Simpler Adjacency Labeling for Planar Graphs with B-Trees

Paweł Gawrychowski [*] Wojciech Janczewski [†]

Abstract

An adjacency labeling scheme consists of an encoder and a decoder. The encoder assigns a binary string, called a label, to each vertex of a given graph G. Then, the decoder should decide, given only the labels assigned to two vertices u and v of the same graph G, whether (u, v) is an edge. While for planar graphs labels consisting of $\mathcal{O}(\log n)$ bits are not too hard to design, determining the exact constant factor in the upper bound remained a challenging open problem, with the only lower bound being $\log n$. Only very recently, Dujmović et al. [FOCS 2020] were able to bring the upper bound down to $\log n + \mathcal{O}(\sqrt{\log n \log\log n}) = \log n + o(\log n)$. At the heart of their construction lies a graph product structure theorem that is used to translate the problem into a data-structure question. The latter is then solved by designing a tailored balanced binary search trees that allow for efficient bulk operations. We show how this can be achieved with an arguably simpler approach based on B-Trees, while the other parts of the whole framework remain relatively unchanged. This allows us to obtain a cleaner upper bound of $\log n + \mathcal{O}(\sqrt{\log n})$ bits on the length of the labels, and additionally decrease the decoding time to constant.

1 Introduction

Labeling schemes. Informative labeling schemes serve the purpose of representing graphs in an extremely distributed manner. A scheme assigns each vertex v a binary string $\ell(v)$, called a label, so that later given just labels of two vertices and no additional information about the graph, we should be able to compute some function defined on those two vertices. In this context perhaps the most fundamental function is adjacency as considered by Kannan et al. [20], where we simply want to decide whether two given vertices are neighbors in the graph. Adjacency labeling schemes are especially interesting due to their direct connection to induced-universal graphs, a combinatorial object considered already several decades ago, for example by Moon [22].

Let $\mathcal{G}$ be a family of graphs. An adjacency labeling scheme for $\mathcal{G}$, as introduced formally by Peleg [24], consists of an encoder and a decoder. The encoder takes a graph $G \in \mathcal{G}$ and assigns a label (binary string) $\ell(u)$ to every vertex $u \in G$. The decoder receives labels $\ell(u)$ and $\ell(v)$, such that $u, v \in G$ for some $G \in \mathcal{G}$ and $u \neq v$, and should report whether u, v are adjacent in G. The decoder is not aware of G and only knows that u and v come from the same graph belonging to $\mathcal{G}$. Our scheme will always create unique labels, so in fact it does not need the assumption that $u \neq v$. We are interested in minimizing the maximum length of a label, that is, $\max_{G\in\mathcal{G}} \max_{u\in G} |\ell(u)|$, as a function of the number of vertices in a graph.

Related work. Construction of the optimal adjacency labeling scheme was considered for numerous classes of graphs, for example undirected, directed and bipartite graphs [5], graphs of bounded degree [13], trees [3] or planar graphs. It should be noted that for many important cases the optimal second-order terms of the length of the labels were obtained. There are multiple publications on designing schemes for other useful functions besides adjacency, like distance [4, 16, 19], ancestry in trees [15], routing [26, 14, 18], connectivity [21], reachability [7] or Steiner tree [24]. See [25] for a now slightly outdated survey.

Adjacency in planar graphs. There exists quite a big body of work on adjacency labeling schemes for planar graphs. Muller [23] noticed that since edges of any planar graphs can be oriented in a way that each vertex has an out-degree of at most 5, we obtain a simple $6 \log n$ labeling scheme by storing unique identifiers of vertices and all their out-neighbors (here we skip ceilings in logarithms). Kannan, Naor, and Rudich [20] described a $4 \log n$-bit scheme, based on the fact that planar graphs have arboricity of 3, thus we can store the ID of a vertex and IDs of its parents in three trees. The same approach, coupled with an optimal $\log n + \mathcal{O}(1)$-bit adjacency labeling for trees [3] gives us scheme for planar graphs with labels of length $3 \log n + \mathcal{O}(1)$. Furthermore, Gavoille and

[*]Institute of Computer Science, University of Wrocław, Poland. Partially supported by the Bekker programme of the Polish National Agency for Academic Exchange (PPN/BEK/2020/1/00444).

[†]Institute of Computer Science, University of Wrocław, Poland.

Labourel [17] used the fact that any planar graph can be edge-partitioned into two outerplanar graphs (which have a constant treewidth) to design a $(2+o(1))\log n$-bit labeling. Recently, Bonamy, Gavoille, and Pilipczuk [8] were able to leverage another structural theorem for planar graphs, the product structure theorem [11], to give a $(4/3+o(1))\log n$-bit labeling scheme.

Finally, very recently Dujmović, Esperet, Gavoille, Joret, Micek, and Morin [10] proved the following:

Theorem 1.1. (from [10]) *There is an adjacency labeling scheme for planar graphs on n vertices with labels of length* $\log n + \mathcal{O}(\sqrt{\log n \log\log n}) = \log n + o(\log n)$.

We remark that even though Dujmović et al. solved the problem of the first-order term of the length of the labels, obtaining optimal second-order terms is an important question considered for many types of labeling schemes and classes of graphs [5, 3, 15]. Here, the only known lower bound is trivial $\log n$, thus any improvements in the upper or lower bounds will still be very interesting.

The main result of [10] is achieved by employing a graph product structure theorem that allows translating the problem into a data-structure question. Then, the technical centerpiece of the construction from [10] is designing balanced binary search trees that handle the so-called bulk operations. Informally, those are batches of insertions and deletions for which we can save short transition labels describing how a root-to-node path changes.

Our contribution in this paper is threefold. While leaving most of the other parts of the previous framework untouched, we replace Bulk Trees with an alternative, arguably simpler approach based on B-Trees. It turns out that this allows for two other minor improvements in the final result. Firstly, we obtain a cleaner upper bound on the length of the labels, $\log n+\mathcal{O}(\sqrt{\log n})$. Secondly, we are able to implement the decoder in constant time, while previously its time complexity was small but superconstant if one wanted to retain the best possible second-order term of the length of the labels. The main theorem of this paper is the following:

Theorem 1.2. *There is an adjacency labeling scheme for planar graphs on n vertices with labels of length $\log n + \mathcal{O}(\sqrt{\log n})$ and the decoder working in constant time.*

Usually, encoding time is not interesting or relevant for the labeling schemes. Here, the encoder works in polynomial time.

Universal graphs. By the standard connection between adjacency labeling schemes and induced universal graphs (see for example [20]) we also have that:

Corollary 1.3. *For every n, there is an* universal *graph U_n with $n^{1+o(1)} = 2^{\log n + \mathcal{O}(\sqrt{\log n})}$ vertices that contains every planar graph on n vertices as an induced subgraph.*

2 Preliminaries

Organization of the labels. The final labels will consist of a constant number of concatenated parts. We can store in each label a constant number of pointers to the beginning of each of those parts, which allows us to separate them and gives constant time access to any part. As the total length of a label will be $\mathcal{O}(\log n)$, the pointers add only $\mathcal{O}(\log\log n)$ bits. For convenience, we assume that the value of $\lceil \log n \rceil$ is known to the decoder, where n is the size of the graph. Including that information in every label also takes only $\mathcal{O}(\log\log n)$ bits.

Weighted prefix-free codes. We will need some weighted prefix-free codes enhanced with successor detection:

Lemma 2.1. *Given positive integers $w_1, w_2, \ldots, w_m$, $\sum_{i=1}^m w_i = n$, we can construct prefix-free codes c for all w_i such that the size of $c(w_i)$ is at most $\log(n/w_i)+3$, and $c(w_i) < c(w_j)$ iff $i<j$. Additionally, we can assign for all w_i labels $\mathsf{succlabel}(w_i), \mathsf{predlabel}(w_i)$ of size $\mathcal{O}(\log\log n)$ allowing us to check for a successor relationship, that is there is a function f such that $f(c(w_i), c(w_j), \mathsf{succlabel}(w_i), \mathsf{predlabel}(w_j)) = 1$ iff $j=i+1$.*

Proof. (Sketch) Construct a full binary tree on at least n leaves and less than $2n$ leaves. Number the leaves from 0 and divide them into groups of exactly w_i consecutive leaves (with some last ones possibly not assigned to any group). For any power of two 2^r we say that its proper subtree is a minimal connected subtree on leaves $[k2^r, (k+1)2^r)$, for any k. For any i and the maximum r such that $2^r \le \max(1, w_i/2)$, 2^r must have a proper subtree on any subset of w_i consecutive leaves. To represent w_i, we choose (arbitrary) such proper subtree for the i-th group of consecutive leaves, and encoding of the path from the root of the tree to the root of this proper

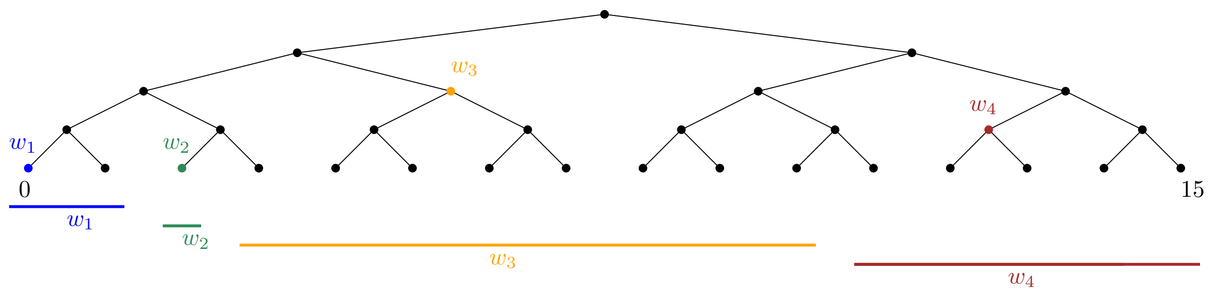

Figure 1: Assignment of weighted prefix-free codes for $w_1 = 2, w_2 = 1, w_3 = 8, w_4 = 5$. We have $c(w_2) = 0010$, $c(w_3) = 01$, $\mathsf{succlabel}(w_2) = \mathsf{predlabel}(w_3) = 1$.

subtree is $c(w_i)$. The length of $c(w_i)$ is at most $\log n + 1 - \log(w_i/4) \le \log(n/w_i) + 3$. All chosen subtrees are disjoint and $c(w_i)$ are lexicographically sorted.

Denote by v_i the root of the proper subtree used to define code $c(w_i)$. As $\mathsf{succlabel}(w_i)$ we store the depth (the distance to the root) of the lowest common ancestor, LCA, of v_i and v_{i+1}. Similarly, $\mathsf{predlabel}(w_i)$ is the depth of LCA of v_i and v_{i-1}. This takes up to $\mathcal{O}(\log\log n)$ bits. Now observe that for any $i < j$, if u is LCA of v_i and v_{i+1}, then v_j lies either in the right subtree of u or $c(w_i)$ and $c(w_j)$ differ on the first $\mathsf{succlabel}(w_i)$ bits. This means that for any $i < j$ we have $i = j + 1$ iff $\mathsf{succlabel}(w_i) = \mathsf{predlabel}(w_j)$ and prefixes of $c(w_i)$ and $c(w_j)$ of length $\mathsf{succlabel}(w_i)$ are equal.

See Figure 1 for an example. $\square$

Fractional cascading. We will need a specific variant of the fractional cascading technique [9]. Given a sequence of sets of integers, we will increase the sizes of those sets, possibly significantly, to ensure that all sets are very similar to their neighbors in the sequence.

LEMMA 2.2. *Given any ordered sets of integers $S_1, S_2, \ldots, S_m$, with $\sum_{i=1}^m |S_i| = n$, and an integer parameter k, we can find sets $V_1, V_2, \ldots, V_m$ such that:*

1. $\sum_{i=1}^m |V_i| = \mathcal{O}(k^2 n)$,
2. $\forall_{i=1}^m |V_i| \le n$,
3. $\forall_{i=1}^m S_i \subseteq V_i$,
4. *for any $t \le k$ and $j < m$, for any t consecutive elements from V_j, at least $t-1$ of them belong to V_{j+1},*
5. *for any $t \le k$ and $j > 1$, for any t consecutive elements from V_j, at least $t-1$ of them belong to V_{j-1}.*

We refer to sets $V_1, V_2, \ldots, V_m$ as augmented *sets.*

Proof. The proof is similar to the one used in [10]. We will use intermediate sets U_i. At the beginning assign $U_1 = S_1$. Next, iterating with i from 1 to $m-1$ we copy all elements of U_i besides every k-th one to S_{i+1}. That is, for $1 \le i < m$ in increasing order, say ordered elements of U_i are $U_i = \{u_0, u_1, \ldots\}$, then assign $U_{i+1} = S_{i+1} \cup \{u_j \in U_i : j \mod k \ne 0\}$. Obtained sets clearly satisfy the third and fourth requirements, but we want to check that their total size is $\mathcal{O}(kn)$. It holds that $|U_{i+1}| \le |S_{i+1}| + \lfloor (1-1/k)|U_i| \rfloor$, and so by induction we have $|U_{i+1}| \le \sum_{j=1}^{i+1} (1-1/k)^{i+1-j} |S_j|$. Therefore:

$$\sum_{i=1}^m |U_i| \le \sum_{i=1}^m \sum_{j=1}^{i+1} (1-1/k)^{i+1-j} |S_j| = \sum_{i=1}^m (|S_i| \sum_{j=i}^m (1-1/k)^{j-i}) < k \sum_{i=1}^m |S_i| = kn.$$

As for the last property, we observe that once (4) is satisfied, copying elements from U_{i+1} to U_i cannot violate it. Thus we can repeat the process of copying elements, this time from right to left, creating the final augmented sets V_i. This again increases the total size of sets at most by a factor of k. The second property holds because we use only numbers present in the initial sets S_i. $\square$

We note that it is possible to obtain sets $V_1, \dots, V_m$ with the same properties (2)-(5) and total size of only $\mathcal{O}(kn)$ by bounding the cascading slightly more carefully, but this is not needed in this work.

3 Framework

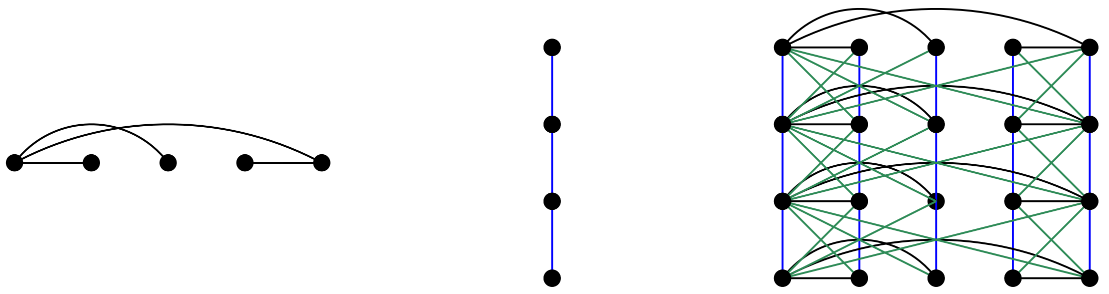

Figure 2: A strong product of a tree and a path.

In this section, we give an overview of the algorithm. Note that this is a brief description to provide some intuitions and informally introduce the necessary tools; the full proofs are presented in Section 5. Throughout the paper, we consider the graphs to have *vertices* and B-trees or BSTs to have *nodes*. One important notion is the strong product of graphs. See Figure 2 for a small example.

Definition 3.1. *The strong product $G \boxtimes H$ of two graphs $G = (V_G, E_G), H = (V_H, E_H)$ is the graph $P = (V_P, E_P)$ with $V_P = V_G \times V_H$, and in E_P there is an edge $((x_1, y_1), (x_2, y_2))$ if and only if one of the following holds:*

1. *$x_1 = x_2$ and $(y_1, y_2) \in E_H$.*
2. *$y_1 = y_2$ and $(x_1, x_2) \in E_G$.*
3. *$(x_1, x_2) \in E_G$ and $(y_1, y_2) \in E_H$.*

To design an adjacency labeling scheme, we need the product structure theorem for planar graphs [11] by Dujmović, Joret, Micek, Morin, Ueckerdt, and Wood:

Theorem 3.2. *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.*

Note that for G with n vertices we can make sure that both H and P also have at most n vertices (by getting rid of vertices of H and P not present in any vertex of the strong product used for the mapping into a subgraph). Very recently the above theorem was further refined by replacing 'treewidth at most 8' with 'simple treewidth at most 6' [27]. For the adjacency labeling scheme, this stronger formulation affects only constants in the second-order term.

Theorem 3.2 is the crucial tool used in [10] and in this paper. Say $P = (p_1, \dots, p_m)$, then for fixed i we refer to all vertices (h_j, p_i), where $h_j \in V(H)$, as i-th row of $H \boxtimes P$. As might be seen in Figure 2, by definition we have three types of edges: vertical edges coming from the path (blue), horizontal edges occurring inside one row (black), and diagonal edges between two consecutive rows (green). For a single row (black edges), it is known how to assign small adjacency labels $\mathsf{neigh}(v)$ using balanced search trees and properties of graphs of bounded treewidth. That is, any H with treewidth of t can be extended to a graph H' having two important properties:

1. By a perfect elimination ordering we can have an orientation of the edges of H' such that for any $v \in V(H')$ its closed out-neighborhood is a clique of size at most $t + 1$ in H'.
2. By using an interval representation of H' together with the notion of path-decomposition, we can construct a BST T and assign vertices of H' to nodes of T in such a way *all* vertices from any clique of H' are assigned to vertices on a single root-to-leaf path in T.

Then, instead of storing codes for all nodes from the clique of neighbors, we can store an encoding of this single path in the tree and at most $t+1$ small pointers to the prefixes of this path.

Therefore the main problem is representing edges between two consecutive rows (green edges). The authors of [10] deal with this issue by introducing transition labels $\delta(v)$, allowing to switch between encodings of vertices in the same column of two consecutive rows.

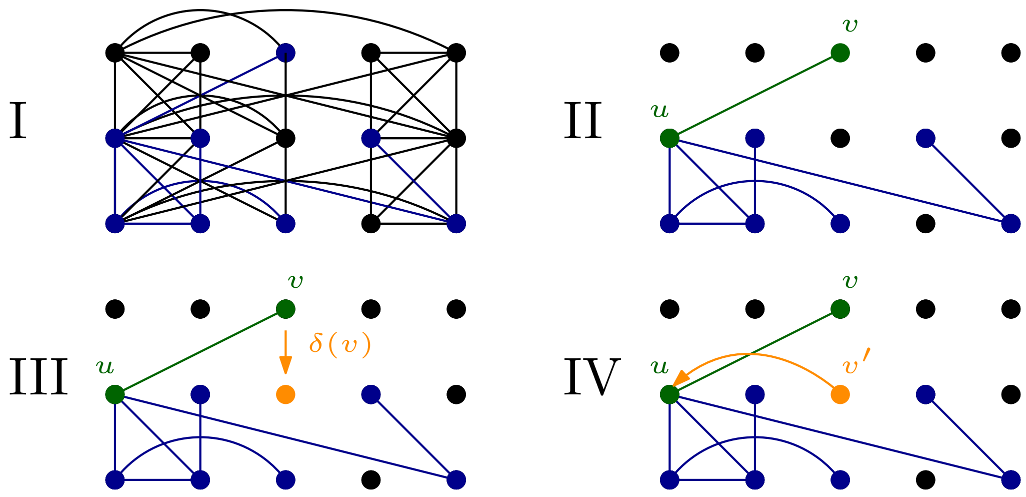

Figure 3: I. We consider the blue subgraph of a strong product of a path and a graph of bounded treewidth.
II. Say that we want to check for the existence of the green edge between v and u. First, we check $\mathsf{row}(v)$ and $\mathsf{row}(u)$ and see that a row of vertex u is the next one after a row of v, so indeed there might be an edge (v, u).
III. Then we use encoding of v in a B-tree for $\mathsf{row}(v)$ and a transition label $\delta(v)$ to get the encoding of the vertex v' in a B-tree for $\mathsf{row}(u)$.
IV. Now, as u and v' lies in one row, they can be seen as vertices of a graph of constant treewidth. Both of those vertices store just constant number of edges in $\mathsf{neigh}(u)$ and $\mathsf{neigh}(v')$, and one of those edges is (u, v'). Then we only need to confirm in a respective $\mathsf{bitarray}$ whether this edge of a strong product belongs to the blue subgraph.

Now let us give a very brief overview of this approach, see also Figure 3 and its description. We can use weighted prefix-free codes from Lemma 2.1 to assign to rows codes $\mathsf{row}(v)$, with lengths depending on the number of vertices of G present in that row (the more vertices, the shorter codes), in a way that we can also check for rows adjacency. Then for any i, vertices from row i are represented as leaves in a B-tree T_i. Adjacency sub-labels $\mathsf{neigh}(v, T_i)$ used for a single row i are assigned by using T_i and exploiting the constant treewidth of H. Moreover, we augment sets of vertices in rows using Lemma 2.2 to ensure that B-trees in the sequence $T_1, T_2, \ldots$ do not change too much, and also if there is a vertex $(v, p_i) \in V(G)$, we store (v, p_{i+1}) in T_{i+1} even if $(v, p_{i+1}) \notin V(G)$. Then, any vertex $(v, p_i) \in V(G)$ is assigned a small transition label $\delta(v, i)$ that together with other parts of $\ell((v, p_i))$ allow us to compute encoding of (v, p_{i+1}) in T_{i+1} and its $\mathsf{neigh}(v, T_i)$, which will be enough to check for any edges between (v, p_i) and vertices in a row $i+1$. As G is actually a subgraph of the considered graph, we additionally need a constant number of bits stored as a $\mathsf{bitarray}((v, p_i))$ to check which of those edges of $H \boxtimes P$ are in fact edges of G.

4 Maintaining B-trees

In this section, we describe how to employ B-trees to encode sets of vertices in rows of $H \boxtimes P$ and create necessary transition labels. This is the main contribution of this work. Replacing Bulk Trees from [10] with B-trees as described here allows us to simplify the adjacency labeling scheme.

We use weight-balanced B-trees. Similarly to the use of trees for example in [6], there is no direct constraint on the number of children of a node, but instead we have a constraint on the size of a subtree of a node on a given level. All leaves of a B-tree are on the same level 0 and store a single integer, for all other nodes their level is the distance from the leaves. The weight of a node is the number of leaves in its subtree. B-tree is parametrized with

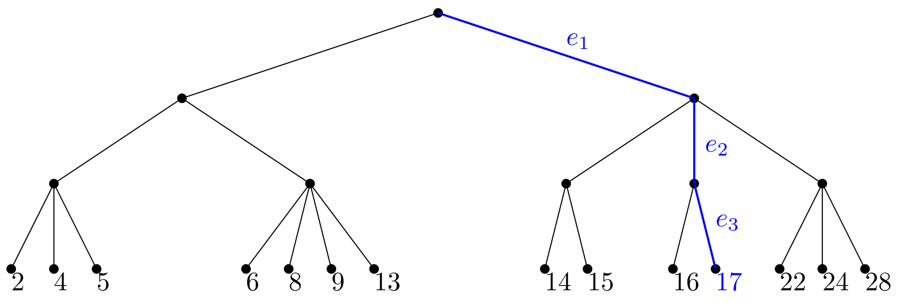

Figure 4: Example of B-tree of height 3. We use it to encode paths to the leaves, not as a dictionary, so internal nodes store no keys. The path to leaf with value 17 is stored simply as a concatenation of prefix-free codes of edges e_1, e_2 and e_3.

an integer $a \geq 4$, used for balancing. We will assign $a = 2^{\sqrt{\log n}}$, which is at least 4 for big enough n. Assume we are given augmented sets $V_1, \ldots, V_m$ as in Lemma 2.2, with parameter k to be specified later. As we will create a sequence of B-trees $T_1, \ldots, T_m$ corresponding to sets V_i, the properties of augmented sets will guarantee that the structure of trees neighboring in the sequence cannot change rapidly. We use the following notion of balance:

Definition 4.1. *A non-root node on level h is called* balanced *if it has weight in $[a^h, 3a^h]$ and* semi-balanced *if it has weight in $[0.5a^h, 6a^h]$. The root is balanced if it has weight of at most $3a^h$ and semi-balanced if it has weight of at most $6a^h$. A B-tree is semi-balanced if all its nodes are semi-balanced. Note that leaves are always balanced.*

Observe that given any set of n integers, we can initialize a balanced B-tree containing elements of the given set in leaves. First we fix the height of a tree by taking the minimal L such that $n \leq 3a^L$. Then we distribute values using a top-down approach, starting with the root and all n values assigned to the whole tree. Assume we are at some node v on level $1 < h \leq L$, and need to create balanced children of v on level $h-1$. We can simply repeatedly create a new child with exactly a^{h-1} values assigned to its subtree until we are left with no more than $2a^{h-1}$ values, then create the last child with the remaining values.

We can assume $n > 1$ and make sure that the root always has at least two children, as otherwise we make its only child the new root. Thus we get the following:

Claim 4.2. *A semi-balanced B-tree with n leaves has the height of at most $\log_a n + 1$.*

Additionally, for a semi-balanced tree it is easy to encode paths. There are at most $\log_a n + 1$ edges on a path from the root, and each node has at most $12a$ children, thus we can encode a single edge on $\log a + \mathcal{O}(1)$ bits. This way we get:

Claim 4.3. *In a semi-balanced B-tree T with n leaves a path from the root to any node can be uniquely encoded on $\log a \cdot \log_a n + \mathcal{O}(\log a + \log_a n) = \log n + \mathcal{O}(\log a + \log_a n)$ bits, by concatenating prefix-free codes of the edges.*

We will denote this encoding for a node v as $\mathsf{path}(v, T)$. In the case of leaves we abuse notation, denoting by $\mathsf{path}(d, T)$ encoding of the path from the root to the leaf v storing value d.

Each augmented set representing one row of a strong product graph is considered in a single phase of the algorithm. T_1 can be initialized as balanced with values from V_1. When we move to the next phase, say when moving from T_i representing V_i to T_{i+1} representing V_{i+1}, we need to insert values from $V_{i+1} \backslash V_i$ and delete values from $V_i \backslash V_{i+1}$. The problem is we cannot completely rebalance B-tree after that, as we want to retain root-to-leaves paths relatively unchanged. Thus, after moving to the next set, we rebalance only a single level in a tree, ensuring with a reshuffling of subtrees according to the weights that all nodes on that level are balanced. This is done cyclically, when moving to the next set we rebalance nodes on the level one higher than previously. During this process, some nodes might stop being balanced, but due to the properties of augmented sets, they will remain semi-balanced until the next rebalancing of their level, as explained below. See Algorithm 1 for a pseudocode.

Handling insertions/deletions. Deletion of leaves (without rebalancing) is straightforward. For insertions of leaves with new values, we need to know where to place them, that is which node on the first level will be their

Algorithm 1 A sequence of B-trees with a cyclic rebalancing.

```
function COMPUTE-LABELS($V_1, \dots, V_m$, $a$)
    Input: augmented sets $V_1, \dots, V_m$, integer balance parameter $a$.
    Output: encodings $\mathsf{path}(v, T_i)$ and $\delta(v, i)$ for every $v \in V_i$.

    Initialize balanced B-tree $T_1$ with $V_1$
    Output $\mathsf{path}(v, T_1)$ for all $v \in V_1$
    $h \leftarrow 1$
    for $i = 2 \dots m$ do
        Delete leaves with values from $V_{i-1} \backslash V_i$
        Insert leaves with values from $V_i \backslash V_{i-1}$
        Rebalance level $h$ of $T_i$
        $h \leftarrow h + 1$
        if $h$ is bigger than the height of $T_i$ then
            $h \leftarrow 1$
        Output $\mathsf{path}(v, T_i)$ for all $v \in V_i$
        Output $\delta(v, i - 1)$ for all $v \in V_{i-1} \cap V_i$
```

parent. To have full control over this process, during the first phase after constructing B-tree and then after each rebalancing of the first level, we assign intervals to nodes on the first level. Let us number nodes on the first level from left to right. Then interval assigned to node j starts at the value of its leftmost leaf and ends at the value of the leftmost leaf of node $j + 1$ decreased by 1. Additionally, the interval of node 1 is open to the left, that is it begins at the minus infinity, and interval of the last node on the first level is open to the right. Those intervals cover all integers and are kept for one whole cycle of rebalancings, as no new nodes on the first level are created and none are deleted until rebalancing happens again for the first level. Any new leaf is inserted below the node with an interval containing its value.

Maintaining semi-balance. We can use Lemma 2.2 to bound changes of weights of nodes. Here we ignore the case of a B-tree of height 1, as it is always balanced. Denote the maximum height of a B-tree by $L = \log_a n + 1$. Assume that sets $V_1, V_2, \dots, V_m$ are obtained by applying Lemma 2.2 with a parameter $k = 2L/\ln 1.5 + 1$.

Let $V_{i,s,t} = \{v \in V_i : s \leq v \leq t\}$. By property (5) from the definition of augmented sets we have that the set $V_{i+1,s,t}$ cannot be much larger than $V_{i,s,t}$. More precisely, if $|V_{i+1,s,t}| \leq k$, then we have that $|V_{i+1,s,t}| \leq |V_{i,s,t}| + 1$. Now assume $|V_{i+1,s,t}| > k$, then we can partition $V_{i+1,s,t}$ into groups of consecutive k elements, with the last group possibly having less elements. Then for each group, all but one element must be contained in $V_{i,s,t}$. For full groups, this is at least $k - 1$ from k elements, resulting in an increase by a factor of at most $1 + 1/(k - 1)$. The last group might have less than k elements (possibly even just one element), so we have an additional additive increase by 1. As there is at least one full group, we have $|V_{i,s,t}| \geq k - 1$. It follows that $|V_{i+1,s,t}| \leq 1 + |V_{i,s,t}|(1 + 1/(k - 1)) \leq |V_{i,s,t}|(1 + 2/(k - 1))$, where the worst case is for $|V_{i,s,t}| = k - 1, |V_{i+1,s,t}| = k + 1$. Thus the following holds:

$$|V_{i+1,s,t}| \leq \max(|V_{i,s,t}| + 1,\ |V_{i,s,t}|(1 + 2/(k - 1))),$$

which means that

$$|V_{i+L,s,t}| \leq (|V_{i,s,t}| + L)(1 + 2/(k - 1))^L.$$

Assume for simplicity that $a \geq 4L$, that is $2^{\sqrt{\log n}} \geq 4 + 4\log_{2^{\sqrt{\log n}}} n$, which holds for big enough n. Recall that a balanced node on the first level has the weight of at least a, so let us assume $|V_{i,s,t}| \geq a$. Then we calculate:

$$|V_{i+L,s,t}| \leq (|V_{i,s,t}| + L)(1 + 2/(k - 1))^L \leq 1.25|V_{i,s,t}|(1 + 1/(L/\ln 1.5))^L <$$
$$< 1.25|V_{i,s,t}|e^{\ln 1.5} < 2|V_{i,s,t}|.$$

Thus, by using Lemma 2.2 as explained above, we can ensure that weights of nodes on the first level change by at most a factor of 2 during at most L phases in which they are not being rebalanced. This in turn means that nodes on the higher levels also increase weights by no more than a factor of 2.

Similarly, we can calculate that weights of nodes do not decrease by more than a factor of 2 between rebalancings. By definition of augmented sets we have:

$$|V_{i+1,s,t}| \geq \min(|V_{i,s,t}| - 1,\ |V_{i,s,t}|(1 - 2/(k+1))),$$

which means that

$$|V_{i+L,s,t}| \geq (|V_{i,s,t}| - L)(1 - 2/(k+1))^L.$$

Recall that a balanced node on the first level has the weight of at least a, so let us assume $|V_{i,s,t}| \geq a$. Then we calculate, as $a \geq 4L$ and $(1 - 1/x)^{x-1} > e^{-1}$:

$$\begin{aligned}|V_{i+L,s,t}| \geq (|V_{i,s,t}| - L)(1 - 2/(k+1))^L \geq 0.75|V_{i,s,t}|(1 - 1/(L/\ln 1.5 + 1))^L > \\ > 0.75|V_{i,s,t}|e^{-\ln 1.5} = 0.5|V_{i,s,t}|.\end{aligned}$$

Putting together the above properties, we get:

Lemma 4.4. *For n big enough and any sequence of augmented sets obtained with a parameter $k = 2(\log_a n + 1)/\ln 1.5 + 1 = \mathcal{O}(\log_a n)$, all nodes in B-trees $T_1, \ldots, T_m$ are kept semi-balanced with cyclic rebalancing.*

Moreover, by the inequality $|V_{i+1,s,t}| \leq \max(|V_{i,s,t}| + 1,\ |V_{i,s,t}|(1 + 2/(k-1)))$, for $a \geq 3$ and $k \geq 7$ we have that a node which is balanced at the end of a phase i can increase its weight by at most a factor of 4/3 after insertion of the new leaves in phase $i + 1$. We will use this in the next paragraph.

Algorithm 2 Rebalancing a single level h of a B-tree T.

```
1: function REBALANCE($T$, $h$, $a$)
2:     Input: B-tree $T$, an integer $h$, parameter $a$.
3:     Output: $T$ with all nodes on level $h$ being balanced.
4:
5:     if $h$ is the height of $T$ then
6:         if the root of $T$ is balanced then return $T$
7:         else
8:             Create the new root $v$, with the only child being the old root
9:     for all $v$ on a level $h+1$ of $T$ do
10:        Let $S$ be the ordered set of subtrees rooted at the grandchildren of $v$
11:        Discard all children of $v$
12:        while the total weight of subtrees in $S$ is larger than $3a^h$ do
13:            Create a new empty node $u$ as the rightmost child of $v$
14:            while the weight of $u$ is less than $a^h$ do
15:                Add the first subtree $s$ from $S$ as the rightmost child of $u$, set $S \leftarrow S \backslash \{s\}$
16:        if $S \neq \emptyset$ then
17:            Create a new rightmost child of $v$ with all subtrees from $S$ as children, in the same order
18:        if $v$ is the root with the only child $u$ then
19:            Delete $v$, make $u$ the new root
20:    return $T$
```

Rebalancing a single level. In one phase we ensure that all the nodes on some level h are balanced. Let v be any node on level $h + 1$ and S be the ordered set of subtrees rooted at grandchildren of v, that is the subtrees rooted at nodes on level $h - 1$ and in the subtree of v, ordered from left to right. If h is the current height of a tree and the root is not balanced, then v is the freshly created root with the only child being the old root. We discard the children of v and create new ones while distributing the subtrees from S in a balanced way, maintaining their initial order. If $h = 1$ this is trivial as S contains leaves, thus we can repeatedly group them into exactly a leaves and store each group in a new child of v until the size of S drops below $2a$, then we create the last child of v containing the leaves remaining in S. If $h > 1$, then in the previous phase we rebalanced level $h - 1$, so at the end of the previous phase all nodes on level $h-1$ had weight of at most $3a^{h-1}$. Because of insertions in the current phase, their weight might be increased, but not by much. By assumptions on a and k, now all those children still have weight of at most $4a^{h-1}$. As $a \geq 4$, $4a^{h-1} \leq a^h$ holds, meaning that again we can process S in a greedy manner, repeatedly creating new children of v with weight in the range $[a^h, 2a^h]$ and stopping when the total weight of S dropped below $3a^h$, then creating the last child. See Algorithm 2 for a pseudocode.

Encoding $\delta(d, i)$. Let T_i, T_{i+1} be B-trees at the end of two consecutive phases, and consider a value d appearing in both of the trees, that is $d \in V_i \cap V_{i+1}$. Recall that the only differences between T_{i+1} and T_i are insertions/deletions of leaves and rebalancing a single level of a B-tree, say h. Creating new nodes on that single level with Algorithm 2 change only edges from level h to $h-1$ and from level $h+1$ to h, which changes their encodings in $\mathsf{path}(d, T_i)$. Additionally, edges going to the leaves might change too, due to insertions and deletions of new values. This means a change from $\mathsf{path}(d, T_i)$ to $\mathsf{path}(d, T_{i+1})$ is a very local one. We can store new prefix-free codes for at most three edges and pointers to indices in $\mathsf{path}(d, T_i)$, where those codes should replace previous ones. This takes no more than $\mathcal{O}(\log a + \log\log n)$ bits.

Lemma 4.5. *Augmented sets $V_1, V_2, \ldots$ can be represented by semi-balanced B-trees $T_1, T_2, \ldots$ in a way that for any i and $d \in V_i \cap V_{i+1}$ we can obtain a transition label $\delta(d, i)$ with the length of $\mathcal{O}(\log a + \log\log n)$ bits, such that from $\mathsf{path}(d, T_i)$ and $\delta(d, i)$ we can compute in constant time $\mathsf{path}(d, T_{i+1})$.*

5 Labels for a Strong Product

In this section, we put together the framework presented in Section 3 with encodings of paths and transition labels on B-trees from Section 4, to obtain the final adjacency labeling scheme. Note that not much here is changed from the previous work [10], but we present most of the details for completeness.

5.1 Elimination order and interval graphs. It is known that by adding edges we can extend any graph H of treewidth t to some t-tree H', having two important properties [10]. A graph H' is a t-tree if it has a perfect elimination ordering $v_1, \ldots, v_m$ of $V(H')$ such that for any i, neighbors of v_i in H' occuring earlier in the order induce a clique of size at most t. We denote that clique by $C_{H'}(v_i)$, including v_i itself, that is $C_{H'}(v_i) = N_{H'}(v_i) \cap \{v_1, v_2, \ldots, v_i\}$, where $N_{H'}(v_i)$ is the closed set of neighbors of v_i in H_i. This means that we have an orientation of edges of H' such that the closed out-neighborhood of any vertex is a clique of size at most $t+1$ in H'.

We can obtain a t-tree H' from the graph H of treewidth t as follows. First, set $H' = H$ and then for any two vertices $v, w \in V(H)$ contained in some bag of the tree-decomposition of H, add an edge between them if it does not already exist. This way, the subgraph of H' induced by the vertices contained in any single bag is a clique, and the tree-decomposition remains valid, with the same width. Moreover, we can compute a perfect elimination ordering as follows. Root the tree-decomposition of H' arbitrarily and run the depth-first search starting from the root. For each visited bag, iterate through all vertices contained in it, and output all vertices seen for the first time, in an arbitrary order. Ordering in which vertices are output during this DFS is a perfect elimination ordering: assume a node v_i was output during iterating through a bag B. All neighbors of v_i output earlier in the ordering must also be contained in B, and thus form a clique of size at most t.

The second property is a mapping of a t-tree to an interval graph. This is connected to a path-decomposition and the relation between treewidth and pathwidth of a graph.

Proposition 5.1. *For any t-tree H' with m vertices, there exists a function* $\mathsf{interval}$ *mapping $V(H')$ into intervals of integers such that:*

- *If $(v, u) \in E(H')$, then intervals* $\mathsf{interval}(v)$ *and* $\mathsf{interval}(u)$ *intersects.*
- *The number of pairwise intersecting intervals in $\{\mathsf{interval}(v) : v \in V(H')\}$ is $\mathcal{O}(t \log m)$.*

Proof. Here we assume that we are given a tree decomposition of H' of width at most t. We use the well known fact that a graph with m vertices and treewidth t has pathwidth bounded by $\mathcal{O}(t \log m)$, which can be shown as follows.

Let us start with a path of length m and all its bags being empty. First, we find the centroid of the tree decomposition of H' and put all vertices from the bag of the centroid in all bags of the path. Assume that deleting the centroid partition the tree decomposition into k subtrees, then we partition our path into k disjoint subpaths, each with the size equal to the size of some subtree, and proceed recursively for each subtree and its assigned subpath. As the depth of the recursion is $\mathcal{O}(\log m)$, and each time we add no more than $t+1$ vertices to the bags, the maximum size of bags in the path is $\mathcal{O}(t \log m)$. It is not hard to check that we obtain a proper path-decomposition. Then, if a vertex v is contained in bags from a to b on the path, we set $\mathsf{interval}(v) = [a, b]$. ◻

Additionally, we can make sure that the right endpoints of those intervals are distinct. We will treat a graph H' and its interval representation as fixed and exclude it from function notation in most cases.

Storing edges $E(H')$ using a B-tree. The above can be used to establish a correspondence between vertices of H' and nodes of a B-tree T. We store in leaves of T all right endpoints of the intervals from an interval representation of H', and say that such T *represents* H'. Then vertex v is *assigned* to LCA node of $\mathsf{interval}(v)$, that is, the lowest node of T such that all leaves with values in $\mathsf{interval}(v)$ are contained in the subtree of this node. Denote this LCA of $v \in H'$ in T by $\mathsf{lca}(v, T)$. We have the following:

Lemma 5.2. *Assume in a B-tree T representing t-tree H' on m vertices the maximum degree of a node is Δ. Then any node w in T is assigned $\mathcal{O}(\Delta t \log m)$ vertices, that is, $|\{v \in H' : \mathsf{lca}(v,T) = w\}| = \mathcal{O}(\Delta t \log m)$.*

Proof. It follows from the bound on the maximum number of pairwise intersecting intervals from Proposition 5.1. If w is a leaf then we are done, otherwise denote children of w as $c_1, \ldots, c_k$, $k \leq \Delta$, and let a_i for $1 < i \leq k$ be some arbitrary value between the values stored in the rightmost leaf in the subtree of c_{i-1} and the leftmost leaf in the subtree of c_i. Observe that for any vertex v with $\mathsf{lca}(v,T) = w$, $\mathsf{interval}(v)$ must contain some a_i, as otherwise LCA of $\mathsf{interval}(v)$ in T would be on a lower level. Naturally, intervals containing the same a_i are pairwise intersecting, thus we get our claim. ◻

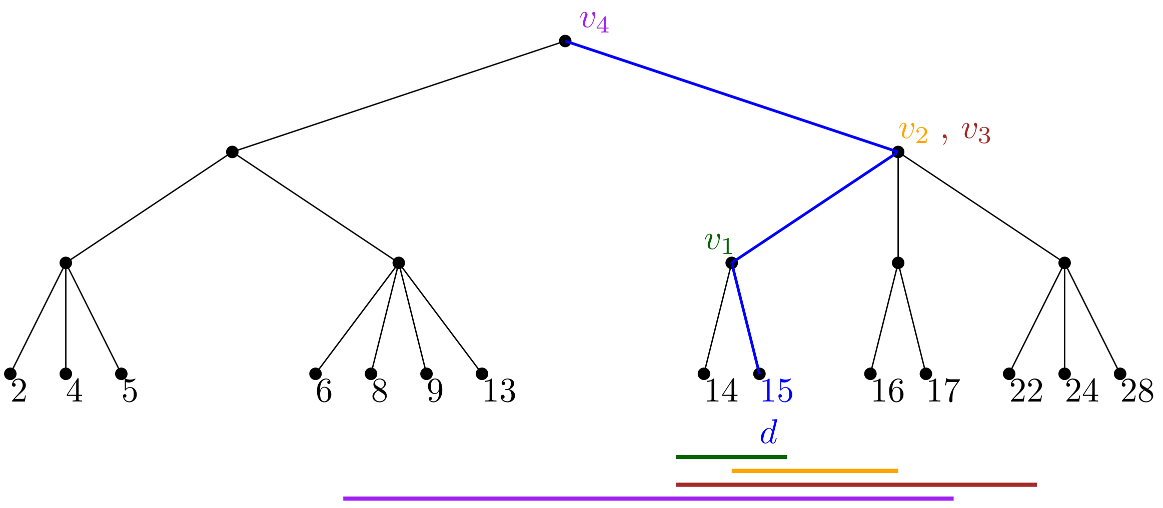

Figure 5: B-tree T representing H', with $C_{H'}(v_2) = \{v_1, v_2, v_3, v_4\}$ and $\mathsf{interval}(v_1) = [14, 15], \mathsf{interval}(v_2) = [15, 16], \mathsf{interval}(v_3) = [14, 22], \mathsf{interval}(v_4) = [8, 17]$. $\mathsf{lca}(v_2, T) = \mathsf{lca}(v_3, T)$. $d = \mathsf{lowest}(v_2, H') = 15$, and all nodes assigned to vertices from $C_{H'}(v_2)$ lie on a single path to the leaf storing value d.

Recall that in T we store all right endpoints of intervals $\mathsf{interval}(u)$ for $u \in H'$. Proposition 5.1 impose strict restrictions on positions of nodes assigned to vertices from $C_{H'}(v)$:

Lemma 5.3. *Assume we are given a B-tree T representing t-tree H' and a vertex $v \in H'$. Let w be the vertex from $C_{H'}(v)$ with the smallest value of the right endpoint of its interval $\mathsf{interval}(w)$, say $\mathsf{interval}(w) = [a, b]$. We denote this smallest right endpoint b by $\mathsf{lowest}(v, H')$. Then all nodes from the set $\{\mathsf{lca}(u,T) : u \in C_{H'}(v)\}$ lie on a path from the root to the leaf with value b.*

Proof. For any $u \in C_{H'}(v)$ we have $\mathsf{interval}(u) \cap \mathsf{interval}(w) \neq \emptyset$. As b is the smallest value of the right endpoint of an interval of any vertex from $C_{H'}(v)$, this means that $b \in \mathsf{interval}(u)$. Therefore by definition of $\mathsf{lca}(u, T)$, it must lie on the path from the root to the leaf with value b. See Figure 5 for a small example. ◻

To sum up, we can represent vertices of H' in a B-tree and orient the edges of H' in such a way that the out-neighborhood of any node is of size at most $t+1$ and lies on a single root-to-leaf path in the tree. Thus, we do not need to separately store paths to all nodes assigned to vertices from the out-neighborhood. It suffices to store a single path and then at most $t+1$ pointers to nodes on that path. Each node of a tree might get assigned many vertices of H', but this is restricted by Lemma 5.2.

5.2 Encoding. Now we have all the tools needed for creating the labels. As any planar graph is a subgraph of $P \boxtimes H$ for a path P and a graph H with treewidth $t \leq 8$, and any H can be extended to a t-tree H', here we will give an adjacency labeling scheme for any n-vertex G subgraph of $P \boxtimes H'$. We consider t to be a constant. Let $V(P) = \{p_1, p_2, \ldots, p_k\}, k \leq n$, and $|H'| = m \leq n$.

Let S_i be the set of vertices of G present in a row i of $P \boxtimes H'$, that is $S_i = \{v \in V(H') : (v, p_i) \in V(G)\}$. First we extend these sets by cliques $C_{H'}(v)$ from the perfect elimination ordering of H', by setting $S'_i = \bigcup_{v \in S_i} C_{H'}(v)$. Then we want to switch to the interval representation, so let X_i be the set of the right endpoints of intervals obtained for vertices in S'_i by applying Proposition 5.1 to H'. Now, let $X'_i = X_{i-1} \cup X_i$, as we need to be able to check edges in consecutive rows. It holds that $\sum_i |X'_i| \leq 2(t+1)n = \mathcal{O}(n)$. Finally, we fix $a = 2^{\sqrt{\log n}}$ and use the fractional cascading approach from Lemma 2.2 and Lemma 4.4 on the sequence $X'_1, \ldots, X'_k$ to obtain a sequence of augmented sets $V_1, \ldots, V_k$ with the property that after $\log_a n + 1 = \sqrt{\log n} + 1$ phases the size of the set changes by at most a factor of two. It holds that $\sum_i |V_i| = \mathcal{O}(n \log n)$, and for all i we have $|V_i| \leq n$ as we never use numbers not present in the initial sets X_i. V_i is a set of numbers, but each number is the unique right endpoint of a vertex from H' by the interval representation, so we can also consider it to be a subset of vertices of H'. We create a sequence of B-trees $T_1, \ldots, T_k$, each T_i storing in its leaves values from V_i, as explained in Section 4.

In order to identify a row of a vertex from $P \boxtimes H'$, we use weighted codes from Lemma 2.1, where weight of a row i is simply $|V_i|$. For $v \in V_i$ we denote that code by $\mathsf{row}(v, i)$, and it holds that $|\mathsf{row}(v, i)| \leq \log n - \log |V_i| + \mathcal{O}(\log \log n)$. Additionally, we have $\mathsf{succlabel}(v, i)$ and $\mathsf{predlabel}(v, i)$ which will allow us to check if other vertices lie in an adjoining row.

Adjacency in rows. To identify edges in a single row, we proceed as explained in Subsection 5.1. Say that we want to create a label for a node (v, p_i), for $v \in S_i$. Using Lemma 5.3 we can efficiently store information about all $C_{H'}(v)$ in the following way. Set $d = \mathsf{lowest}(v, H')$ and recall that all nodes from the set $\{\mathsf{lca}(u, T_i) : u \in C_{H'}(v)\}$ lie on a path from the root to the leaf with the value d. First part of the label will be $\mathsf{path}(d, T_i)$, encoding of the path to d as in Claim 4.3. Since $a = 2^{\sqrt{\log n}}$, we get that $|\mathsf{path}(d, T_i)| = \log |V_i| + \mathcal{O}(\sqrt{\log n})$. The encodings of paths to nodes other than d with assigned vertices from $C_{H'}(v)$ are just prefixes of $\mathsf{path}(d, T_i)$. We need to store at most $t+1$ pointers of length $\mathcal{O}(\log \log n)$ to indices ending those prefixes. Recall that each node in T_i might be assigned many vertices from V_i, but by Lemma 5.2 this amount is bounded by $\mathcal{O}(2^{\sqrt{\log n}} \log n)$. We order the vertices assigned to each node arbitrarily and give them unique identifiers $\mathsf{nodeid}(v, T_i)$, each of length $\mathcal{O}(\sqrt{\log n})$ bits. We concatenate the three mentioned elements and store it as $\mathsf{neigh}(v, T_i)$, that is:

- $\mathsf{path}(d, T_i)$,
- pointers to prefixes being paths to nodes assigned to vertices from $C_{H'}(v)$,
- and the identifier of each vertex $w \in C_{H'}(v)$ among the vertices assigned to $\mathsf{lca}(w, T_i)$.

This is all information needed for adjacency in a single row (which is a subset of H').

Remainder of the label. To allow for checking edges to vertices in the next row $i+1$, we need to know $\mathsf{path}(d, T_{i+1})$. To this end, we store $\delta(d, i)$ from Lemma 4.5, which takes $\mathcal{O}(\sqrt{\log n})$ bits. Then using $\mathsf{path}(d, T_i)$ and $\delta(d, i)$ we can compute $\mathsf{path}(d, T_{i+1})$, and we store the remaining part of $\mathsf{neigh}(v, T_{i+1})$ in the same manner as $\mathsf{neigh}(v, T_i)$.

As we are dealing with a subgraph G of $P \boxtimes H'$, the last parts of the label are $\mathsf{bitarray}((v, p_i), j)$ for $j \in \{-1, 0, 1\}$, that is three bit arrays storing for each $w \in C_{H'}(v)$ a bit 1 iff $((v, p_i), (w, p_{i+j})) \in E(G)$.

Length of a label. Let $d = \mathsf{lowest}(v, H')$. The label $\ell((v, p_i))$ consists of the following:

1. $\mathsf{row}(v, i)$ of length $\log n - \log |V_i| + \mathcal{O}(\log \log n)$, $\mathsf{succlabel}(v, i)$ and $\mathsf{predlabel}(v, i)$ of length $\mathcal{O}(\log \log n)$.
2. $\mathsf{neigh}(v, T_i)$, which is:
   (a) $\mathsf{path}(d, T_i)$, of length $\log |V_i| + \mathcal{O}(\sqrt{\log n})$.
   (b) A constant number of pointers to prefixes of $\mathsf{path}(d, T_i)$, of length $\mathcal{O}(\log \log n)$.
   (c) A constant number of $\mathsf{nodeid}(w, T_i)$ for all $w \in C_{H'}(v)$, of length $\mathcal{O}(\sqrt{\log n})$.
3. A transition label $\delta(d, i)$ of length $\mathcal{O}(\sqrt{\log n})$.
4. $\mathsf{neigh}(v, T_{i+1})$, but with $\mathsf{path}(d, T_{i+1})$ stored indirectly, being computable from the two previous parts of the label. Thus here the total length is $\mathcal{O}(\sqrt{\log n})$.
5. $\mathsf{bitarray}((v, p_i), j)$ for $j \in \{-1, 0, 1\}$ of constant length.

We have that the total length of $\ell((v, p_i))$ is $\log n + \mathcal{O}(\sqrt{\log n})$.

5.3 Decoding. Given two labels $\ell(v), \ell(w)$ the decoder first uses $\mathsf{row}, \mathsf{succlabel}$ and $\mathsf{predlabel}$ to check whether v and w are vertices from one row of $H' \boxtimes P$ or some two consecutive rows. If not, an edge (v, w) cannot exist. Otherwise, we have two cases.

Assume vertices lie in the same row i (note the value of i is not known to the decoder, but this is not needed), so $v = (v', p_i)$ and $w = (w', p_i)$ for $v', w' \in H'$. If there is an edge (v, w), then it has to be that $v' \in C_{H'}(w')$ or $w' \in C_{H'}(v')$. To check whether $w' \in C_{H'}(v')$, for each $u' \in C_{H'}(v')$ we need to compute $\mathsf{path}(u', T_i)$ and $\mathsf{nodeid}(u', T_i)$ using $\mathsf{neigh}(v', T_i)$, then compare it with $\mathsf{path}(w', T_i)$ and $\mathsf{nodeid}(w', T_i)$ computed using $\mathsf{neigh}(w', T_i)$. If for some u' both encodings of paths and identifiers match, we check $\mathsf{bitarray}(v, 0)$ for the bit corresponding to the matched u'. If it is 1, we declare that the vertices are adjacent. Checking whether $v' \in C_{H'}(w')$ is symmetric.

Now assume vertices lie in consecutive rows. Without loss of generality assume that v lies in a row i and w in $i + 1$, so $v = (v', p_i)$ and $w = (w', p_{i+1})$ for $v', w' \in H'$. First we extract $\mathsf{path}(\mathsf{lowest}(v', H'), T_i)$ and $\delta(\mathsf{lowest}(v', H'), i)$, from which we compute $\mathsf{path}(\mathsf{lowest}(v', H'), T_{i+1})$ and thus we have the whole $\mathsf{neigh}(v', T_{i+1})$. With this, we can proceed as in the previous case of a single row, using the respective $\mathsf{bitarray}$ at the end.

Finally, Theorem 1.2 is an immediate consequence of Theorem 3.2 and the following theorem proved in this section:

THEOREM 5.4. *For any constant t, there is an adjacency labeling scheme with labels of length $\log n + \mathcal{O}(\sqrt{\log n})$ for the family of n-vertex subgraphs of $H' \boxtimes P$, where H' is a t-tree with at most n vertices, and P is a path of at most n vertices. The encoder works in polynomial time and the decoder working in constant time in the standard word RAM model.*

Note that the encoder can be implemented in polynomial time because the strong product and mapping to a subgraph from Theorem 3.2 can be found in polynomial time [11].

5.4 Extensions and open problems. As stated in Corollary 1.3, the adjacency labeling from [10] leads directly to an induced-universal graph with $n^{1+o(1)}$ vertices, but that graph can have up to n^2 edges. Very recently, Esperet, Joret and Morin [12] further refined the labeling scheme to obtain an induced universal graph on just $n^{1+o(1)}$ edges:

THEOREM 5.1. ([12]) *For every n, there is an universal graph U_n with $n^{1+o(1)}$ vertices and edges that contains every planar graph on n vertices as an induced subgraph.*

We leave replacing Bulk Trees with B-Trees in their modified construction for future work.

Open problems. We believe that in the case of adjacency labeling schemes for planar graphs, the most important open problem is finding any lower bound on the length of the labels better than trivial $\log n$, say, at least $\log n + \Omega(\log \log n)$. It is known that trees, outerplanar graphs, or generally graphs with bounded treewidth all have adjacency labeling scheme with labels of length just $\log n + \mathcal{O}(1)$, but that might not be the case for planar graphs. Further improving the second-order term in the upper bound is another natural open question.

References

[1] Noga Alon and Vera Asodi. Sparse universal graphs. *Journal of Computational and Applied Mathematics*, 142:1–11, May 2002.

[2] Noga Alon and Michael R. Capalbo. Sparse universal graphs for bounded-degree graphs. *Random Struct. Algorithms*, 31(2):123–133, 2007.

[3] Stephen Alstrup, Søren Dahlgaard, and Mathias Bæk Tejs Knudsen. Optimal induced universal graphs and adjacency labeling for trees. In *56th FOCS*, pages 1311–1326, 2015.

[4] Stephen Alstrup, Cyril Gavoille, Esben Bistrup Halvorsen, and Holger Petersen. Simpler, faster and shorter labels for distances in graphs. In *27th SODA*, pages 338–350, 2016.

[5] Stephen Alstrup, Haim Kaplan, Mikkel Thorup, and Uri Zwick. Adjacency labeling schemes and induced-universal graphs. In *47th STOC*, pages 625–634, 2015.

[6] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory (extended abstract). In *FOCS*, pages 560–569. IEEE Computer Society, 1996.

[7] Marthe Bonamy, Louis Esperet, Carla Groenland, and Alex D. Scott. Optimal labelling schemes for adjacency, comparability, and reachability. In *STOC*, pages 1109–1117. ACM, 2021.

[8] Marthe Bonamy, Cyril Gavoille, and Michał Pilipczuk. Shorter labeling schemes for planar graphs. In *31st SODA*, pages 446–462, 2020.

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

[10] Vida Dujmovic, Louis Esperet, Cyril Gavoille, Gwenaël Joret, Piotr Micek, and Pat Morin. Adjacency labelling for planar graphs (and beyond). In *FOCS*, pages 577–588. IEEE, 2020.

[11] Vida Dujmovic, Gwenaël Joret, Piotr Micek, Pat Morin, Torsten Ueckerdt, and David R. Wood. Planar graphs have bounded queue-number. *J. ACM*, 67(4):22:1–22:38, 2020.

[12] Louis Esperet, Gwenaël Joret, and Pat Morin. Sparse universal graphs for planarity. *CoRR*, abs/2010.05779, 2020.

[13] Louis Esperet, Arnaud Labourel, and Pascal Ochem. On induced-universal graphs for the class of bounded-degree graphs. *Inf. Process. Lett.*, 108(5):255–260, 2008.

[14] Pierre Fraigniaud and Cyril Gavoille. Routing in trees. In *28th ICALP*, pages 757–772, 2001.

[15] Pierre Fraigniaud and Amos Korman. An optimal ancestry scheme and small universal posets. In *42th STOC*, pages 611–620, 2010.

[16] Ofer Freedman, Paweł Gawrychowski, Patrick K. Nicholson, and Oren Weimann. Optimal distance labeling schemes for trees. In *36th PODC*, pages 185–194, 2017.

[17] Cyril Gavoille and Arnaud Labourel. Shorter implicit representation for planar graphs and bounded treewidth graphs. In *ESA*, volume 4698 of *Lecture Notes in Computer Science*, pages 582–593. Springer, 2007.

[18] Pawel Gawrychowski, Wojciech Janczewski, and Jakub Lopuszanski. Shorter labels for routing in trees. In *SODA*, pages 2174–2193. SIAM, 2021.

[19] Pawel Gawrychowski and Przemyslaw Uznanski. Better distance labeling for unweighted planar graphs. In *WADS*, volume 12808 of *Lecture Notes in Computer Science*, pages 428–441. Springer, 2021.

[20] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.

[21] Amos Korman. Labeling schemes for vertex connectivity. *ACM Trans. Algorithms*, 6(2):39:1–39:10, 2010.

[22] J. W. Moon. On minimal n-universal graphs. *Proceedings of the Glasgow Mathematical Association*, 7(1):32–33, 1965.

[23] John H. Muller. Local structure in graph classes. *PhD thesis, School of Information and Computer Science*, 1988.

[24] David Peleg. Informative labeling schemes for graphs. *Theor. Comput. Sci.*, 340(3):577–593, 2005.

[25] Noy Galil Rotbart. *New Ideas on Labeling Schemes*. PhD thesis, University of Copenhagen, 2016.

[26] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *13th SPAA*, pages 1–10, 2001.

[27] Torsten Ueckerdt, David R. Wood, and Wendy Yi. An improved planar graph product structure theorem. *CoRR*, abs/2108.00198, 2021.