alpha(j)) = \begin{cases} 0 & \text{if } j = i; \\ 1 & \text{if } j = i+1; \\ -1 & \text{if } j = i-1; \\ \perp & \text{otherwise.} \end{cases}$$

Proof. Define $w' : \{1,\dots,h\} \to \mathbb{R}^+$ as $w'(i) = w(i) + W/h$ and let $W' := \sum_{i=1}^{h} w'(i) = 2W$. Using Lemma 5, construct a biased binary search tree T on $\{1,\dots,h\}$ using w' so that

$$d_T(i) \leqslant \log(2W) - \log(w(i) + W/h) \leqslant \log W - \log w(i) + 1$$

and

$$d_T(i) \leqslant \log(2W) - \log(w(i) + W/h) \leqslant \log W - \log(W/h) + 1 \leqslant \log h + 1,$$

for each $i \in \{1,\dots,h\}$. This latter inequality implies that $h(T) \leqslant \log h + 1$.

The code $\alpha(i)$ for i consists of three parts. The first part, $\gamma(|\sigma_T(i)|)$, is the Elias encoding of the length of the path $P_T(i)$ from the root to i in T. The second part $\sigma_T(i)$ encodes the left/right turns along this path. The third part, $\delta_T(i)$, is defined in Observation 6. The length of $\delta_T(i)$ is $\mathcal{O}(\log h(T)) = \mathcal{O}(\log\log h)$. Note that since γ is prefix-free and two distinct sequences $\sigma_T(i)$ and $\sigma_T(j)$ of the same length cannot be prefix of one another, the code α is also prefix-free (and thus injective).

The function A is given by a simple algorithm: Given $\alpha(i)$ and $\alpha(j)$, first observe that the values of $\gamma(\cdot)$, $\sigma_T(\cdot)$, and $\delta_T(\cdot)$ can be extracted: $\gamma(\cdot)$ is first extracted using the fact that Elias encoding is prefix-free, this then gives us the length of $\sigma_T(\cdot)$, and finally $\delta_T(\cdot)$ consists of the remaining bits. The function A extracts the values and lexicographically compares $\sigma_T(i)$ and $\sigma_T(j)$. If $\sigma_T(i) = \sigma_T(j)$, then A outputs 0. Otherwise, assume for now that $\sigma_T(i)$ is lexicographically less than $\sigma_T(j)$ so that, by Observation 4, $i < j$. Now A computes $D(\sigma_T(j), \delta_T(j)) = \sigma_T(j-1)$ as described in Observation 6. If $\sigma_T(j-1) = \sigma_T(i)$ then A outputs 1, otherwise A outputs $\perp$. In the case where $\sigma_T(i)$ is lexicographically greater than $\sigma_T(j)$, A proceeds in the same manner, but reversing the roles of i and j and outputting -1 in the case where $\sigma_T(i-1) = \sigma_T(j)$. □

2.3 Chunked Sets

For non-empty finite sets $X, Y \subset \mathbb{R}$ and an integer a, we say that X *a-chunks* Y if, for any $a+1$-element subset $S \subseteq Y$, there exists $x \in X$, such that $\min S \leqslant x \leqslant \max S$. Observe that, if X a-chunks Y, then $|Y \setminus X| \leqslant a(|X|+1) \leqslant 2a|X|$ so $|X \cup Y| \leqslant (2a+1)|X|$. A sequence $V_1,\dots,V_h \subset \mathbb{R}$ is *a-chunking* if V_y a-chunks V_{y+1} and V_{y+1} a-chunks V_y for each $y \in \{0,\dots,h-1\}$.

Lemma 8. *For any finite sets* $S_1,\dots,S_h \subset \mathbb{R}$, *there exist sets* $V_1,\dots,V_h \subset \mathbb{R}$ *such that*

1. *for each* $y \in \{1,\dots,h\}$, $V_y \supseteq S_y$;
2. $V_1,\dots,V_h$ *is 3-chunking;*
3. $\sum_{y=1}^{h} |V_y| \leqslant 2\sum_{y=1}^{h} |S_y|$.

A proof of a much more general version of Lemma 8 (with larger constants) is implicit in the iterated search structure of Chazelle and Guibas [11]. For the sake of completeness, Appendix A includes a proof of Lemma 8 that borrows heavily from the amortized analysis of partially persistent data structures [13, Section 2.3].

2.4 Product Structure Theorems

The *strong product* $A \boxtimes B$ of two graphs A and B is the graph whose vertex set is the Cartesian product $V(A \boxtimes B) := V(A) \times V(B)$ and in which two distinct vertices (x_1, y_1) and (x_2, y_2) are adjacent if and only if:

1. $x_1 x_2 \in E(A)$ and $y_1 y_2 \in E(B)$; or
2. $x_1 = x_2$ and $y_1 y_2 \in E(B)$; or
3. $x_1 x_2 \in E(A)$ and $y_1 = y_2$.

Theorem 9 (Dujmović et al. [15])**.** *Every planar graph G is a subgraph of a strong product $H \boxtimes P$ where H is a graph of treewidth at most 8 and P is a path.*

Theorem 9 can be generalized (replacing 8 with a larger constant) to bounded genus graphs, and more generally to apex-minor free graphs.

Dujmović, Morin, and Wood [16] gave analogous product structure theorems for some non-minor closed families of graphs including k-planar graphs, powers of bounded-degree planar graphs, and k-nearest-neighbour graphs of points in $\mathbb{R}^2$. Dujmović, Esperet, Morin, Walczak, and Wood [14] proved that a similar product structure theorem holds for graphs of bounded degree from any (fixed) proper minor-closed class. This is summarized in the following theorem:

Theorem 10 ([15],[14],[16])**.** *Every graph G in each of the following families of graphs is a subgraph of a strong product $H \boxtimes P$ where P is a path and H is a graph of bounded treewidth:*

- *graphs of bounded genus and, more generally, apex-minor free graphs;*
- *bounded degree graphs that exclude a fixed graph as a minor;*
- *k-planar graphs and, more generally, (g, k)-planar graphs.*

3 Bulk Trees

Our labelling scheme for a subgraph G of $H \boxtimes P$ uses labels that depend in part on the rows (H-coordinates) of G, where each row corresponds to one vertex of P: Say P consists of vertices $1, 2, \ldots, h$ in this order, then the *i-th row* of G is the subgraph H_i of G induced by the vertex set $\{(v, i) \in V(G)\}$. A naive approach to create labels for each H_i is to use a labelling scheme for bounded treewidth graphs; roughly, this entails building a specific binary search tree T_i and mapping each vertex v of H_i onto a node x of T_i that we call the *position* of v in T_i. The label of (v, i) encodes the position of v in T_i plus some small extra information (see Section 5). This way, we can determine if two vertices (v, i) and (w, i) in the same row are adjacent.

The key problems that we face here though are queries of the type (v,i) and $(w,i+1)$: We would like to determine adjacency on the H-coordinate using T_i or T_{i+1}. We could extend the node set of T_{i+1} so that it represents all vertices from H_i. This way we know that both v and w are represented in T_{i+1}. However, we still have a major issue: the label of (v,i) describes the position of v in T_i but not in T_{i+1}. In this setup, in order to determine if v and w are adjacent in H we need to know their respective positions in the same binary search tree. However, there is in principle no relation between the position of v in T_i and its position in T_{i+1}.

To circumvent this difficulty, we build the binary search trees $T_1,\dots,T_h$ one by one, starting with a balanced binary search tree, in such a way that T_{i+1} is obtained from T_i by performing carefully structured changes. By storing some small extra information related to these changes in the label of (v,i), this will allow us to obtain the position of v in T_{i+1}. Finally, we also need to guarantee that the binary search trees in our sequence are balanced so that the labels are of length $\log|T_i|$ plus a lower-order term.

In this section, we introduce three operations on a binary search tree that will allow us to carry out this plan. These operations are called *bulk insertion*, *bulk deletion*, and *rebalancing*. Starting from a perfectly balanced binary search tree T_1, each tree T_i in our sequence $T_1,\dots,T_h$ will be obtained from T_{i-1} by applying these three operations.

3.1 Bulk Insertion

The bulk insertion operation, BulkInsert(I), in which a finite set $I \subset \mathbb{R} \setminus V(T)$ of new values are inserted into a binary search tree T, is implemented as follows: Let $z_0,\dots,z_{|T|}$ denote the external nodes of T. For each $i \in \{0,\dots,|T|\}$, let I_i consist of all $x \in I$ such that $P_T(x)$ ends at z_i. For each $i \in \{0,\dots,|T|\}$, construct a perfectly balanced binary search tree T_i with vertex set I_i. For each $i \in \{1,\dots,|T|\}$, replace z_i with T_i in T^+. The resulting tree is the outcome of the operation.

Lemma 11. *Let T be any binary search tree and let I be a finite set of values from $\mathbb{R} \setminus V(T)$ such that $V(T)$ 3-chunks I.*[5] *Apply* BulkInsert(I) *to T to obtain T'. Then T' is a supergraph of T and $h(T') \leqslant h(T)+2$.*

Proof. That T' is a supergraph of T is obvious. Note that $|I_i| \leqslant 3$ since $V(T)$ 3-chunks I. Therefore, $h(T') \leqslant h(T)+2$ since, for each $i \in \{0,\dots,|T|\}$, $|T_i| = |I_i| \leqslant 3$ and T_i is perfectly balanced, so $h(T_i) \leqslant \lfloor \log 3 \rfloor = 1$. Any root-to-leaf path in T' consists of a root-to-leaf path in T followed by at most 2 elements of T_i for some $i \in \{1,\dots,|T|\}$. Therefore the length of any root-to-leaf path in T' is at most $h(T)+2$. □

Lemma 12. *Let T be any binary search tree and let I be a finite set of values from $\mathbb{R} \setminus V(T)$ such that $V(T)$ 3-chunks I. Apply* BulkInsert(I) *to T to obtain T'. Let x be any node of T and let T_x and T'_x be the subtrees of T and T', respectively, rooted at x. Then $|T_x| \leqslant |T'_x| \leqslant 8|T_x|$.*

Proof. We clearly have $|T_x| \leqslant |T'_x|$. By definition, $V(T)$ 3-chunks $I := V(T') \setminus V(T)$. This

[5] There is nothing special about the constant 3 here. The data structure and its analysis work with 3 replaced by any constant a. The constant 3 comes from an application of Lemma 8.

implies that $V(T_x)$ 3-chunks $I_x := V(T'_x) \setminus V(T_x)$. Therefore $|I_x| \leqslant 3(|T_x|+1)$, so $|T'_x| = |T_x| + |I_x| \leqslant 4|T_x| + 3 \leqslant 8|T_x|$. □

3.2 Bulk Deletion

The bulk deletion operation, BulkDelete(D), of a subset D of nodes of a binary search tree T is implemented as a series of $|D|$ individual deletions, performed in any order. For each $x \in D$, the deletion of x is implemented by running the following recursive algorithm: If x is a leaf, then simply remove x from T. Otherwise, x has at least one child. If x has a left child, then recursively delete the largest value x' in the subtree of T rooted at the left child of x and then replace x with x'. Otherwise x has a right child, so recursively delete the smallest value x' in the subtree of T rooted at the right child of x and then replace x with x'.

Lemma 13. *Let T be any binary search tree and let D be a finite set of values from $V(T)$. Apply* BulkDelete(D) *to T to obtain a new tree T'. Then, for any node x in T', $\sigma_{T'}(x)$ is a prefix of $\sigma_T(x)$. In particular, $h(T') \leqslant h(T)$.*

Proof. This follows immediately from the fact the only operations performed during a bulk deletion are (i) deletion of leaves and (ii) using a value x' to replace the value of one of its T-ancestors x. The deletion of a leaf has no effect on $\sigma_{T'}(x)$ for any node x in T'. For any node z in T' other than x', (ii) has no effect on $\sigma_T(z)$. For the node x', (ii) has the effect of replacing $\sigma_T(x')$ by its length-$d_T(x)$ prefix. □

Lemma 14. *Let T be any binary search tree and let D be a finite set of values from $V(T)$ such that $V(T) \setminus D$ 3-chunks D. Apply* BulkDelete(D) *to T to obtain a new tree T'. Then $|T|/8 \leqslant |T'| \leqslant |T|$.*

Proof. We clearly have $|T'| \leqslant |T|$. Since $V(T) \setminus D$ 3-chunks D, we have $|D| \leqslant 3(|V(T)| - |D| + 1) \leqslant 6(|V(T)| - |D|)$, so $|D| \leqslant (6/7)|V(T)|$. Thus $|T'| \geqslant |T| - 6/7|T| \geqslant |T|/7 \geqslant |T|/8$. □

3.3 Rebalancing

The rebalancing operation on a binary search tree T uses several subroutines that we now discuss, beginning with the most fundamental one: Split(x).

3.3.1 Split(x)

The argument of Split(x) is a node x in T and the end result of the subroutine is to split T into two binary search trees $T_{<x}$ and $T_{>x}$ where $V(T_{<x}) = \{z \in V(T) : z < x\}$ and $V(T_{>x}) = \{z \in V(T) : z > x\}$. Refer to Figure 3. Let $P_T(x_r) = x_0, \ldots, x_r$ be the path in T from the root x_0 of T to $x = x_r$. Partition $x_0, \ldots, x_{r-1}$ into two subsequences $a := a_1, \ldots, a_s$ and $b := b_1, \ldots, b_t$ where the elements of a are less than x and the elements of b are greater than x. Note that the properties of a binary search tree guarantee that

$$a_1 < \cdots < a_s < x < b_t < \cdots < b_1.$$

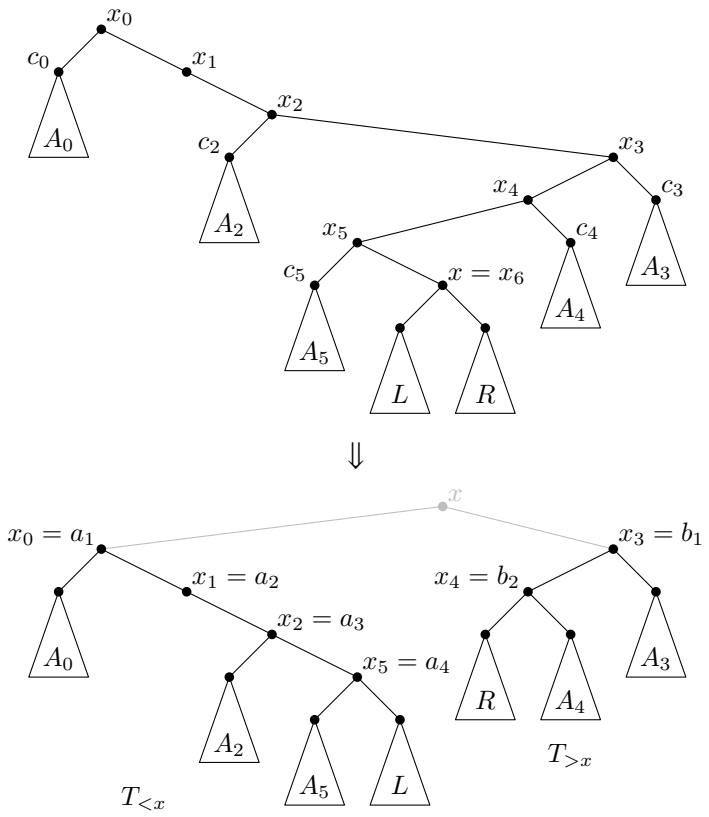

Figure 3: The operation of Split(x).

Make a binary search tree T_0 that has x as root, the path $a_1, \ldots, a_s$ as the left subtree of x and the path $b_1, \ldots, b_t$ as the right subtree of x. Note that a_{i+1} is the right child of a_i for each $i \in \{1, \ldots, s-1\}$ and b_{i+1} is the left child of b_i for each $i \in \{1, \ldots, t-1\}$.

Next, consider the forest $F := T - \{x_0, \ldots, x_r\}$. This forest consists of $r+2$ (possibly empty) trees $A_1, \ldots, A_{r-1}, L, R$ where L and R are the subtrees of T rooted at the left and right child of x in T_x and, for each $i \in \{1, \ldots, r-1\}$, A_i is the subtree of T rooted at the child $c_i \neq x_{i+1}$ of x_i (if such a child exists, otherwise A_i is empty). Make a binary search tree T_x by replacing each of the $r+2$ external nodes of T_0^+ with the corresponding tree in F. Finally, let $T_{<x}$ be the subtree of T_x rooted at the left child of x and let $T_{>x}$ be the subtree of T_x rooted at the right child of x in T_x.

Lemma 15. *Let T be any binary search tree, let x be any node of T, and apply* Split(x) *to obtain $T_{<x}$ and $T_{>x}$. Then $h(T_{<x}) \leqslant h(T)$ and $h(T_{>x}) \leqslant h(T)$.*

Proof. Note that for each node z of $T_{<x}$, we have $V(P_{T_{<x}}(z)) \subseteq V(P_T(z))$, so $d_{T_{<x}}(z) \leqslant d_T(z)$. Therefore $h(T_{<x}) \leqslant h(T)$. The argument for $T_{>x}$ is symmetric. □

The following observation shows that there is a simple relationship between a node's signature in T before calling $\textsc{Split}(x)$ and its signature in $T_{<x}$ or $T_{>x}$.

Observation 16. *Let T, x, $x_0,\ldots,x_r$, $A_1,\ldots,A_{r-1},L,R$, $a_1,\ldots,a_s$, and $b_1,\ldots,b_t$ be defined as above. Then*

1. *for each $j \in \{1,\ldots,s\}$ where $a_j = x_i$*
   (a) *$\sigma_{T_{<x}}(a_j) = 1^{j-1}$, and*
   (b) *$\sigma_{T_{<x}}(z) = 1^{j-1},0,\sigma_{A_i}(z)$ for each $z \in V(A_i)$;*
2. *for each $j \in \{1,\ldots,t\}$ where $b_j = x_i$*
   (a) *$\sigma_{T_{>x}}(b_j) = 0^{j-1}$, and*
   (b) *$\sigma_{T_{>x}}(z) = 0^{j-1},1,\sigma_{A_i}(z)$ for each $z \in V(A_i)$;*
3. *$\sigma_{T_{<x}}(z) = 1^s,\sigma_L(z)$ for each $z \in V(L)$; and*
4. *$\sigma_{T_{>x}}(z) = 0^t,\sigma_R(z)$ for each $z \in V(R)$.*

In particular, for any $z \in V(T) \setminus \{x\}$, $\sigma_{T_{<x}}(z)$ or $\sigma_{T_{>x}}(z)$ can be obtained from $\sigma_T(z)$ by deleting a prefix and replacing it with one of the $4 \cdot h(T)$ strings in $\Pi := \bigcup_{j=0}^{h(T)-1}\{0^j,0^j1,1^j,1^j0\}$.

3.3.2 $\textsc{MultiSplit}(x_1,\ldots,x_c)$

From the $\textsc{Split}(x)$ operation we build the $\textsc{MultiSplit}(x_1,\ldots,x_c)$ operation that takes as input a sequence of nodes $x_1 < \cdots < x_c$ of T. For convenience, define $x_0 = -\infty$ and $x_{c+1} = \infty$. The effect of $\textsc{MultiSplit}(x_1,\ldots,x_c)$ is to split T into a sequence of binary search trees $T_0,\ldots,T_c$ where, for each $i \in \{0,\ldots,c\}$, $V(T_i) = \{z \in V(T) : x_i < z < x_{i+1}\}$.

The implementation of $\textsc{MultiSplit}(x_1,\ldots,x_c)$ is straightforward divide-and-conquer: If $c = 0$, then there is nothing to do. Otherwise, call $\textsc{Split}(x_{\lceil c/2\rceil})$ to obtain $T_{<x_{\lceil c/2\rceil}}$ and $T_{>x_{\lceil c/2\rceil}}$. Next, apply $\textsc{MultiSplit}(x_1,\ldots,x_{\lceil c/2\rceil-1})$ to $T_{<x_{\lceil c/2\rceil}}$ to obtain $T_0,\ldots,T_{\lceil c/2\rceil-1}$ and then apply $\textsc{MultiSplit}(x_{\lceil c/2\rceil+1},\ldots,x_c)$ to $T_{>x_{\lceil c/2\rceil}}$ to obtain $T_{\lceil c/2\rceil},\ldots,T_c$.

The following lemma is immediate from Lemma 15.

Lemma 17. *Let T be any binary search tree and apply $\textsc{MultiSplit}(x_1,\ldots,x_c)$ to T to obtain $T_0,\ldots,T_c$. Then $h(T_i) \leqslant h(T)$ for each $i \in \{0,\ldots,c\}$.*

3.3.3 $\textsc{Balance}(x,k)$

The $\textsc{Balance}(x,k)$ operation operates on the subtree T_x of T rooted at some node x in T. The goal of this operation is to balance the size of all the subtrees rooted at nodes of depth $d_T(x)+k+1$ and contained in T_x. Refer to Figure 4.

If $|V(T_x)| < 2^k$, then this operation simply replaces T_x with a perfectly balanced binary search tree containing $V(T_x)$. Otherwise, let $Z := \{z \in V(T_x) : d_{T_x}(z) < k\}$. Call the $m \leqslant 2^k - 1$ elements of Z $z_1 < z_2 < \cdots < z_m$ and, for convenience, define $z_0 = -\infty$ and $z_{m+1} = \infty$.

Select the nodes $X := \{x_1,\ldots,x_{2^k-1}\}$ of T_x where each x_j has rank $\lfloor j|V(T_x)|/2^k\rfloor$ in $V(T_x)$.[6] The $\textsc{Balance}(x,k)$ operation will turn T_x into a tree with a top part $\hat{T}_0$ that is a perfectly balanced binary search tree on $Z \cup X$. We now describe how this is done.

[6]For a finite set $X \subset \mathbb{R}$, and $x \in \mathbb{R}$, the *rank* of x in S is $|\{x' \in S : x' < x\}|$.

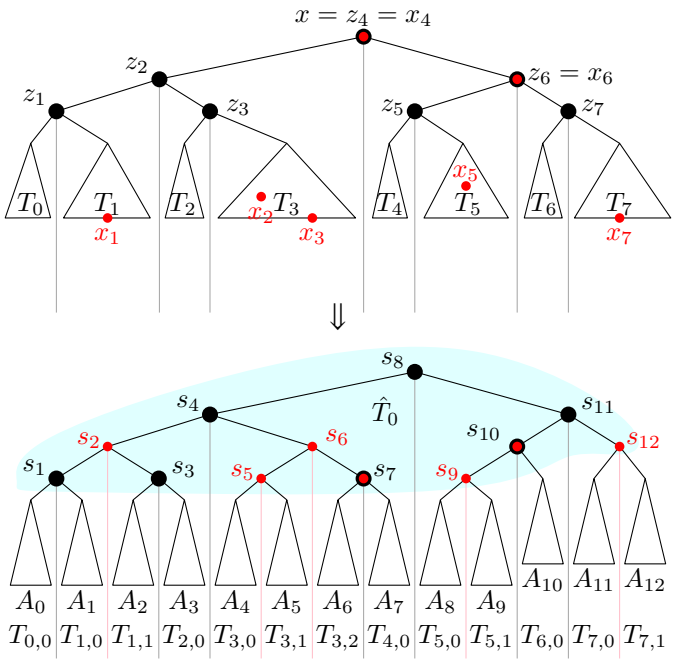

Figure 4: The operation of Balance(x,k)

$T_x - Z$ is a forest consisting of $m+1 \leqslant 2^k$ trees $T_0,\dots,T_m$. (Some of these trees may be empty.) Order $T_0,\dots,T_m$ so that, for each $i \in \{0,\dots,m\}$ and each $x' \in V(T_i)$, $z_i < x' < z_{i+1}$. For each $i \in \{0,\dots,m\}$, let $\{x_{i,1},\dots,x_{i,c_i}\} := X \cap V(T_i)$ where $x_{i,1} < \cdots < x_{i,c_i}$ and define $x_{i,0} := z_i$ and $x_{i,c_i+1} := z_{i+1}$. Note that for each $i \in \{0,\dots,m\}$, $c_i \leqslant |X| \leqslant 2^k - 1$.

For each $i \in \{0,\dots,m\}$, apply MultiSplit$(x_{i,1},\dots,x_{i,c_i})$ to the tree T_i. As a result of these calls, we obtain sequences of trees $T_{i,0},\dots,T_{i,c_i}$ where, for each $i \in \{0,\dots,m\}$, each $j \in \{0,\dots,c_i\}$, and each $x' \in V(T_{i,j})$, we have $x_{i,j} < x' < x_{i,j+1}$. Note that if $c_i = 0$ (i.e., if X does not intersect $V(T_i)$), then the result of this call is a single tree $T_{i,0} = T_i$. Observe that $\bigcup_{i=0}^{m}\bigcup_{j=0}^{c_i} V(T_{i,j}) = V(T_x) \setminus (Z \cup X)$.

Let $p := |Z \cup X|$, let $s_1 < \cdots < s_p$ denote the elements of $Z \cup X$ and define $s_0 := -\infty$ and $s_{p+1} := \infty$. For each $\ell \in \{0,\dots,p\}$, let $i_\ell := |Z \cap \{s_1,\dots,s_\ell\}|$ and $j_\ell := \ell - \max\{q \in \{1,\dots,\ell\} : s_q \in Z\}$ and let $A_\ell := T_{i_\ell,j_\ell}$. Then, for each $\ell \in \{0,\dots,p\}$ and each $x' \in V(A_\ell)$, we have $s_\ell < x' < s_{\ell+1}$.

Now construct a perfectly balanced tree $\hat{T}_0$ with vertex set $V(\hat{T}_0) := \{s_1,\dots,s_p\} = Z \cup X$. The tree $\hat{T}_0$ has $p+1$ external nodes $a_0,\dots,a_p$. We obtain a new tree T'_x by replacing a_ℓ with A_ℓ for each $\ell \in \{0,\dots,p\}$ in $\hat{T}_0^+$. In the encompassing bulk tree T we replace the subtree T_x with T'_x.

Lemma 18. *Let T be any binary search tree, let x be any node of T, and apply* Balance(x,k) *to T to obtain a new tree T'. Then $h(T') \leqslant h(T) + 1$.*

Proof. Since Balance(x,k) only affects the subtree T_x rooted at x, it suffices to show that

$h(T'_x) \leqslant h(T_x) + 1$. For each $i \in \{0,\dots,m\}$, T_i is rooted at a depth-k node of T_x, so $h(T_i) \leqslant h(T_x) - k$. For each $\ell \in \{0,\dots,p\}$, A_ℓ is obtained by an application of MultiSplit to T_i for some $i \in \{0,\dots,m\}$ so, by Lemma 17, $h(A_\ell) \leqslant h(T_i) \leqslant h(T_x) - k$. Next, $|Z \cup X| \leqslant |Z| + |X| \leqslant 2^k - 1 + 2^k - 1 < 2^{k+1} - 1$ and $\hat{T}_0$ is a perfectly balanced binary search tree of size $|\hat{T}_0| = |Z \cup X|$. Therefore $h(\hat{T}_0) \leqslant \lfloor \log |\hat{T}_0| \rfloor \leqslant \lfloor \log(2^{k+1} - 1) \rfloor = k$. Finally, $h(T'_x) \leqslant h(\hat{T}_0) + 1 + \max\{h(A_\ell) : \ell \in \{0,\dots,p\}\} \leqslant k + 1 + h(T_x) - k \leqslant h(T_x) + 1$. □

The following statement captures what we win after an application of Balance(x,k) to a binary search tree.

Lemma 19. *Let T be any binary search tree, let x be any node of T, let T_x be the subtree of T rooted at x, and apply* Balance(x,k) *to T to obtain a new tree T'. Then, for each T'-descendant z of x with $d_{T'}(z) = d_T(x) + k + 1$, the subtree of T' rooted at z has size at most $|T_x|/2^k$.*

Proof. Each such subtree is a subtree of A_ℓ for some $\ell \in \{0,\dots,p\}$. Now, $V(A_\ell) \subset (x_j, x_{j+1})$ for some $j \in \{1,\dots,2^{k-1}\}$. The values x_j and x_{j+1} have ranks $\lfloor j|T_x|/2^k \rfloor$ and $\lfloor (j+1)|T_x|/2^k \rfloor$ in the set $V(T_x)$. Therefore, $|A_\ell| \leqslant \lfloor (j+1)|T_x|/2^k \rfloor - \lfloor j|T_x|/2^k \rfloor - 1 < |T_x|/2^k$. □

3.3.4 BulkBalance(θ,k)

The ultimate restructuring operation in bulk trees is BulkBalance(θ,k). It calls Balance(x,k) for each node x of depth θ in T. (Note that this operation has no effect if there is no such node.) The following two lemmas are immediate consequences of Lemma 18 and Lemma 19, respectively.

Lemma 20. *Let T be any binary search tree and apply the* BulkBalance(θ,k) *operation to obtain a new tree T'. Then $h(T') \leqslant h(T) + 1$.*

Lemma 21. *Let T be any binary search tree and apply the* BulkBalance(θ,k) *operation to obtain a new tree T'. Let x be any node of T of depth θ and let T_x be the subtree of T rooted at x. Then, for each T'-descendant z of x with $d_{T'}(z) = \theta + k + 1$, the subtree of T' rooted at z has size at most $|T_x|/2^k$.*

3.4 Bulk Tree Sequences

Let $k \geqslant 7$ be an integer[7] and let $S_0,\dots,S_q$ be a 3-chunking sequence. We define a *one-phase k-bulk tree sequence* based on $S_0,\dots,S_q$ to be a sequence $T_0,\dots,T_q$ of binary search trees such that T_0 is an arbitrary binary search tree on node set S_0 and, for each $y \in \{0,\dots,q-1\}$, we have $h(T_y) > y \cdot (k+1)$ and the tree T_{y+1} is obtained from T_y by applying

(i) BulkBalance$(y \cdot (k+1), k)$, then
(ii) BulkInsert(I) with $I := S_{y+1} \setminus S_y$, and finally
(iii) BulkDelete(D) with $D := S_y \setminus S_{y+1}$.

[7] $k \geqslant 7$ is a technical requirement, to make sure that some inequalities hold later on.

Note that $V(T_y) = S_y$ for each $y \in \{0,\dots,q\}$. The sequence is *complete* if $h(T_q) \leqslant q \cdot (k+1)$.

For $k \geqslant 7$ and a 3-chunking sequence $S_1,\dots,S_h$, we define a *k-bulk tree sequence* based on $S_1,\dots,S_h$ to be a sequence $T_1,\dots,T_h$ of binary search trees satisfying: T_1 is a perfectly balanced binary search tree with $V(T_1) = S_1$, and there exist indices $h_1, h_2, \dots, h_\ell$ with $1 = h_1 < h_2 < \cdots < h_\ell = h$ such that $T_{h_j}, T_{h_j+1}, \dots, T_{h_{j+1}}$ is a complete one-phase k-bulk tree sequence based on $S_{h_j}, S_{h_j+1}, \dots, S_{h_{j+1}}$ for each $j \in \{1,\dots,\ell-2\}$, and $T_{h_{\ell-1}}, T_{h_{\ell-1}+1}, \dots, T_{h_\ell}$ is a (non-necessarily complete) one-phase k-bulk tree sequence based on $S_{h_{\ell-1}}, S_{h_{\ell-1}+1}, \dots, S_{h_\ell}$.

Note that if we fix the 3-chunking sequence $S_1,\dots,S_h$, the integer $k \geqslant 7$, and the starting perfectly balanced binary search tree T_1 with $V(T_1) = S_1$, a k-bulk tree sequence based on $S_1,\dots,S_h$ and starting with T_1 exists and is unique. It is obtained by a sequence of one-phase k-bulk tree sequences, where we start a new one-phase sequence as soon as the current one is complete.

This will not be needed until the final sections, but it is helpful to keep in mind that we will ultimately take $k = \max\left\{7, \left\lceil \sqrt{\log n / \log\log n} \right\rceil\right\}$ when considering a k-bulk tree sequence built for our n-vertex graph G, so that the expression $\mathcal{O}(k + k^{-1}\log n)$ (which appears many times in what follows), is $\omega(1)$ and $o(\log n)$.

Lemma 22. *Let $T_0,\dots,T_q$ be a one-phase k-bulk tree sequence. Then, for each $y \in \{0,\dots,q\}$*

(i) $h(T_y) \leqslant h(T_0) + 3y$;

(ii) each subtree of T_y rooted at a node of depth $y \cdot (k+1)$ has size at most $|T_0| \cdot 2^{-y(k-3)}$.

Proof. The proof is by induction on y. For the base case $y = 0$, both properties are trivial: (i) asserts that $h(T_0) \leqslant h(T_0)$ and (ii) asserts that the subtree of T_0 rooted at the root of T_0 has size at most $|T_0|$.

For the inductive step, assume $y + 1 \geqslant 0$ and (i) holds for T_y. In order to get T_{y+1}, we first apply BulkBalance$(y \cdot (k+1), k)$ to T_y to obtain T'. By Lemma 20, we have $h(T') \leqslant h(T_y) + 1$. Let $I := V(T_{y+1}) \setminus V(T') = V(T_{y+1}) \setminus V(T_y)$. Since $V(T_y)$ 3-chunks $V(T_{y+1})$ we know that $V(T')$ 3-chunks I. Next we apply BulkInsert(I) to T' to obtain T''. Thus, by Lemma 11 we have $h(T'') \leqslant h(T') + 2$. Finally, we apply BulkDelete(D) to T'' and obtain T_{y+1}, where $D := V(T_y) \setminus V(T_{y+1})$. By Lemma 13, we have $h(T_{y+1}) \leqslant h(T'')$. Altogether we have

$$h(T_{y+1}) \leqslant h(T'') \leqslant h(T') + 2 \leqslant h(T_y) + 3 \leqslant h(T_0) + 3y + 3 = h(T_0) + 3(y+1).$$

Thus, (i) holds for T_{y+1}.

Next we establish (ii). Assume that (ii) holds for T_y. Thus, every subtree of T_y rooted at a node of depth $y(k+1)$ has size at most $|T_0| \cdot 2^{-y(k-3)}$. Again, the first step when constructing T_{y+1} from T_y is to apply BulkBalance$(y(k+1), k)$ to T_y. By Lemma 21, this results in a tree T' in which every subtree rooted at a node of depth $(y+1)(k+1)$ has size at most $|T_0| \cdot 2^{-y(k-3)} \cdot 2^{-k}$. The second step is to apply BulkInsert(I) to T' to obtain a new tree T''. By Lemma 12, every subtree of T'' rooted at a node of depth $(y+1)(k+1)$ has size at most $|T_0| \cdot 2^{-y(k-3)} \cdot 8 \cdot 2^{-k}$. Finally, the third step is to perform BulkDelete(D) on T'' to obtain T_{y+1}. Bulk deletion does not increase the size of any subtree, so every subtree of T_{y+1} rooted at a node of depth $(y+1)(k+1)$ has size at most $|T_0| \cdot 2^{-(y+1)(k-3)}$, as desired. □

Corollary 23. *Let $T_0,\dots,T_q$ be a one-phase k-bulk tree sequence. Then,*

$$q \leqslant \left\lceil \tfrac{\log|T_0|}{k-3} \right\rceil.$$

Proof. Arguing by contradiction, suppose that $q > \left\lceil \frac{\log|T_0|}{k-3} \right\rceil$. Note that when we take the logarithm of the upper bound in Lemma 22(ii) for $y := \left\lceil \frac{\log|T_0|}{k-3} \right\rceil$, we have

$$\log|T_0| - \left\lceil \tfrac{\log|T_0|}{k-3} \right\rceil \cdot (k-3) \leqslant \log|T_0| - \tfrac{\log|T_0|}{k-3} \cdot (k-3) \leqslant 0.$$

Thus, each subtree of T_y rooted at a node of depth $y(k+1)$ has size at most

$$|T_0| \cdot 2^{-y(k-3)} \leqslant 1,$$

and hence $h(T_y) \leqslant y(k+1)$, which violates the height condition in the definition of a one-phase k-bulk tree sequence. □

Lemma 24. *Let $T_0,\dots,T_q$ be a complete one-phase k-bulk tree sequence, and let $r_0 := h(T_0) - \log|T_0|$. Then, for each $y \in \{0,\dots,q\}$,*

(i) $|T_0|/8^y \leqslant |T_y|$, *and thus* $\log|T_0| \leqslant \log|T_y| + 3y$;
(ii) $q = \mathcal{O}(k^{-1}\log|T_y|)$;
(iii) $h(T_y) = \log|T_y| + r_0 + \mathcal{O}(k^{-1}\log|T_y|)$; *and*
(iv) $h(T_q) = \log|T_q| + \mathcal{O}(k + k^{-1}\log|T_q|)$.

Proof. Let $I_0,\dots,I_{q-1}$ and $D_0,\dots,D_{q-1}$ be the sets so that T_{y+1} is obtained from T_y by rebalancing and then applying BulkInsert(I_y) and BulkDelete(D_y), for each $y \in \{0,\dots,q-1\}$. First, recall that by Lemma 12 and Lemma 14 we have $|T_{y+1}| \geqslant |T_y|/8$. Iterating this starting with T_0 implies that $|T_0|/8^y \leqslant |T_y|$, and thus $\log|T_0| \leqslant \log|T_y| + 3y$ for each $y \in \{0,\dots,q\}$, which proves (i).

By Corollary 23, we have $q \leqslant \left\lceil \frac{\log|T_0|}{k-3} \right\rceil$. Note that, for each $y \in \{0,\dots,q\}$, we have

$$q \leqslant \tfrac{\log|T_0|}{k-3} + 1 \leqslant \tfrac{\log|T_y|+3q}{k-3} + 1, \qquad \text{(by (i), and since } y \leqslant q)$$

and rewriting this yields (using that $k \geqslant 7$)

$$\begin{aligned} q &\leqslant \tfrac{\log|T_y|}{k-6} + \tfrac{k-3}{k-6} \leqslant \tfrac{k}{k-6} \cdot \tfrac{\log|T_y|}{k} + \tfrac{k-3}{k-6} \\ &\leqslant 7 \cdot \tfrac{\log|T_y|}{k} + 4 = \mathcal{O}(k^{-1}\log|T_y|), \end{aligned}$$

which proves (ii). (iii) follows as for each $y \in \{0,\dots,q\}$, we have

$$\begin{aligned} h(T_y) &\leqslant h(T_0) + 3y && \text{(by Lemma 22(i))} \\ &= \log|T_0| + 3y + r_0 \\ &\leqslant \log|T_y| + 6y + r_0 && \text{(by (i))} \\ &\leqslant \log|T_y| + 6q + r_0 && \text{(since } y \leqslant q) \\ &= \log|T_y| + \mathcal{O}(k^{-1}\log|T_y|) + r_0. && \text{(by (ii))} \end{aligned}$$

(iv) follows from

$$\begin{aligned}
h(T_q) &\leqslant (k+1)q && \text{(since the sequence is complete)}\\
&\leqslant (k+1)\left(\tfrac{\log|T_0|}{k-3}+1\right)\\
&= \left(\tfrac{k+1}{k-3}\right)\cdot\log|T_0| + (k+1)\\
&\leqslant \left(\tfrac{k+1}{k-3}\right)\cdot(\log|T_q|+3q)+(k+1) && \text{(by (i))}\\
&= \log|T_q| + \tfrac{4}{k-3}\log|T_q| + 3\cdot\tfrac{k+1}{k-3}q + k + 1\\
&= \log|T_q| + \mathcal{O}(k+k^{-1}\log|T_q|). && \text{(as } \tfrac{k+1}{k-3}\leqslant\tfrac{8}{5}\text{, and by (ii))}
\end{aligned}$$

□

The following lemma shows that trees in a bulk tree sequence are balanced at all times:

Lemma 25. *Let $T_1,\dots,T_h$ be a k-bulk tree sequence and let $y\in\{1,\dots,h\}$. Then*

$$h(T_y)\leqslant \log|T_y| + \mathcal{O}(k+k^{-1}\log|T_y|).$$

Proof. By the definition of a k-bulk tree sequence, T_1 is a perfectly balanced binary tree so $h(T_1)=\lceil\log|T_1|\rceil$ and the statement is satisfied for $y=1$. Let $h_1,h_2,\dots,h_\ell$ be indices with $1=h_1<h_2<\cdots<h_\ell=h$ such that $T_{h_j},\dots,T_{h_{j+1}}$ is a complete one-phase k-bulk tree sequence for each $j\in\{1,\dots,\ell-2\}$, and $T_{h_{\ell-1}},\dots,T_{h_\ell}$ is a one-phase k-bulk tree sequence. Let $Y:=\{h_1,h_2,\dots,h_{\ell-1}\}$. For $y\in Y\setminus\{1\}$, Lemma 24(iv) implies that T_y satisfies the conditions of the lemma.

All that remains is to show that the conditions of the lemma are satisfied for each $y\in\{1,\dots,h\}\setminus Y$. To show this, let $y_0:=\max\{y'\in Y : y'<y\}$. That is, T_{y_0} is the tree that began the one-phase k-bulk tree sequence in which T_y takes part. In this case, Lemma 24(iii) implies that

$$h(T_y)\leqslant\log|T_y|+\mathcal{O}(k^{-1}\log|T_y|)+h(T_{y_0})-\log|T_{y_0}|.$$

Thus, all that is required is to show that $r_0:=h(T_{y_0})-\log|T_{y_0}|\in\mathcal{O}(k+k^{-1}\log|T_y|)$ so that is what we do. Note that by Lemma 24(ii) we have $y-y_0=\mathcal{O}(k^{-1}\log|T_y|)$.

$$\begin{aligned}
r_0 &= h(T_{y_0})-\log|T_{y_0}|\\
&= \mathcal{O}(k+k^{-1}\log|T_{y_0}|) && \text{(by Lemma 24(iv))}\\
&= \mathcal{O}(k+k^{-1}(\log|T_y|+3(y-y_0))) && \text{(by Lemma 24(i))}\\
&= \mathcal{O}(k+k^{-1}\log|T_y|+k^{-2}\log|T_y|) && \text{(by Lemma 24(ii))}\\
&= \mathcal{O}(k+k^{-1}\log|T_y|).
\end{aligned}$$

□

3.5 Transition Codes for Nodes

We now arrive at the *raison d'être* of bulk tree sequences: For two consecutive trees T_y and T_{y+1} in a bulk tree sequence and any $z\in V(T_y)\cap V(T_{y+1})$, the signatures $\sigma_{T_y}(z)$ and $\sigma_{T_{y+1}}(z)$

are so closely related that $\sigma_{T_{y+1}}(z)$ can be derived from $\sigma_{T_y}(z)$ and a short *transition code* $\nu_y(z)$. The following two lemmas make this precise.

Lemma 26. *There exists a function $B : (\{0,1\}^*)^2 \to \{0,1\}^*$ such that, for every binary search tree T, for any integers θ and k with $1 \leqslant \theta \leqslant h(T)$ and $k \geqslant 1$, the following holds. Let T' be the binary search tree obtained by an application of* BulkBalance(θ, k) *to T. For each $z \in V(T)$, there exists $\nu(z) \in \{0,1\}^*$ with $|\nu(z)| = \mathcal{O}(k \log h(T))$ such that $B(\sigma_T(z), \nu(z)) = \sigma_{T'}(z)$.*

Proof. Recall that by Lemma 20, $h(T') \leqslant h(T)+1$. Recall also that $\gamma : \mathbb{N} \to \{0,1\}^*$ is a prefix-free encoding of the natural numbers such that $|\gamma(i)| = \mathcal{O}(\log i)$, for every natural number i as in Lemma 3.

BulkBalance(θ, k) calls Balance(x, k) for each node x of depth θ in T. Recall that the changes caused by Balance(x, k) are limited to the subtree of T rooted at x. Thus, $\sigma_T(z)$ can be affected by Balance(x, k) only if x is a T-ancestor of z. In particular when $|\sigma_T(z)| < \theta$, the signature of z does not change, that is, $\sigma_T(z) = \sigma_{T'}(z)$. In this case, we define

$$\nu(z) := \gamma(\theta).$$

Note that in this case $|\nu(z)| = \mathcal{O}(\log \theta) = \mathcal{O}(\log h(T))$.

Assume now that $|\sigma_T(z)| \geqslant \theta$ and let x be the T-ancestor of z at depth θ. Recall that the application of Balance(x, k) first identifies two sets of nodes Z and X that eventually form a perfectly balanced tree $\hat{T}_0$ of height at most k which forms the top part of the subtree that replaces the subtree of x in T. This means that if $z \in Z \cup X$ then $\sigma_{T'}(z) = \sigma_T(x), \sigma_{\hat{T}_0}(z)$. In this case, we define

$$\nu(z) := \gamma(\theta), 0, \gamma(|\sigma_{\hat{T}_0}(z)|), \sigma_{\hat{T}_0}(z).$$

Note that in this case $|\nu(z)| = \mathcal{O}(\log \theta) + \mathcal{O}(\log k) + \mathcal{O}(k) = \mathcal{O}(\log h(T) + k)$.

Now we are left with the case that $|\sigma_T(z)| \geqslant \theta$ and $z \notin Z \cup X$. In particular, the node z lies in some tree T_i of the forest $T_x - Z$ where T_x is the subtree of T rooted at x. (Further on we reuse the notations T_i, $T_{i,0}, \ldots, T_{i,c_i}$, etc. introduced in the definition of Balance(x, k).) Recall that Balance(x, k) calls MultiSplit$(x_{i,1}, \ldots, x_{i,c_i})$ on T_i to obtain a sequence of trees $T_{i,0}, \ldots, T_{i,c_i}$ and the node z ends up in one of these trees, say in $T_{i,a}$. Note that

(i) $\sigma_{T_i}(z)$ is a suffix of $\sigma_T(z)$;

(ii) the application of MultiSplit$(x_{i,1}, \ldots, x_{i,c_i})$ to T_i calls Split$(x_{\lceil c_i/2 \rceil})$ (given $c_i > 0$) to obtain two trees $T_{<x_{\lceil c_i/2 \rceil}}$ and $T_{>x_{\lceil c_i/2 \rceil}}$, and then recursively calls MultiSplit$(x_{i,1}, \ldots, x_{i,\lceil c_i/2 \rceil - 1})$ on $T_{<x_{i,\lceil c_i/2 \rceil}}$ and MultiSplit$(x_{i,\lceil c_i/2 \rceil + 1}, \ldots, x_{i,c_i})$ on $T_{>x_{i,\lceil c_i/2 \rceil}}$; the node z lies in one of the trees $T_{<x_{i,\lceil c_i/2 \rceil}}$, $T_{>x_{i,\lceil c_i/2 \rceil}}$ and by Observation 16 the signature of z in the new tree can be obtained from $\sigma_{T_i}(z)$ by deleting a prefix and replacing it with one of the $4h(T_i)$ strings in $\bigcup_{j=0}^{h(T_i)-1} \{0^j, 0^j1, 1^j, 1^j0\}$;

(iii) the application of MultiSplit$(x_{i,1}, \ldots, x_{i,c_i})$ thus defines a sequence of trees starting with T_i and ending with $T_{i,a}$ that all contain z; by Lemma 15 the height of each of these trees is at most $h(T_i)$; the signature of z in these trees, which is $\sigma_{T_i}(z)$ at the beginning, undergoes at most $1 + \log c_i \leqslant 1 + k$ changes before becoming $\sigma_{T_{i,a}}(z)$; let b denote the number of these changes and, for each $j \in \{1, \ldots, b\}$, let d_j be the length of the

prefix of the signature being deleted during the j-th change and let $q_j \in \{1, \ldots, 4h(T_i)\}$ be a number identifying the string that this prefix is replaced with during the j-th change (here we use that all the trees in the sequence have height at most $h(T_i)$).

Finally, BALANCE(x,k) replaces the external nodes of $\hat{T}_0$ with the trees output by MULTISPLIT$(x_{i,1}, \ldots, x_{i,c_i})$. Let z' be the external node of $\hat{T}_0$ that is replaced with $T_{i,a}$. Therefore, the signature of z in T' is the concatenation of $\sigma_T(x)$, $\sigma_{\hat{T}_0}(z')$, and $\sigma_{T_{i,a}}(z)$. We define $\nu(z)$ in this case as follows.

$$\nu(z) := \gamma(\theta), 1, \gamma(|\sigma_{\hat{T}_0}(z')|), \sigma_{\hat{T}_0}(z'), \gamma(b), \gamma(d_1), \gamma(q_1), \ldots, \gamma(d_b), \gamma(q_b).$$

Note that in this case $|\nu(z)| = \mathcal{O}(\log\theta) + \mathcal{O}(\log k) + \mathcal{O}(k) + \mathcal{O}(\log k) + \mathcal{O}(2k \cdot \log h(T)) = \mathcal{O}(k \log h(T))$. This completes the definition of $\nu(z)$.

The function B is defined as expected: Given $\sigma_T(z)$ and $\nu(z)$, the function B first decodes θ and checks whether $|\sigma_T(z)| < \theta$. If this is the case then the signatures of z in T and T'' are the same, so B outputs $\sigma_T(z)$. If $|\sigma_T(z)| \geqslant \theta$ then B reads the next bit of $\nu(z)$. If it is 0 then this corresponds to the case where $z \in Z \cup X$ described above, and the information encoded after is enough to recover and output $\sigma_{T'}(z) = \sigma_T(x), \sigma_{\hat{T}_0}(z)$. If the bit under consideration was 1, then this corresponds to the case where $z \notin Z \cup X$. Recall that the set Z are just all nodes in T_x of depths less than k. Thus, just removing first $\theta + k$ bits of $\sigma_T(z)$, the function B obtains $\sigma_{T_i}(z)$ where T_i is aforementioned subtree of $T_x - Z$ containing z. The function B then reads the rest of the information $\nu(z)$, which allows it to follow all the changes made to $\sigma_{T_i}(z)$ until it becomes $\sigma_{T_{i,a}}(z)$ of the signature that are described above. This way again B computes and outputs $\sigma_{T'}(z) = \sigma_T(x), \sigma_{\hat{T}_0}(z'), \sigma_{T_{i,a}}(z)$. This completes the definition of B and the proof of the lemma. □

Lemma 27. *There exists a function $B' : (\{0,1\}^*)^2 \to \{0,1\}^*$ such that, for each k-bulk tree sequence $T_1, \ldots, T_h$, each $y \in \{1, \ldots, h-1\}$, and each $z \in V(T_y) \cap V(T_{y+1})$, there exists $\nu_y(z) \in \{0,1\}^*$ with $|\nu_y(z)| = \mathcal{O}(k \log h(T_y))$ such that $B'(\sigma_{T_y}(z), \nu_y(z)) = \sigma_{T_{y+1}}(z)$.*

Proof. Let $T_1, \ldots, T_h$ be a k-bulk tree sequence and let $y \in \{1, \ldots, h-1\}$. Let $I := V(T_{y+1}) \setminus V(T_y)$ and $D := V(T_y) \setminus V(T_{y+1})$. Recall that the transformation of T_y into T_{y+1} occurs in three steps: applying BULKBALANCE(θ, k) to T_y with the appropriate value of θ to obtain T', applying BULKINSERT(I) to T' to obtain T'', and applying BULKDELETE(D) to T'' to obtain T_{y+1}. Recall that whenever BULKBALANCE(θ, k) is applied in this context we have $\theta < h(T_y)$.

By Lemma 13, Lemma 11, and Lemma 20 we have $h(T_{y+1}) \leqslant h(T'') \leqslant h(T') + 2 \leqslant h(T_y) + 3$. Thus the heights of all these trees are $\mathcal{O}(h(T_y))$.

Given a node z appearing in both T_y and T_{y+1} we are going to describe $\nu_y(z)$. The transition code $\nu_y(z)$ consists of two parts $\nu_y^{\text{BAL}}(z)$ and $\nu_y^{\text{DEL}}(z)$ devoted to different steps of the transformation from T_y to T_{y+1}.

The first part $\nu_y^{\text{BAL}}(z)$ is simply defined as

$$\nu_y^{\text{BAL}}(z) := \nu(z),$$

where $\nu(z)$ is given by an application of Lemma 26 with $T = T_y$.

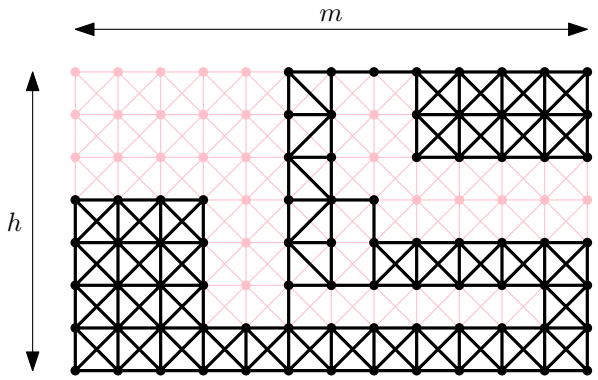

Figure 5: The special case where G is a subgraph of $P_1 \boxtimes P_2$.

Recall that the bulk insertion of new nodes in T' does not affect the signature of existing nodes in the tree, since new nodes are inserted at the leaves of T'.

We next describe $\nu_y^{\text{DEL}}(z)$ that serves to reconstruct $\sigma_{T_{y+1}}(z)$ from $\sigma_{T''}(z)$. This turns out to be fairly easy. By Lemma 13 we have that $\sigma_{T_{y+1}}(z)$ is just a prefix of $\sigma_{T''}(z)$. Therefore, it is enough to define

$$\nu_y^{\text{DEL}}(z) := \gamma(|\sigma_{T_{y+1}}(z)|).$$

Finally, we define $\nu_y(z)$ to be the concatenation of $\nu_y^{\text{BAL}}(z)$ and $\nu_y^{\text{DEL}}(z)$. It follows from that $|\nu_y(z)| = \mathcal{O}(k \log h(T_y))$.

The function B' is defined as expected: Given $\sigma_{T_y}(z)$ and $\nu_y(z)$, the function B' first decodes $\nu(z)$ and computes $B(\sigma_{T_y}(z), \nu(z)) = \sigma_{T'}(z) = \sigma_{T''}(z)$. Then B' decodes $|\sigma_{T_{y+1}}(z)|$ and computes a prefix of this size of $\sigma_{T''}(z)$. As we have seen, the prefix is $\sigma_{T_{y+1}}(z)$, which is output by B'. □

4 Subgraphs of $P \boxtimes P$

Before continuing, we show that using the techniques developed thus far, we can already solve a non-trivial special case. In particular, we consider the case in which G is an n-vertex subgraph of $P_1 \boxtimes P_2$ where P_1 is a path on m vertices and P_2 is a path on h vertices. Thus, we identify each vertex of G with a point $(x, y) \in \{1, \ldots, m\} \times \{1, \ldots, h\}$ in the $m \times h$ grid with diagonals, and G is just a subgraph of this grid, see Figure 5. Obviously, we may assume that $m \leqslant n$ and $h \leqslant n$.

Our motivation for considering this special case is expository: The vertices of P_1 are integers $1, \ldots, m$ that can be stored directly in a binary search tree. This makes it easier to understand the role that bulk tree sequences play in our solution. The extension of this solution to subgraphs of $H \boxtimes P$, which is the topic of Section 5, uses exactly the same ideas but requires another level of indirection since there is no natural mapping from the vertices of H onto real numbers.

4.1 The Labels

For each $y \in \{1,\dots,h\}$, we let

$$L_y = \{x : (x,y) \in V(G)\}, \text{ and}$$
$$L_y^+ = L_y \cup \{x-1 : (x,y) \in V(G)\}.$$

Note that $\sum_{y=1}^{h} |L_y| = n$ and $\sum_{y=1}^{h} |L_y^+| \leqslant 2n$. Let $L_0^+ := \emptyset$.

Let $V_1,\dots,V_h$ be the 3-chunking sequence obtained by applying Lemma 8 to the sequence $L_1^+ \cup L_0^+,\dots,L_h^+ \cup L_{h-1}^+$. Thus for each $y \in \{1,\dots,h\}$, we have

$$V_y \supseteq L_y^+ \cup L_{y-1}^+, \text{ and}$$
$$\textstyle\sum_{y=1}^{h} |V_y| \leqslant 2\sum_{y=1}^{h} |L_y^+ \cup L_{y-1}^+| \leqslant 8n.$$

Next, let $T_1,\dots,T_h$ be a k-bulk tree sequence based on $V_1,\dots,V_h$ (recall that if we fix the starting perfectly balanced binary search tree T_1 with vertex set V_1, this sequence exists and is unique). We discuss the asymptotically optimal choice for the value of k at the end of the section. By Lemma 25, for each $y \in \{1,\dots,h\}$, we have

$$\begin{aligned} h(T_y) &= \log|T_y| + \mathcal{O}(k + k^{-1}\log|T_y|) \\ &\leqslant \log|T_y| + \mathcal{O}(k + k^{-1}\log n). \end{aligned}$$

Let $A : (\{0,1\}^*)^2 \to \{0,1\}^*$ be the function, given by Lemma 7 such that using the weight function $w(y) := |T_y|$ for each $y \in \{1,\dots,h\}$, we have a prefix-free code $\alpha : \{1,\dots,h\} \to \{0,1\}^*$ such that

$$\begin{aligned} |\alpha(y)| &= \log\left(\textstyle\sum_{i=1}^{h} |T_i|\right) - \log|T_y| + \mathcal{O}(\log\log h) \\ &\leqslant \log n - \log|T_y| + \mathcal{O}(\log\log n), \end{aligned}$$

for each $y \in \{1,\dots,h\}$, and $A(\alpha(i),\alpha(j))$ outputs 0, 1, -1, or $\perp$, depending whether the value of j is i, $i+1$, $i-1$, or some other value, respectively.

Let $B' : (\{0,1\}^*)^2 \to \{0,1\}^*$ be the function, given by Lemma 27, such that for each $y \in \{1,\dots,h-1\}$ and each $x \in L_y \subseteq V(T_y) \cap V(T_{y+1})$, there exists a code $\nu_y(x)$ with $|\nu_y(x)| = \mathcal{O}(k\log h(T_y)) = \mathcal{O}(k\log\log n + k\log k)$ such that $B'(\sigma_{T_y}(x), \nu_y(x)) = \sigma_{T_{y+1}}(x)$.

Let $D : (\{0,1\}^*)^2 \to \{0,1\}^*$ be the function, given by Observation 6, such that for every binary search tree T, and every i such that $i-1$ and i are in T, there exists $\delta_T(i) \in \{0,1\}^*$ with $|\delta_T(i)| = \mathcal{O}(\log h(T))$ such that $D(\sigma_T(i), \delta_T(i)) = \sigma_T(i-1)$.

Finally, given a vertex $v = (x,y)$ of G, we define an array $a(v)$ of 8 bits indicating whether each of the edges between (x,y) and $(x \pm 1, y \pm 1)$ are present in G. Note that some of these 8 vertices may not even be present in G in which case the resulting bit is set to 0 since the edge is not present in G.

Now, in the labelling scheme for G, each vertex $v = (x,y) \in V(G)$ receives a label that is the concatenation of the following bitstrings:

(P1) $\alpha(y)$;
(P2) $\gamma(|\sigma_{T_y}(x)|)$, $\sigma_{T_y}(x)$;
(P3) $\delta_{T_y}(x)$;
(P4) if $y \neq h$ then $1, \delta_{T_{y+1}}(x)$;
if $y = h$ then 0;
(P5) if $y \neq h$ then $1, \nu_y(x)$;
if $y = h$ then 0; and
(P6) $a(v)$.

Two major components of this label are $\alpha(y)$, of length $\log n - \log|T_y| + \mathcal{O}(\log\log n)$, and $\sigma_{T_y}(x)$, of length $\log|T_y| + \mathcal{O}(k + k^{-1}\log n)$. Together they have length $\log n + \mathcal{O}(k + k^{-1}\log n + \log\log n)$. The lengths of the remaining components are as follows: $\gamma(|\sigma_{T_y}(x)|)$, $\delta_{T_y}(x)$, and $\delta_{T_{y+1}}(x)$ have length $\mathcal{O}(\log\log n + \log k)$, $\nu_y(x)$ has length $\mathcal{O}(k\log\log n + k\log k)$, and $a(v)$ has length $\mathcal{O}(1)$. Thus, in total the label has length $\log n + \mathcal{O}(k\log\log n + k\log k + k^{-1}\log n)$.

4.2 Adjacency Testing

First note that from a given label of $v = (x, y) \in V(G)$, we can decode each block of the label. This is because $\alpha(y)$ is prefix-free, $\gamma(|\sigma_{T_y}(x)|)$ is prefix-free so when we read it we know how long is $\sigma_{T_y}(x)$ and we can isolate it as well. The δ-codes are prefix-free again and $\nu_y(x)$ can be decoded as outlined in the proof of Lemma 27. Finally, the last 8-bits correspond to $a(v)$.

Given the labels of two vertices $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$ in G we can test if they are adjacent as follows.

Looking up the value of $A(\alpha(y_1), \alpha(y_2))$, we determine which of the following applies:

1. $|y_1 - y_2| \geqslant 2$: In this case we immediately conclude that v_1 and v_2 are not adjacent in G since they are not adjacent even in $P_1 \boxtimes P_2$.

2. $y_1 = y_2$: In this case, let $y := y_1 = y_2$. If the two bitstrings $\sigma_{T_y}(x_1)$, $\sigma_{T_y}(x_2)$ are the same, we conclude that $x_1 = x_2$ and $y_1 = y_2$, so $v_1 = v_2$ and we should output that they are not adjacent. Otherwise, we lexicographically compare $\sigma_{T_y}(x_1)$ and $\sigma_{T_y}(x_2)$. Without loss of generality, $\sigma_{T_y}(x_1)$ is smaller than $\sigma_{T_y}(x_2)$. Therefore, by Observation 4, $x_1 < x_2$. Recall that $x_2 \in L_y$ and $L_y^+ \subseteq V(T_y)$, so $x_2 - 1 \in V(T_y)$. We compute $D(\sigma_{T_y}(x_2), \delta_{T_y}(x_2)) = \sigma_{T_y}(x_2 - 1)$. If $\sigma_{T_y}(x_2 - 1) \neq \sigma_{T_y}(x_1)$, then we immediately conclude that $x_2 < x_1 - 1$, so v_1 and v_2 are not adjacent in G, since they are not adjacent even in $P_1 \boxtimes P_2$. Otherwise, we know that $v_1 = (x_2 - 1, y)$ and $v_2 = (x_2, y)$ are adjacent in $P_1 \boxtimes P_2$. Now we use the relevant bit of $a(v_1)$ (or $a(v_2)$) to determine if v_1 and v_2 are adjacent in G.

3. $y_1 = y_2 - 1$: In this case, we compute $B'(\sigma_{T_{y_1}}(x_1), \nu_{y_1}(x_1)) = \sigma_{T_{y_2}}(x_1)$. Let $y := y_2$. If the two bitstrings $\sigma_{T_y}(x_1)$, $\sigma_{T_y}(x_2)$ are the same, we conclude that $x_1 = x_2$. Thus $v_1 = (x_1, y - 1)$ and $v_2 = (x_1, y)$ are adjacent in $P_1 \boxtimes P_2$. Now we look up the relevant bit of $a(v_1)$ (or $a(v_2)$) to determine if v_1 and v_2 are adjacent in G. Otherwise, we lexicographically compare $\sigma_{T_y}(x_1)$ and $\sigma_{T_y}(x_2)$. If $\sigma_{T_y}(x_1)$ is smaller than $\sigma_{T_y}(x_2)$, then we conclude that $x_1 < x_2$. Recall that $x_2 \in L_y$ and $L_y^+ \subseteq V(T_y)$, so $x_2 - 1 \in V(T_y)$.

We compute $D(\sigma_{T_y}(x_2), \delta_{T_y}(x_2)) = \sigma_{T_y}(x_2 - 1)$. If $\sigma_{T_y}(x_2 - 1) \neq \sigma_{T_y}(x_1)$, then we immediately conclude that v_1 and v_2 are not adjacent in G, since they are not adjacent even in $P_1 \boxtimes P_2$. Otherwise, we know that $v_1 = (x_2 - 1, y - 1)$ and $v_2 = (x_2, y)$ are adjacent in $P_1 \boxtimes P_2$. Now we use the relevant bit of $a(v_1)$ (or $a(v_2)$) to determine if v_1 and v_2 are adjacent in G. If $\sigma_{T_y}(x_1)$ is larger than $\sigma_{T_y}(x_2)$, then we conclude that $x_1 > x_2$. Recall that $x_1 \in L_{y-1}$ and $L^+_{y-1} \subseteq V(T_y)$, so $x_1 - 1 \in V(T_y)$. We compute $D(\sigma_{T_y}(x_1), \delta_{T_y}(x_1)) = \sigma_{T_y}(x_1 - 1)$. If $\sigma_{T_y}(x_1 - 1) \neq \sigma_{T_y}(x_2)$, then we immediately conclude that v_1 and v_2 are not adjacent in G, since they are not adjacent even in $P_1 \boxtimes P_2$. Otherwise, we know that $v_1 = (x_1, y - 1)$ and $v_2 = (x_1 - 1, y)$ are adjacent in $P_1 \boxtimes P_2$. Now we use the relevant bit of $a(v_1)$ (or $a(v_2)$) to determine if v_1 and v_2 are adjacent in G.

4. $y_2 = y_1 - 1$: In this case, we compute $B'(\sigma_{T_{y_2}}(x_2), \nu_{y_2}(x_2)) = \sigma_{T_{y_1}}(x_2)$. Now we proceed as in the previous case.

This establishes our first result:

Theorem 28. *The family $\mathcal{G}$ of n-vertex subgraphs of a strong product $P \boxtimes P$ where P is a path has a $(1 + o(1)) \log n$-bit adjacency labelling scheme.*

Remark 29. The $o(\log n)$ term in the label length of Theorem 28 is $\mathcal{O}(k \log\log n + k \log k + k^{-1} \log n)$. An asymptotically optimal choice of k is therefore $k = \max\left\{7, \left\lceil \sqrt{\log n / \log\log n} \right\rceil\right\}$, yielding labels of length $\log n + \mathcal{O}\left(\sqrt{\log n \log\log n}\right)$.

5 Subgraphs of $H \boxtimes P$

In this section we describe adjacency labelling schemes for graphs G that are subgraphs of $H \boxtimes P$ where H is a graph of treewidth t and P is a path.

Let t be a positive integer. A graph H is a *t-tree* if there is an ordering $v_1, \ldots, v_m$ of $V(H)$ such that for every $i \in \{1, \ldots, m\}$, the neighbors of v_i earlier in the order, i.e., $N_H(v_i) \cap \{v_1, \ldots, v_{i-1}\}$ induce a clique of size at most t in H. (Let us emphasize that this is slightly more general than the usual definition of t-trees from the literature, which requires the neighbors of v_i earlier in the order to be a clique of size exactly $\min\{i - 1, t\}$; this broader definition will be more convenient for our purposes.) A vertex-ordering witnessing that H is a t-tree is called an *elimination ordering*. Note that if H is a t-tree with a given elimination ordering, then for any subset X of vertices of H, the subgraph $H[X]$ of H induced by X is a t-tree and the restriction of the elimination ordering of H to X is an elimination ordering of $H[X]$. Every graph of treewidth t is a spanning subgraph of a t-tree. For this reason, we may restrict ourselves to the case $H \boxtimes P$ where H is a t-tree, which we do.

Given a t-tree H, we fix an elimination ordering $v_1, \ldots, v_m$. For every $i \in \{1, \ldots, m\}$, the *family clique* $C_H(v_i)$ is defined as $N_H[v_i] \cap \{v_1, \ldots, v_i\}$. Note that $v_i \in C_H(v_i)$.

5.1 t-Trees and Interval Graphs

The *clique number* $\omega(G)$ of a graph G is the maximum size of a clique in G. The closed real interval with endpoints $a < b$ is denoted by $[a, b]$. For a finite set S of intervals, the *interval*

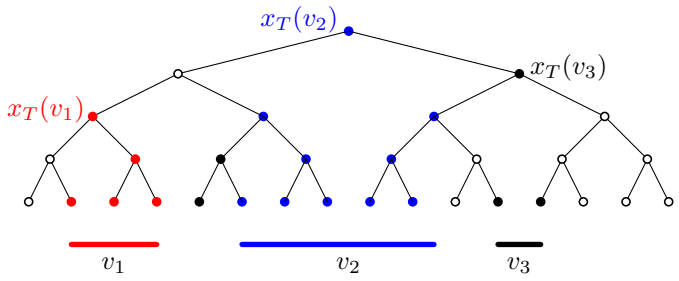

Figure 6: The definition of $x_T(v)$.

intersection graph G_S of S is the graph with vertex set $V(G_S) := S$ and in which there is an edge between two distinct intervals if and only if the intervals intersect.

The following well-known result states that every m-vertex t-tree is a subgraph of an interval graph of clique number $\mathcal{O}(t \log m)$.[8]

Lemma 30. *For every m-vertex t-tree H, there exists a mapping f assigning to every vertex v in H an interval $f(v)$ so that the following holds. Let $S := \{f(v) : v \in V(H)\}$. Then,*

1. *$f(v)$ intersects $f(w)$, so $f(v)f(w) \in E(G_S)$, for every edge $vw \in E(H)$, and*
2. *$\omega(G_S) \leqslant (t+1)\lfloor \log_3(2m+1) + 1 \rfloor$.*

Furthermore, for every proper $(t+1)$-colouring $\varphi' : V(H) \to \{1, \ldots, t+1\}$ of H, there exists a colouring $\varphi'' : V(H) \to \{1, \ldots, \lfloor \log_3(2m+1) + 1 \rfloor\}$ such that (φ', φ'') is a proper colouring of G_S.[9]

In light of Lemma 30, we call an *interval representation* of H a mapping f assigning to every vertex v of a graph H an interval $f(v)$ in such a way that $f(v)$ and $f(w)$ intersect for every edge $vw \in E(H)$. (Let us remark that $f(v)$ and $f(w)$ may or may not intersect when v, w are two non-adjacent vertices, thus H is a subgraph of the intersection graph of the intervals.) We will always assume that all the endpoints of intervals in the representation are distinct. This can be easily achieved by local perturbations not changing the intersection graph.

A finite set $X \subset \mathbb{R}$ *stabs* a set S of intervals if $X \cap [a, b] \neq \emptyset$ for every $[a, b] \in S$. Let H be a graph and f be an interval representation of H. We say that a binary search tree T *stabs* H if $V(T)$ stabs the set of intervals $\{f(v) : v \in V(H)\}$. For v a vertex in H, we let $x_T(v)$ denote the lowest common T-ancestor of $V(T) \cap f(v)$, see Figure 6. For $U \subseteq V(H)$, we let $x_T(U) := \{x_T(v) : v \in U\}$.

Lemma 31. *Let H be a graph with a fixed interval representation $v \mapsto [a_v, b_v]$. Let T be a binary search tree that stabs H. Then,*

1. *for every vertex v in H, we have $x_T(v) \in [a_v, b_v]$; and*

[8] The specific value $\log_3(2m+1)+1$ in Lemma 30 is obtained by applying a result of Scheffler [26] on the tree underlying the width-t tree decomposition of H.

[9] This property is only used when discussing small optimizations in label lengths at the end of the paper.

2. *for every clique C in H, the set of nodes $x_T(C)$ lie in a single root-to-leaf path in T.*

Proof. For the proof of the first item, consider $x := x_T(v)$. Either we have $x \in [a_v, b_v]$, in which case there is nothing to prove, or there are two nodes $x_1, x_2 \in V(T) \cap [a_v, b_v]$ such that x_1 is in the subtree of T rooted at the left child of x and x_2 is in the subtree of T rooted at the right child of x. By the binary search tree property, $x_1 < x < x_2$. But since $x_1, x_2 \in [a_v, b_v]$, we have $a_v \leqslant x_1 < x < x_2 \leqslant b_v$, so $x \in [a_v, b_v]$, as desired.

For the proof of the second item, we just show it for C being of size 2, so for a single edge. The statement for general cliques will follow immediately by induction. Thus, consider two adjacent vertices u_1 and u_2 in H and let $x_1 = x_T(u_1)$ and $x_2 = x_T(u_2)$. In order to get a contradiction, suppose that x_1 and x_2 do not lie on a single root-to-leaf path in T. Then there exists x in T such that x_1 is in the left subtree of x and x_2 is in the right subtree of x. In particular, $x_1 < x < x_2$ by the binary search tree property. Since $u_1u_2 \in E(H)$ the corresponsing intervals $[a_{u_1}, b_{u_1}]$, $[a_{u_2}, b_{u_2}]$ intersect, so their union is an interval as well. Since x_1, x_2 lie in the union and $x_1 < x < x_2$, the node x lies in the union as well. Therefore $x \in [a_{u_1}, b_{u_1}]$ or $x \in [a_{u_2}, b_{u_2}]$. This contradicts the choice of $x_T(u_1)$ or $x_T(u_2)$ and completes the proof of the second item. □

5.2 A Labelling Scheme for t-Trees

We describe a labelling scheme for t-trees that, like our labelling scheme for paths, is based on a binary search tree. The ideas behind this scheme are not new; this is essentially the labelling scheme for t-trees described by Gavoille and Labourel [18]. However, we present these ideas in a manner that makes it natural to generalize the results of Section 4.

We are given a t-tree H on m vertices with an interval representation $v \mapsto [a_v, b_v]$ as in Lemma 30. In particular, the clique number of the resulting interval graph is at most $(t+1)\lfloor \log_3(2m+1) + 1 \rfloor$. Since interval graphs are perfect, their clique number coincides with their chromatic number. Let $\varphi : V(H) \to [(t+1)\lfloor \log_3(2m+1) + 1 \rfloor]$ be a colouring such that u and v have distinct colours whenever the intervals of u and v intersect.

The following easy observation shows that a vertex v of H is uniquely identified by $\varphi(v)$ and the $x_T(v)$ value in a binary search tree T that stabs H. This gives ground for an adjacency labelling scheme.

Observation 32. *Let T be a binary search tree that stabs H. Let v and w be two distinct vertices in H. Then, $x_T(v) \neq x_T(w)$ or $\varphi(v) \neq \varphi(w)$. Consequently, $\sigma_T(x_T(v)) \neq \sigma_T(x_T(w))$ or $\varphi(v) \neq \varphi(w)$.*

Proof. If $x_T(v) = x = x_T(w)$, then by Lemma 31(1) intervals $[a_v, b_v]$ and $[a_w, b_w]$ each contain x, so they intersect. Therefore, $\varphi(v) \neq \varphi(w)$. □

Let T be a binary search tree that stabs H and let v be a vertex in H. Fix an elimination ordering of H. Recall that by Lemma 31(2), for every vertex v in H, all the nodes in $x_T(C_H(v))$ lie on a single root-to-leaf path in T. We define

$$\sigma_{H,T}(v) := \sigma_T(x), \quad \text{where } x \text{ is the node in } x_T(C_H(v)) \text{ of maximum depth in } T.$$

Note that $d_T(x_T(v)) \leqslant |\sigma_{H,T}(v)|$, and equality holds only if $x_T(v)$ is the deepest node in $x_T(C_H(v))$.

Now, we define the label $\tau_{H,T}(v)$ of a vertex v in H. Let $d = |C_H(v)|$ and let $u_1, \ldots, u_d$ be the vertices in $C_H(v)$, in any order, so v is one of them. Recall that γ is the Elias encoding of natural numbers. The label $\tau_{H,T}(v)$ is defined as the concatenation of

(T1) $\gamma(|\sigma_{H,T}(v)|)$ and $\sigma_{H,T}(v)$;
(T2) $\gamma(\varphi(v))$;
(T3) $\gamma(d)$;
(T4) $\gamma(d_T(x_T(u_i)))$ for each $i \in \{1, \ldots, d\}$;
(T5) $\gamma(\varphi(u_i))$ for each $i \in \{1, \ldots, d\}$.

Lemma 33. *There exists a function $F : (\{0,1\}^2) \to \{0, -1, 1, \perp\}$ such that for any t-tree H with a fixed elimination ordering and a fixed interval representation, and a proper colouring φ of the interval representation, and for any binary search tree T stabbing H, for any two vertices v, w in H, we have*

$$F(\tau_{H,T}(v), \tau_{H,T}(w)) = \begin{cases} 0 & \text{if } v = w; \\ -1 & \text{if } v \text{ and } w \text{ are adjacent in } H, \text{ and } w \in C_H(v); \\ 1 & \text{if } v \text{ and } w \text{ are adjacent in } H, \text{ and } v \in C_H(w); \\ \perp & \text{otherwise.} \end{cases}$$

Note that the labels $\tau_{H,T}(v)$ depend on the choice of elimination ordering and interval representation of H but the function F does not. Recall also that $d_T(x_T(u)) \leqslant |\sigma_{H,T}(u)| \leqslant h(T)$, for all vertices u in H. Moreover, $|C_H(u)| \leqslant t + 1$ for all vertices u in H. Thus when H is an m-vertex t-tree and φ takes values bounded in $\mathcal{O}(t \log m)$, we get labels $\tau_{H,T}(v)$ of length $h(T) + \mathcal{O}(t \cdot (\log h(T) + \log t + \log\log m))$. In particular, we can take a perfectly balanced tree T whose vertex set $V(T)$ is just the set of all endpoints of intervals representing H. Then T stabs H and $h(T) = \lfloor \log(2m) \rfloor$. This way the labels $\tau_{H,T}(v)$ are of length $\log m + \mathcal{O}(t \log\log m + t \log t)$.

Proof of Lemma 33. For the adjacency testing, first note that from a given label $\tau_{H,T}(v)$ of a vertex v in H, we can decode each block of the label. This is just because γ, the Elias encoding, is prefix-free.

Note that from the blocks of $\tau_{H,T}(v)$ we can determine $\sigma_T(x_T(v))$, $\varphi(v)$, and $\sigma_T(x_T(u))$, $\varphi(u)$, for all $u \in C_H(v)$.

Given the labels of two vertices v and w in H we can test if they are adjacent as follows.

1. If $\sigma_T(x_T(v)) = \sigma_T(x_T(w))$ and $\varphi(v) = \varphi(w)$, we conclude that $v = w$ (by Observation 32), so F outputs 0 in this case.

2. Let $d = |C_H(v)|$ and $u_1, \ldots, u_d$ be the vertices in $C_H(v)$. From the label of v, we decode the values of d, $\sigma_T(x_T(u_i))$ and $\varphi(u_i)$ for each $i \in \{1, \ldots, d\}$. If $\sigma_T(x_T(w)) = \sigma_T(x_T(u_i))$ and $\varphi(w) = \varphi(u_i)$ for some $i \in \{1, \ldots, d\}$, then we conclude that $w = u_i$ (by Observation 32) and F outputs -1.

3. Now, let $d = |C_H(w)|$ and let $u_1, \ldots, u_d$ be the vertices in $C_H(w)$. From the label of w, we decode the values of d, $\sigma_T(x_T(u_i))$ and $\varphi(u_i)$ for each $i \in \{1, \ldots, d\}$. If $\sigma_T(x_T(v)) = \sigma_T(x_T(u_i))$ and $\varphi(v) = \varphi(u_i)$ for some $i \in \{1, \ldots, d\}$, then we conclude that $v = u_i$ (by Observation 32) and F outputs 1.

4. Otherwise, $v \neq w$, $v \notin C_H(w)$, and $w \notin C_H(v)$. This implies that v and w are not adjacent in H because each edge in H connects a vertex u with a vertex in $C_H(u)$, for some u in H. Thus, in this case F outputs $\perp$.

□

In fact, we can get labels of length $\log m + \mathcal{O}(t \log\log m)$ instead of $\log m + \mathcal{O}(t \log\log m + t \log t)$. This can be done by improving the length of (T5) from $\mathcal{O}(t \log(t \log m))$ to $\mathcal{O}(t \log\log m)$, as we now explain. In order to achieve this we need to work with the colouring $(\varphi'(v), \varphi''(v))$ of H given by Lemma Lemma 30 instead of $\varphi(v)$. Recall that the image of φ' is $\{1, \ldots, t+1\}$ and the image of φ'' is $\{1, \ldots, \lfloor \log_3(2m+1) + 1 \rfloor\}$. Given a vertex v of H, we order the vertices $u_1, \ldots, u_d$ in $C_H(v)$ according to their φ'-colours. Now the improved (T5) block of the labelling is an array R of $t+1$ entries indexed by φ' colours. We set $R[\varphi'(w)] := \varphi''(w)$, for each $w \in C_H(v)$. Note that some entries may be undefined if $d < t+1$. This way only $\mathcal{O}(t \log\log m)$ bits suffice to encode R.

5.3 Interval Transition Labels

We now show that the solution presented in Section 4 generalizes to the current setting.

Let G be an n-vertex subgraph of $H \boxtimes P$ where H is an m-vertex t-tree and $P = 1, \ldots, h$ is a path. Clearly, we can assume that $m \leqslant n$ and $h \leqslant n$.

Fix an elimination ordering of H and an interval representation of H with clique number at most $(t+1)\lfloor \log_3(2m+1) + 1 \rfloor$, see Lemma 30. Let $\varphi : V(H) \to \{1, \ldots, (t+1)\lfloor \log_3(2m+1) + 1 \rfloor\}$ be a proper colouring of the interval representation of H.

For each $y \in \{1, \ldots, h\}$, let

$$S_y = \{v \in V(H) : (v, y) \in V(G)\}, \text{ and}$$
$$S_y^+ = \bigcup_{v \in S_y} C_H(v).$$

Note that $\sum_{y=1}^{h} |S_y| = n$ and $\sum_{y=1}^{h} |S_y^+| \leqslant (t+1)n$. Let $S_0 = \emptyset$. For each $y \in \{1, \ldots, h\}$, let $X_y \subset \mathbb{R}$ be the set of all endpoints of intervals representing vertices in $S_y^+ \cup S_{y-1}^+$. Apply Lemma 8 to the sequence $X_1, \ldots, X_h$ to obtain a 3-chunking sequence $V_1, \ldots, V_h$ such that $V_y \supseteq X_y$ for each $y \in \{1, \ldots, h\}$, and $\sum_{y=1}^{h} |V_y| \leqslant 2 \sum_{y=1}^{h} |X_y|$. Let $T_1, \ldots, T_h$ be a k-bulk tree sequence based on $V_1, \ldots, V_h$ with $k := \max\left\{7, \left\lceil \sqrt{\log n / \log\log n} \right\rceil\right\}$ (recall that if we fix the starting perfectly balanced binary search tree T_1 with vertex set V_1, this sequence exists and is unique).

For each $y \in \{1, \ldots, h\}$, let H_y^+ be the subgraph of H induced by $S_y^+ \cup S_{y-1}^+$. In particular, each H_y^+ is a t-tree. We fix the elimination ordering of H_y^+ inherited from H. Similarly, we fix the interval representation of H_y^+ and the proper colouring φ of the interval representation, as the projection of respective ones for H.

Since $X_y \subseteq V_y = V(T_y)$, we have that T_y stabs H_y^+.

By construction, we have

$$\sum_{y=1}^{h} |T_y| \leqslant 2\sum_{y=1}^{h} |X_y| \leqslant 4\sum_{y=1}^{h} |S_y^+ \cup S_{y-1}^+| \leqslant 8(t+1)n.$$

By Lemma 25, for each $y \in \{1,\ldots,h\}$ we have

$$\begin{aligned} h(T_y) &= \log|T_y| + \mathcal{O}(k + k^{-1}\log|T_y|) \\ &\leqslant \log|T_y| + \mathcal{O}(k + k^{-1}\log n). \end{aligned}$$

The following lemma, which is analogous to Lemma 27, is the last piece of the puzzle needed for an adjacency labelling scheme for subgraphs of $H \boxtimes P$.

Lemma 34. *There exists a function $J : (\{0,1\}^*)^2 \to \{0,1\}^*$ such that, for any H, P, G, φ, $S_1,\ldots,S_h$, $X_1,\ldots,X_h$, and each k-bulk tree sequence $T_1,\ldots,T_h$ defined as above, for each $y \in \{1,\ldots,h-1\}$ and each $v \in S_y$, there exists $\mu_y(v) \in \{0,1\}^*$ with $|\mu_y(v)| = \mathcal{O}(k \log h(T_y))$ such that $J(\sigma_{H_y^+,T_y}(v), \mu_y(v)) = \sigma_{H_{y+1}^+,T_{y+1}}(v)$.*

For strings a and b, let $a \preccurlyeq b$ denote that a is a prefix of b.

Proof. Let $v \in S_y$. First of all, note that v is a vertex in H_y^+ and H_{y+1}^+. Since T_y stabs H_y^+ and T_{y+1} stabs H_{y+1}^+, we conclude that $\sigma_{H_y^+,T_y}(v)$, $\sigma_{H_{y+1}^+,T_{y+1}}(v)$ are well-defined.

As in the proof of Lemma 26 and Lemma 27, we must dig into the details of the three bulk tree operations that transform T_y into T_{y+1}. Let $I := V(T_{y+1}) \setminus V(T_y)$ and $D := V(T_y) \setminus V(T_{y+1})$. Recall that the three steps are: applying BulkBalance(θ,k) to T_y with the appropriate value of θ to obtain T', applying BulkInsert(I) to T' to obtain T'', and applying BulkDelete(D) to T'' to obtain T_{y+1}. Recall that whenever BulkBalance(θ,k) is applied in this context we have $\theta < h(T_y)$.

By Lemma 13, Lemma 11, and Lemma 20 we have $h(T_{y+1}) \leqslant h(T'') \leqslant h(T') + 2 \leqslant h(T_y) + 3$. Thus the heights of all these trees are $\mathcal{O}(h(T_y))$.

The transition code $\mu_y(v)$ is the concatenation of two parts $\mu_y^{\text{Bal}}(v)$ and $\mu_y^{\text{Del}}(v)$ devoted to different steps of the transformation from T_y to T_{y+1}. First, the code $\mu_y^{\text{Bal}}(v)$ will serve to move from $\sigma_{H_y^+,T_y}(v)$ to $\sigma_{H_y^+,T'}$. Next, we will argue that $\sigma_{H_y^+,T'}(v) = \sigma_{H_{y+1}^+,T''}(v)$. Then the code $\mu_y^{\text{Del}}(v)$ will serve to move from $\sigma_{H_{y+1}^+,T''}(v)$ to $\sigma_{H_{y+1}^+,T_{y+1}}(v)$.

We start with a discussion on rebalancing that leads to the definition of $\mu_y^{\text{Bal}}(v)$. The tree T_y is rebalanced by an application of BulkBalance(θ,k) with an appropriate value of $\theta < h(T_y)$ and the resulting tree is T'. Recall that BulkBalance(θ,k) calls Balance(x,k) for each node x of depth-θ in T_y. Recall also that the changes made by Balance(x,k) are limited to the subtree of T_y rooted at x. See Figure 7. Let Q be the path in T_y encoded by $\sigma_{H_y^+,T_y}(v)$. Thus Q is the path from the root of T_y to the deepest node z in $x_{T_y}(C_{H_y^+}(v))$. Clearly, if Q is not hitting vertices of depth at least θ, then $\sigma_{H_y^+,T_y}(v) = \sigma_{H_y^+,T'}(v)$. Thus, in the case that $|\sigma_{H_y^+,T_y}(v)| < \theta$, we define

$$\mu_y^{\text{Bal}}(v) := \gamma(\theta).$$

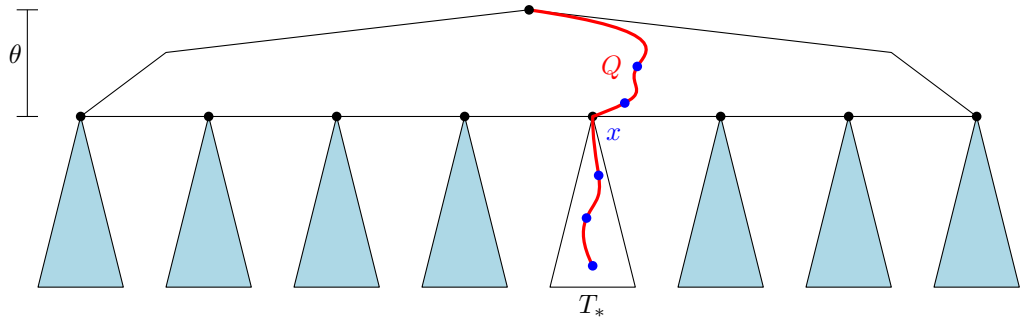

Figure 7: Only a call to Balance(x,k) on a node x lying on a path described by $\sigma_{H_y^+,T_y}(v)$ can affect $\sigma_{H_y^+,T'}(v)$.

Note that in this case $|\mu_y^{\text{Bal}}(v)| = \mathcal{O}(\log\theta) = \mathcal{O}(\log h(T_y))$.

Assume now that $|\sigma_{H_y^+,T_y}(v)| \geqslant \theta$ and let x be the T_y-ancestor of z at depth θ. Let T_* be the subtree of T_y rooted at x and let T'_* be the new tree obtained after calling Balance(x,k) on the root, x, of T_*. (So T_* is a subtree of T_y and T'_* is a subtree of T'.) Recall that the application of Balance(x,k) identifies two sets of nodes Z and X that eventually form a perfectly balanced tree $\hat{T}_0$ of height at most k which forms the top part of T'_*. Let Q' be the path in T' encoded by $\sigma_{H_y^+,T'}(v)$, so Q' is the path from the root of T' to the deepest node z' in $x_{T'}(C_{H_y^+}(v))$.

If $z' \in Z \cup X$, then $\sigma_{H_y^+,T'}(v) = \sigma_{T_y}(x), \sigma_{\hat{T}_0}(z')$. In this case, we define

$$\mu_y^{\text{Bal}}(v) := \gamma(\theta), 0, \gamma(|\sigma_{\hat{T}_0}(z')|), \sigma_{\hat{T}_0}(z').$$

Note that in this case $|\mu_y^{\text{Bal}}(v)| = \mathcal{O}(\log\theta) + \mathcal{O}(\log k) + \mathcal{O}(k) = \mathcal{O}(\log h(T_y) + k)$.

Now we are left with the case $|\sigma_{H_y^+,T_y}(v)| \geqslant \theta$ and $z' \notin Z \cup X$. Let $w \in C_{H_y^+}(v)$ be the vertex witnessing $z' = x_{T'}(w)$. By Lemma 31(1), z' is the unique node of Q' contained in $[a_w, b_w]$, i.e., the interval representing w. By the properties of a binary search tree, we conclude that

$$[a_w, b_w] \cap (Z \cup X) = \emptyset.$$

Consider now the node $x_{T_y}(w)$ in T_y. Since $[a_w, b_w] \cap Z = \emptyset$ we know that $x_{T_y}(w)$ lie in one of the trees, say T_i, of the forest $T_* - Z$. Recall that Balance(x,k) calls MultiSplit$(x_{i,1}, \ldots, x_{i,c_i})$ on T_i to obtain a sequence of trees $T_{i,0}, \ldots, T_{i,c_i}$. This, in turn results in zero or more calls to Split(x') for nodes $x' \in V(T_i) \cap X$. The following claim explains the effect of one individual call to Split(x'):

Claim 35. *Let T be a binary search tree that stabs an interval $[a_w, b_w]$, and let $T_{<x}$ and $T_{>x}$ be the two trees resulting from calling* Split(x) *on T for some $x \in V(T)$. Then, exactly one of the following is true:*

1. $a_w \leqslant x \leqslant b_w$;
2. $x < a_w$, *in which case* $x_{T_{>x}}(w) = x_T(w)$; *or*
3. $b_w < x$, *in which case* $x_{T_{<x}}(w) = x_T(w)$.

Proof. That exactly one of the three cases applies is obvious. Case (1) has no specific requirements and Cases (2) and (3) are symmetric, so we focus on Case (2), so $x < a_w \leqslant b_w$.

By Lemma 31(1), $z = x_T(v)$ is the unique node z in T such that the path P from the root to z has exactly one node in $[a_w, b_w]$. Recall that by construction the path P' from the root to z in $T_{>x}$ is obtained from P by deleting all values less than or equal to x. Therefore the path P' in $T_{>x}$ still has exactly one node in $[a_w, b_w]$, namely z, so $z = x_{T_{>x}}(w)$. This completes the proof of Claim 35. □

Since all the calls $\textsc{Split}(x')$ generated by $\textsc{MultiSplit}(x_{i,1}, \ldots, x_{i,c_i})$ on the subtree T_i are called with $x' \in X$ and $[a_w, b_w] \cap X = \emptyset$, Claim 35 guarantees that

$$x_{T_y}(w) = x_{T'}(w) = z', \text{ and}$$
$$\sigma_{H_y^+, T'}(v) = \sigma_{T'}(z').$$

Finally, by Lemma 26 there exists a function B and a bitstring $\nu(z')$ of length $\mathcal{O}(k \log h(T_y))$ such that $B(\sigma_{T_y}(z'), \nu(z')) = \sigma_{T'}(z')$.

All this justifies the following definition, in the case that $|\sigma_{H_y^+, T_y}(v)| \geqslant \theta$ and $z' \notin Z \cup X$:

$$\mu_y^{\textsc{Bal}}(v) := \gamma(\theta), 1, \gamma(|\sigma_{T_y}(x_{T_y}(w))|), \nu(x_{T_y}(w)).$$

Note that in this case $|\mu_y^{\textsc{Bal}}(v)| = \mathcal{O}(\log \theta) + \mathcal{O}(\log h(T_y)) + \mathcal{O}(k \log h(T_y)) = \mathcal{O}(k \log h(T_y))$.

Now we shall argue that

$$\sigma_{H_y^+, T'}(v) = \sigma_{H_y^+, T''}(v) = \sigma_{H_{y+1}^+, T''}(v).$$

Recall that T'' comes as a result of an application of $\textsc{BulkInsert}(I)$ to T' that attaches some small subtrees to the leaves of T'. This way, for every $u \in C_{H_y^+}(v) = C_H(v) \subseteq S_y^+ \subseteq V(H_y^+)$, we have that $x_{T'}(u)$ is a T''-ancestor of any node x in T'' such that $x \in I$ and $x \in [a_u, b_u]$. Hence, $x_{T'}(u) = x_{T''}(u)$ and $\sigma_{T'}(x_{T'}(u)) = \sigma_{T''}(x_{T''}(u))$. Therefore, $\sigma_{H_y^+, T'}(v) = \sigma_{H_y^+, T''}(v)$. Recall that $\sigma_{H_y^+, T''}(v)$ and $\sigma_{H_{y+1}^+, T''}(v)$ encode paths in T'' from the root to the deepest node in $x_{T''}(C_{H_y^+}(v))$ and in $x_{T''}(C_{H_{y+1}^+}(v))$, respectively. Again, since $v \in S_y$ we have that $C_H(v) \subseteq S_y^+ \subseteq V(H_y^+) \cap V(H_{y+1}^+)$. Therefore, $C_{H_y^+}(v) = C_H(v) = C_{H_{y+1}^+}(v)$ and $\sigma_{H_y^+, T''}(v) = \sigma_{H_{y+1}^+, T''}(v)$.

Next we describe $\mu_y^{\textsc{Del}}(v)$. The transition code $\mu_y^{\textsc{Del}}(v)$ serves to move from $\sigma_{H_{y+1}^+, T''}(v)$ to $\sigma_{H_{y+1}^+, T_{y+1}}(v)$. An application of $\textsc{BulkDelete}(D)$ to T'' results in a sequence of individual deletions. Consider a single deletion of an element x and let T^{bef} and T^{aft} denote the trees before and after the deletion, respectively.

Claim 36. *For every $u \in S_y^+$, we have*

$$\sigma_{T^{aft}}(x_{T^{aft}}(u)) \preccurlyeq \sigma_{T^{bef}}(x_{T^{bef}}(u)).$$

Proof of Claim 36. See Figure 8. At a global level, the deletion of a value x from T^{bef} involves finding a sequence of consecutive values $x_0 < x_1 < \cdots < x_r$ or $x_0 > x_1 > \cdots > x_r$ where $x = x_0$, x_r is a leaf and x_{i-1} is a T^{bef}-ancestor of x_i for each $i \in \{1, \ldots, r\}$. The leaf containing

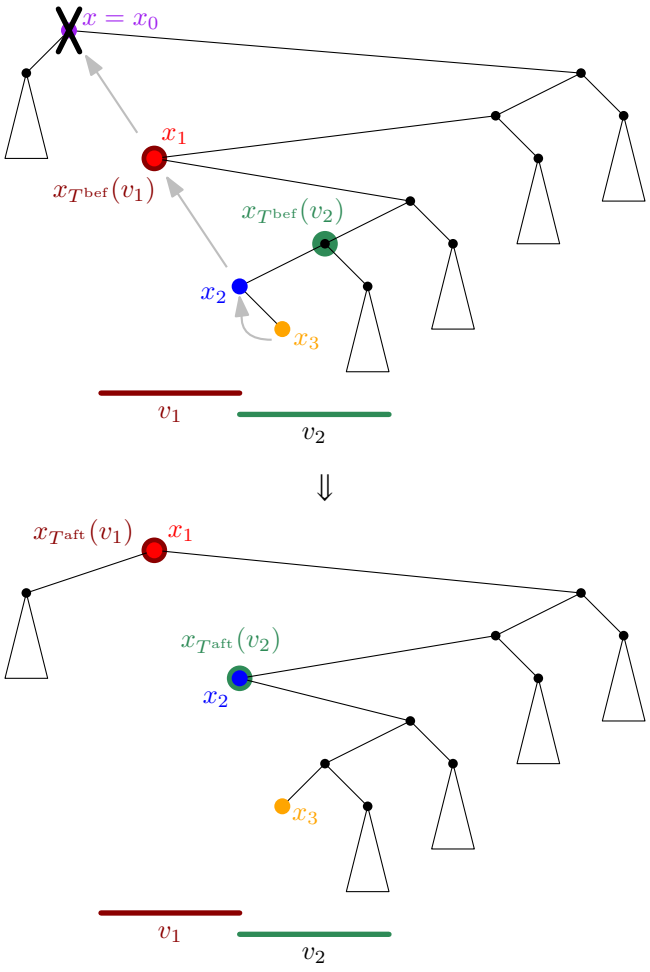

Figure 8: The effect of a single deletion on $x_{T^{\text{bef}}}(v)$.

x_r is deleted and, for each $i \in \{0,\dots,r-1\}$, the (value of) node x_i is replaced with (the value in) node x_{i+1}. The resulting tree is T^{aft}.

First we look at the case $x_{T^{\text{bef}}}(u) = x_0$. Recall that $u \in S_y^+$, so X_{y+1} contains both endpoints of the interval representing u, say $[a_u, b_u]$. This means that $x_0 \neq a_u$ and $x_0 \neq b_u$ and both endpoints lie in the subtree of T^{bef} rooted at x_0. In particular, x_0 is not a leaf and $r \geqslant 1$. By Lemma 31(1), we have $a_u < x_0 < b_u$. Since x_1 is the smallest value in the right subtree of x_0 or the largest value in the left subtree of x_0, we conclude that $a_u \leqslant x_1 \leqslant b_u$. Thus, in this case we have $x_{T^{\text{aft}}}(u) = x_1$ and

$$\sigma_{T^{\text{aft}}}(x_{T^{\text{aft}}}(u)) = \sigma_{T^{\text{aft}}}(x_1) = \sigma_{T^{\text{bef}}}(x_0) = \sigma_{T^{\text{bef}}}(x_{T^{\text{bef}}}(u)).$$

If $x_{T^{\text{bef}}}(u) = x_i$ for some $i \in \{1,\dots,r\}$, then $x_{T^{\text{aft}}}(u) = x_i$ (see the interval v_1 in Figure 8) and

$$\sigma_{T^{\text{aft}}}(x_{T^{\text{aft}}}(u)) = \sigma_{T^{\text{aft}}}(x_i) = \sigma_{T^{\text{bef}}}(x_{i-1}) \prec \sigma_{T^{\text{bef}}}(x_i) = \sigma_{T^{\text{bef}}}(x_{T^{\text{bef}}}(u)).$$

Finally, if $\sigma_{T^{\text{bef}}}(x_{T^{\text{bef}}}(u)) \neq \sigma_{T^{\text{aft}}}(x_{T^{\text{aft}}}(u))$ and $x_{T^{\text{bef}}}(u) \neq x_i$ for all $i \in \{0,\dots,r\}$, then the only possibility is that $x_{T^{\text{aft}}}(u) = x_i$ for some $x_i \in [a_u, b_u]$ (see the interval v_2 in Figure 8). This can only happen if $x_{T^{\text{bef}}}(u)$ is a T^{bef}-ancestor of x_i and x_i is a T^{aft}-ancestor of $x_{T^{\text{bef}}}(u)$. The latter is equivalent with the fact that x_{i-1} is a T^{bef}-ancestor of $x_{T^{\text{bef}}}(u)$. Therefore, we have

$$\sigma_{T^{\text{aft}}}(x_{T^{\text{aft}}}(u)) = \sigma_{T^{\text{aft}}}(x_i) = \sigma_{T^{\text{bef}}}(x_{i-1}) \prec \sigma_{T^{\text{bef}}}(x_{T^{\text{bef}}}(u)).$$

This completes the proof of the claim. □

Since BulkDelete(D) is a sequence of individual deletions, by multiple applications of Claim 36 we get that

$$\begin{aligned}
\sigma_{H^+_{y+1},T_{y+1}}(v) &= \sigma_{T_{y+1}}(x_{T_{y+1}}(u)) && \text{(for some } u \in C_{H^+_{y+1}}(v) = C_H(v)\text{)}\\
&\preccurlyeq \sigma_{T''}(x_{T''}(u)) && \text{(by Claim 36)}\\
&\preccurlyeq \sigma_{H^+_{y+1},T''}(v).
\end{aligned}$$

We define

$$\mu_y^{\text{Del}}(v) := \gamma(|\sigma_{H^+_{y+1},T_{y+1}}(v)|).$$

Note that $|\mu_y^{\text{Del}}(v)| = \mathcal{O}(\log h(T_{y+1})) = \mathcal{O}(\log h(T_y))$.

The function J is defined as expected: Given $\sigma_{H^+_y,T_y}(v)$ and $\mu_y(v)$, the function J first decodes the value of θ which is always the first block of $\mu_y^{\text{Bal}}(v)$. If $|\sigma_{H^+_y,T_y}(v)| < \theta$, then J concludes that $\sigma_{H^+_y,T'}(v) = \sigma_{H^+_y,T_y}(v)$. Otherwise, in the case $|\sigma_{H^+_y,T_y}(v)| \geqslant \theta$, the function J reads the next bit of $\mu_y(v)$. If it is 0 then J decodes the value of $\sigma_{\hat{T}_0}(z')$, computes $\sigma_{T_y}(x)$ which is the prefix of $\sigma_{H^+_y,T_y}(v)$ of length θ, and concludes that $\sigma_{H^+_y,T'}(v) = \sigma_{T_y}(x), \sigma_{\hat{T}_0}(z')$. Otherwise, the bit is 1. In this case, first J decodes $|\sigma_{T_y}(x_{T_y}(w))|$ and $\nu(x_{T_y}(w))$. Next, J computes $\sigma_{T_y}(x_{T_y}(w))$ which is the prefix of $\sigma_{H^+_y,T_y}(v)$ of length $|\sigma_{T_y}(x_{T_y}(w))|$. Next, J computes $B(\sigma_{T_y}(x_{T_y}(w)), \nu(x_{T_y}(w))) = \sigma_{T'}(x_{T'}(w))$ and in this case J concludes that $\sigma_{H^+_y,T'}(v) = \sigma_{T'}(x_{T'}(w))$.

Thus, in either case J establishes the value of $\sigma_{H_y^+,T'}(v) = \sigma_{H_{y+1}^+,T''}(v)$. Now, J looks up $\mu_y^{\text{DEL}}(v)$ and decodes $|\sigma_{H_{y+1}^+,T_{y+1}}(v)|$. The value of $\sigma_{H_{y+1}^+,T_{y+1}}(v)$ is simply the prefix of $\sigma_{H_{y+1}^+,T''}(v)$ of length $|\sigma_{H_{y+1}^+,T_{y+1}}(v)|$.

This completes the proof of the lemma. □

5.4 The Labels

We are ready to combine everything together and devise labels for vertices of G.

Let $A : (\{0,1\}^*)^2 \to \{0,1\}^*$ be the function given by Lemma 7 such that, using the weight function $w(y) := |T_y|$ for each $y \in \{1,\dots,h\}$, we have a prefix-free code $\alpha : \{1,\dots,h\} \to \{0,1\}^*$ such that

$$\begin{aligned}|\alpha(y)| &= \log\left(\textstyle\sum_{i=1}^{h} |T_i|\right) - \log|T_y| + \mathcal{O}(\log\log h)\\ &\leqslant \log\big(8(t+1)n\big) - \log|T_y| + \mathcal{O}(\log\log n)\\ &\leqslant \log n - \log|T_y| + \mathcal{O}(\log\log n + \log t),\end{aligned}$$

for each $y \in \{1,\dots,h\}$, and $A(\alpha(i),\alpha(j))$ outputs 0, 1, -1, or $\perp$, depending whether the value of j is i, $i+1$, $i-1$, or some other value, respectively.

Let $F : (\{0,1\}^*)^2 \to \{0,1\}^*$ be the function given by Lemma 33.

Let $J : (\{0,1\}^*)^2 \to \{0,1\}^*$ be the function given by Lemma 34 such that for each $y \in \{1,\dots,h-1\}$ and each $v \in S_y$, there exists a code $\mu_y(v)$ with $|\mu_y(v)| = \mathcal{O}(k \log h(T_y)) = \mathcal{O}(k \log\log n + k\log k)$ such that $J(\sigma_{H_y^+,T_y}(v), \mu_y(v)) = \sigma_{H_{y+1}^+,T_{y+1}}(v)$.

Let $z = (v,y)$ be a vertex in G. Recall that $S_y^+ \subseteq V(H_y^+)$ and if $i \neq h$ then also $S_y^+ \subseteq V(H_{y+1}^+)$. Therefore $C_H(v) = C_{H_y^+}(v) = C_{H_{y+1}^+}(v)$. Let $d = |C_H(v)|$ and let $u_1,\dots,u_d$ be the vertices of $C_H(v)$. Recall that $d \leqslant t+1$. We define $a(z)$ to be an array of $3d$ bits indicating whether each of the edges between (v,y) and $(u_i, y+\{-1,0,1\})$ are present in G. The label of $z = (v,y)$ is the concatenation of the following bitstrings:

(L1) $\alpha(y)$;
(L2) $\gamma(|\sigma_{H_y^+,T_y}(v)|)$ and $\sigma_{H_y^+,T_y}(v)$;
(L3) $\gamma(\varphi(v))$;
(L4) $\gamma(d)$;
(L5) $\gamma(\varphi(u_i))$ for each $i \in \{1,\dots,d\}$;
(L6) $\gamma(d_{T_y}(x_{T_y}(u_i)))$ for each $i \in \{1,\dots,d\}$;
(L7) if $y \neq h$ then 1, $\gamma(d_{T_{y+1}}(x_{T_{y+1}}(u_i)))$ for each $i \in \{1,\dots,d\}$;
if $y = h$ then 0;
(L8) if $y \neq h$ then 1, $\mu_y(v)$;
if $y = h$ then 0; and
(L9) $a(z)$.

The length of the components are as follows: (L1) is of length $\log n - \log|T_y| + \mathcal{O}(\log\log n + \log t)$, (L2) is of length $\log|T_y| + \mathcal{O}(k + k^{-1}\log n)$, (L3) is of length $\mathcal{O}(\log t + \log\log n)$, (L4) is of length $\mathcal{O}(\log t)$, (L5) is of length $\mathcal{O}(t \cdot (\log t + \log\log n))$, (L6) and (L7) are of lengths $\mathcal{O}(t \cdot (\log\log n + \log k))$, (L8) is of length $\mathcal{O}(k\log\log n + k\log k)$, and (L9) is of length $\mathcal{O}(t)$. In

total, the label length is $\log n + \mathcal{O}(k \log\log n + k^{-1} \log n + k \log k + t \log\log n + t \log k + t \log t)$. In particular, if t is a fixed constant, the label length is $\log n + \mathcal{O}(k \log\log n + k^{-1} \log n + k \log k)$.

We remark that, using the same trick as described at the end of Section 5.2, we can get the length of (L5) down to $\mathcal{O}(t \cdot \log\log n)$. Assuming $k = o(\log n)$, the total label length becomes then $\log n + \mathcal{O}((k+t)\log\log n + k^{-1} \log n + k \log k)$.

5.5 Adjacency Testing

First note that from a given label of $z = (v, y) \in V(G)$, we can decode each block of the label. Note also that once we decode $d = |C_H(v)|$, $\sigma_{T_y}(v)$, $\varphi(v)$, and $\varphi(u_i)$, $d_{T_y}(x_T(u_i))$ for all $i \in \{1, \ldots, d\}$, by Lemma 31(2) we can determine $\sigma_{T_y}(x_{T_y}(w))$ for each $w \in \{v, u_1, \ldots, u_d\}$.

Given the labels of two vertices $z_1 := (v_1, y_1)$ and $z_2 := (v_2, y_2)$ in G we test if the vertices are adjacent as follows. Looking up the value of $A(\alpha(y_1), \alpha(y_2))$, we determine which of the following cases applies:

1. $|y_1 - y_2| \geqslant 2$: In this case, we immediately conclude that z_1 and z_2 are not adjacent in G since they are not adjacent even in $H \boxtimes P$.

2. $y_1 = y_2$: In this case, let $y := y_1 = y_2$. Note that (L2), (L3), (L4), (L5), (L7) contain all the pieces of the labels $\tau_{H_y^+, T_y}(v_1)$ and $\tau_{H_y^+, T_y}(v_2)$ from Lemma 33. We compute $F(\tau_{H_y^+, T_y}(v_1), \tau_{H_y^+, T_y}(v_2))$ and determine if v_1 and v_2 are adjacent in H_y^+ (which is an induced subgraph of H). If v_1 and v_2 are not adjacent in H_y^+, then they are not adjacent in H and as a consequence, $z_1 = (v_1, y)$ and $z_2 = (v_2, y)$ are not adjacent in $H \boxtimes P$ and thus also not adjacent in G. If v_1 and v_2 are adjacent in H_y^+ (and thus also in H), then $z_1 = (v_1, y)$ and $z_2 = (v_2, y)$ are adjacent in $H \boxtimes P$. If $F(\tau_{H_y^+, T_y}(v_1), \tau_{H_y^+, T_y}(v_2)) = 1$, we identify the position of v_1 on the list of vertices in $C_H(v_2)$ (by its φ-colour) and then finally we look up the appropriate bit in $a(z_2)$ to see whether the corresponding edge in $H \boxtimes P$ is present in G or not. If $F(\tau_{H_y^+, T_y}(v_1), \tau_{H_y^+, T_y}(v_2)) = -1$, we identify the position of v_2 on the list of vertices in $C_H(v_1)$ and then finally we look up the appropriate bit in $a(z_1)$ to see whether the corresponding edge in $H \boxtimes P$ is present in G or not.

3. $y_1 = y_2 - 1$: In this case, let $y = y_1$. Since $v_1 \in S_y$, by Lemma 34, we can compute $J(\sigma_{H_y^+, T_y}(v_1), \mu_y(v_1)) = \sigma_{H_{y+1}^+, T_{y+1}}(v_1)$. Now, $\sigma_{H_{y+1}^+, T_{y+1}}(v_1)$ was the only missing piece of $\tau_{H_{y+1}^+, T_{y+1}}(v_1)$ and just from the label $z_2 = (v_2, y+1)$ we have $\tau_{H_{y+1}^+, T_{y+1}}(v_2)$. We compute $F(\tau_{H_{y+1}^+, T_{y+1}}(v_1), \tau_{H_{y+1}^+, T_{y+1}}(v_2))$ to test whether $v_1 = v_2$ or $v_1 v_2 \in E(H_{y+1}^+)$. If $v_1 \neq v_2$ and $v_1 v_2 \notin E(H_{y+1}^+)$, then z_1 and z_2 are not adjacent in $H \boxtimes P$ so they are not adjacent in G either. If $v_1 = v_2$ or $v_1 v_2 \in E(H_{y+1}^+)$ then we know that z_1 and z_2 are adjacent in $H \boxtimes P$. In this case, we can now consult the relevant bit of $a(z_1)$ or $a(z_2)$ to determine if z_1 and z_2 are adjacent in G.

4. $y_2 = y_1 - 1$: This case is symmetric to the preceding case, with the roles of z_1 and z_2 reversed.

This completes the proof of our main result.

Theorem 37. *For every fixed $t \in \mathbb{N}$, the family of all graphs G such that G is a subgraph of $H \boxtimes P$ for some t-tree H and some path P has a $(1+o(1))\log n$-bit adjacency labelling scheme.*

Theorem 1 and Theorem 2 are immediate consequences of Theorem 37, Theorem 9, and Theorem 10.

6 Conclusion

We conclude with a few remarks on the computational complexity of our labelling scheme. Given an n-vertex planar graph G, finding an 8-tree H (with mapping f as in Lemma 30 and colouring φ), a path P, and a mapping of G into a subgraph of $H \boxtimes P$ can be done in $\mathcal{O}(n \log n)$ time [23]. The process of computing the labels of $V(G)$ as described in Section 4 and Section 5 has a straightforward $\mathcal{O}(n \log n)$ time implementation. Thus, the adjacency labels described in Theorem 1 are computable in $\mathcal{O}(n \log n)$ time for n-vertex planar graphs.

In the discussion on the adjacency test function below, we focus on the case where t is a constant (which is the case in all our applications). The adjacency testing function can then be implemented in time $\mathcal{O}(k)$ in the standard w-bit word RAM model, providing for binary words of length $w = \Omega(\log n)$, bitwise logical operations, bitwise shift operations, and a most-significant-bit operation[10]. We note that Theorem 37 holds whenever k ranges from $\omega(1)$ to $\mathcal{O}(\sqrt{\log n / \log\log n})$. This yields a trade-off between adjacency test time and label length complexities. On one side, by choosing $k = \omega(1)$, we have labels of $(1+o(1))\log n$ bits and an adjacency test running in nearly constant time. On the other side, by selecting an adjacency test time complexity of $k = \mathcal{O}(\sqrt{\log n / \log\log n})$, the label length is minimized and has length $\log n + \mathcal{O}(\sqrt{\log n \log\log n})$.

The current result leaves two obvious directions for future work:

1. The precise length of the labels in Theorem 1 and Theorem 37 is, at best, $\log n + \mathcal{O}(\sqrt{\log n \log\log n})$. The only known lower bound is $\log n + \Omega(1)$. Closing the gap in the lower-order term remains an open problem.

2. Theorem 37 implies a $(1+o(1))\log n$-bit labelling schemes for any family of graphs that excludes an apex graph as a minor. Can this be extended to any K_t-minor free family of graphs?

Acknowledgement

Part of this research was conducted during the Eighth Workshop on Geometry and Graphs, held at the Bellairs Research Institute, January 31–February 7, 2020. We are grateful to the organizers and participants for providing a stimulating research environment. The authors are particularly grateful to Tamara Mchedlidze and David Wood for helpful discussions.

[10] The only purpose of the most-significant-bit operation is allow decoding of the Elias γ code in constant time.

References

[1] Mikkel Abrahamsen, Stephen Alstrup, Jacob Holm, Mathias Bæk Tejs Knudsen, and Morten Stöckel. Near-optimal induced universal graphs for bounded degree graphs. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 128:1–128:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.128.

[2] David Adjiashvili and Noy Rotbart. Labeling schemes for bounded degree graphs. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 375–386. Springer, 2014. doi:10.1007/978-3-662-43951-7_32.

[3] Noga Alon. Asymptotically optimal induced universal graphs. *Geometric and Functional Analysis*, 27(1):1–32, February 2017. doi:10.1007/s00039-017-0396-9.

[4] Stephen Alstrup, Søren Dahlgaard, and Mathias Bæk Tejs Knudsen. Optimal induced universal graphs and adjacency labeling for trees. *J. ACM*, 64(4):27:1–27:22, 2017. doi:10.1145/3088513.

[5] Stephen Alstrup, Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Ely Porat. Sublinear distance labeling. In Piotr Sankowski and Christos D. Zaroliagis, editors, *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 5:1–5:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.5.

[6] Stephen Alstrup, Cyril Gavoille, Esben Bistrup Halvorsen, and Holger Petersen. Simpler, faster and shorter labels for distances in graphs. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 338–350. SIAM, 2016. doi:10.1137/1.9781611974331.ch25.

[7] Stephen Alstrup, Inge Li Gørtz, Esben Bistrup Halvorsen, and Ely Porat. Distance labeling schemes for trees. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 132:1–132:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.132.

[8] Stephen Alstrup, Haim Kaplan, Mikkel Thorup, and Uri Zwick. Adjacency labeling schemes and induced-universal graphs. *SIAM J. Discrete Math.*, 33(1):116–137, 2019. doi:10.1137/16M1105967.

[9] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium*

on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA, pages 947–953. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545504.

[10] Marthe Bonamy, Cyril Gavoille, and Michał Pilipczuk. Shorter labeling schemes for planar graphs. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 446–462. SIAM, 2020. doi:10.1137/1.9781611975994.27.

[11] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986. doi:10.1007/BF01840440.

[12] Fan R. K. Chung. Universal graphs and induced-universal graphs. *Journal of Graph Theory*, 14(4):443–454, 1990. doi:10.1002/jgt.3190140408.

[13] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

[14] Vida Dujmović, Louis Esperet, Pat Morin, Bartosz Walczak, and David R. Wood. Clustered 3-colouring graphs of bounded degree. *CoRR*, abs/2002.11721, 2020. URL: http://arxiv.org/abs/2002.11721, arXiv:2002.11721.

[15] Vida Dujmović, Gwenaël Joret, Piotr Micek, Pat Morin, Torsten Ueckerdt, and David R. Wood. Planar graphs have bounded queue-number. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 862–875. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00056.

[16] Vida Dujmović, Pat Morin, and David R. Wood. The structure of k-planar graphs. *CoRR*, abs/1907.05168, 2019. URL: http://arxiv.org/abs/1907.05168, arXiv:1907.05168.

[17] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Information Theory*, 21(2):194–203, 1975. doi:10.1109/TIT.1975.1055349.

[18] Cyril Gavoille and Arnaud Labourel. Shorter implicit representation for planar graphs and bounded treewidth graphs. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, volume 4698 of *Lecture Notes in Computer Science*, pages 582–593. Springer, 2007. doi:10.1007/978-3-540-75520-3_52.

[19] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 334–343. ACM, 1988. doi:10.1145/62212.62244.

[20] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM J. Discrete Math.*, 5(4):596–603, 1992. doi:10.1137/0405049.

[21] Stephen G. Kobourov, Giuseppe Liotta, and Fabrizio Montecchiani. An annotated bibliography on 1-planarity. *Comput. Sci. Rev.*, 25:49–67, 2017. doi:10.1016/j.cosrev.2017.06.002.

[22] Pat Morin. *Open Data Structures*. Athabasca University Press, 2013. URL: https://opendatastructures.org/.

[23] Pat Morin. A fast algorithm for the product structure of planar graphs. *CoRR*, abs/2004.02530, 2020. URL: http://arxiv.org/abs/2004.02530, arXiv:2004.02530.

[24] John H. Muller. *Local Structure in Graph Classes*. PhD thesis, School of Information and Computer Science, March 1988.

[25] C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 36(1):445–450, 1961. doi:10.1112/jlms/s1-36.1.445.

[26] Petra Scheffler. Optimal embedding of a tree into an interval graph in linear time. In Jaroslav Nešetřil and Miroslav Fiedler, editors, *Fourth Czechoslovakian Symposium on Combinatorics, Graphs and Complexity*, volume 51 of *Annals of Discrete Mathematics*, pages 287–291. Elsevier, 1992. doi:10.1016/S0167-5060(08)70644-7.

[27] Jeremy P. Spinrad. *Efficient graph representations*, volume 19 of *Fields Institute monographs*. American Mathematical Society, 2003. URL: http://www.ams.org/bookstore-getitem/item=fim-19.

A Proof of Lemma 8

Proof of Lemma 8. $V_1, \ldots, V_h$ are constructed incrementally by a procedure BuildV$(S_1, \ldots, S_h)$ that makes use of a recursive subroutine Add(x, y). In the following code, V_0 and V_{h+1} act as sentinels whose only purpose to eliminate distracting boundary cases.

BuildV$(S_1, \ldots, S_h)$:

1: $V_0 \leftarrow V_{h+1} \leftarrow \mathbb{Z}$
2: $V_y \leftarrow \emptyset$ for each $y \in \{1, \ldots, h\}$
3: **for** $x = 1, \ldots, m$ **do**
4: **for** $y = 1, \ldots, h$ **do**
5: **if** $x \in S_y \setminus V_y$ **then**
6: Add(x, y)

Add(x, y):

1: **if** $y \in \{1, \ldots, h\}$ **then**
2: $V_y \leftarrow V_y \cup \{x\}$
3: **if** $|V_y| \geqslant 4$ **then**
4: let $x_{-1} > x_{-2} > \cdots > x_{-4}$ be the 4 largest elements in V_y (so $x_{-1} = x$)
5: **if** $\{x_{-1}, \ldots, x_{-4}\} \cap V_{y-1} = \emptyset$ or $\{x_{-1}, \ldots, x_{-4}\} \cap V_{y+1} = \emptyset$ **then**
6: Add$(x, y - 1)$
7: Add$(x, y + 1)$

It is easiest to think of the sets V_y as sequences, sorted in increasing order, so that Line 2 in Add(x, y) appends x to V_y.

That the procedure produces sets $V_1, \ldots, V_h$ such that $V_y \supseteq S_y$ for each $y \in \{1, \ldots, h\}$ is obvious. So the resulting sets $V_1, \ldots, V_h$ satisfy the first condition of the lemma.

To prove that $V_1, \ldots, V_h$ satisfy the second condition, we establish the loop invariant that, outside of Add(x, y), V_{y-1} and V_{y+1} each 3-chunk V_y for each $y \in \{1, \ldots, h\}$. Indeed, the only instant at which V_{y-1} fails to 3-chunk V_y is immediately after appending some value x to V_y in Line 2 of Add(x, y). If this occurs, it is immediately detected in Lines 3–5 and corrected in Line 6. Similarly, if V_{y+1} fails to 3-chunk V_y then this is immediately detected and corrected in Line 7.

Finally, we need to argue that $V_1, \ldots, V_h$ satisfy the third condition. For convenience, define $n := \sum_{y=1}^{h} |S_y|$ so that our task is to show that $\sum_{y=1}^{h} |V_y| \leqslant 2n$.

We do this with a *credit scheme* that maintains the following invariant during the execution of the algorithm: For each V_y, let c_y be the length the longest suffix $x_{-k}, \ldots, x_{-1}$ of V_y that does not intersect V_{y-1} or does not intersect V_{y+1}. Except during the execution of Add(x, y), $c_y \leqslant 3$, since V_{y-1} and V_{y+1} each 3-chunk V_y. We maintain the invariant that V_y stores c_y credits at all times. When we append to the list V_y in Line 2 of Add(x, y) we will pay with one credit that is spent and can never be used again.

To maintain our credit invariant, we will create 2 credits each time BuildV calls Add(x, y) in Line 6. Line 6 executes at most once for each of the n values in $S_1, \ldots, S_h$. Therefore Line 6 executes at most n times and at most $2n$ credits are created. Since each execution of Line 2 in Add(x, y) takes away one credit, this means that the total number of

times we append to lists in $V_1, \ldots, V_h$ is at most $2n$. Therefore, $\sum_{y=1}^{h} |V_y| \leqslant 2n$.

To manage these credits, we will pass two credits into each invocation of $\textsc{Add}(x, y)$, including the recursive invocations. For the invocations of $\textsc{Add}(x, y)$ in Line 6 of BuildV, the two credits passed in are the two newly-created credits.

When $\textsc{Add}(x, y)$ executes, one of the two credits passed to it is used to pay for the execution of Line 2, and this credit disappears forever, leaving one extra credit that we add to V_y since the newly-added value $x \in V_y$ may have increased c_y by 1. Thus far the credit invariant is maintained.

If no further recursive invocations of $\textsc{Add}(x, y)$ are made, then there is nothing further to do, so we consider the case where the two recursive invocations in Lines 6 and 7 are made. In this case, $c_y = 4$ before these recursive invocations are made. Afterwards, $c_y = 0$ since these invocations add x to V_{y-1} and V_{y+1}. This frees 4 credits. We pass 2 of these free credits into the recursive invocation of $\textsc{Add}(x, y-1)$ and the other 2 free credits into the recursive invocation of $\textsc{Add}(x, y+1)$. □