Uniwersytet Wrocławski
Wydział Matematyki i Informatyki

Schematy Etykietowania Grafów oraz Dynamiczny Najdłuższy Podciąg Rosnący

Autor:
Wojciech Janczewski

Rozprawa sporządzona pod opieką promotorów:
dr Paweł Gawrychowski, prof. UWr
prof. dr hab. Krzysztof Loryś

2024

University of Wrocław
Faculty of Mathematics and Computer Science

Graph Labeling Schemes and Dynamic Longest Increasing Subsequence

Author:
Wojciech Janczewski

Doctoral dissertation supervised by:
dr Paweł Gawrychowski, prof. UWr
prof. dr hab. Krzysztof Loryś

2024

Streszczenie

Niniejsza rozprawa składa się z publikacji poświęconych dwóm tematom badawczym. Pierwszym i głównym jest zdecentralizowana reprezentacja grafów, sformalizowana jako schematy etykietowania. W tym podejściu, zamiast przechowywać globalną strukturę danych, rozdzielamy informację o grafie na części zapisane w pojedynczych wierzchołkach. Każdemu wierzchołkowi przypisywany jest ciąg bitów (etykieta) w taki sposób, że posiadając etykiety dwóch wierzchołków, możemy bez żadnych dodatkowych informacji poznać, na przykład, odległość w grafie między nimi. Drugim tematem są algorytmy dynamiczne, podobnie jak etykietowania przydatne w obsłudze wielkich zbiorów danych; ich celem jest umożliwienie modyfikacji danych wejściowych bez konieczności każdorazowego uruchamiania klasycznych algorytmów od zera.

Ta praca koncentruje się na trzech schematach etykietowania. Po krótkim wstępie na temat wielofunkcyjnego schematu etykietowania drzew (łączącego sąsiedztwo z zapytaniami o bycie przodkiem), opisano wydajny pamięciowo schemat etykietowania dla trasowania w drzewach. W tym scenariuszu, otrzymawszy etykiety dwóch wierzchołków u, v chcemy poznać numer pierwszej krawędzi na najkrótszej ścieżce od u do v. Istniejące metody etykietowania dla trasowania w grafach ogólnych używają jako podproblemu trasowania w odpowiednio dobranych drzewach.

Następnie przedstawiony jest schemat etykietowania dla sąsiedztwa w grafach planarnych, w którym proponowane uproszczenie uprzednio znanych metod usprawnia efektywność pamięciową i czasową. Ponieważ sąsiedztwo w grafach ogólnych wymaga dużych etykiet o rozmiarze wielomianowym, klasy grafów dla których istnieją bardziej efektywne etykiety wielkości logarytmicznej mają szczególne znaczenie, a grafy planarne są jedną z największych takich klas. W tym kontekście ważne jest również powiązanie etykietowań dla sąsiedztwa z grafami uniwersalnymi.

Kolejny rozdział poświęcony etykietom przedstawia optymalny schemat dla odległości w grafach permutacyjnych. Poza sąsiedztwem to właśnie odległość jest funkcją najczęściej rozważaną dla etykietowań, a dla grafów permutacyjnych istnieją szczególnie wydajne schematy.

W ostatnim rozdziale opisano dynamiczną aproksymację dla problemu najdłuższego podciągu rosnącego (LIS), jednego z klasycznych problemów algorytmicznych. Istniejące metody wyznaczające LIS dla dynamicznie zmieniającej się tablicy działają w czasie wielomianowym. Przejście z rozwiązań dokładnych do aproksymacji pozwala nam na uzyskanie czasu logarytmicznego, nawet w złożoności najgorszego przypadku. Nie są znane (nietrywialne) ograniczenia dolne dla dynamicznego LIS, dlatego też następnie wprowadzane są naturalne warianty ważonych elementów i zapytań 1D, w których da się uzyskać warunkowe wielomianowe ograniczenia dolne.

Abstract

This dissertation is composed of papers on two research topics. The first and main one is a decentralised representation of graphs, formalised as labeling schemes. Instead of storing one global data structure, we aim to evenly distribute information about the graph among single vertices. Each vertex is assigned a bitstring, called its label, such that later given the labels of two vertices and no additional information, we can decide for example whether these two vertices are adjacent in a graph, or what is the distance between them. The second topic is dynamic algorithms, also useful in handling large data sets, as their purpose is to enable data modification without the need to every time recompute regular algorithms from scratch.

The focus of this work is on three labeling schemes. After a short warm-up about multi-functional labeling schemes for trees (combining ancestry, adjacency, and more), an efficient routing labeling scheme for trees is described. In such a scenario, given the labels of two vertices u, v we want to decide on a number of the first edge on the shortest path from u to v. Trees are especially interesting in this context, as many existing routing methods in general graphs depend, as a subproblem, on routing in trees.

Then, a simpler than previously existing adjacency labeling scheme for planar graphs is designed. As adjacency in general graphs requires large polynomial-sized labels, classes of graphs for which more viable logarithmic-size labels exist are of special importance, and planar graphs are one of the most complex among them. Adjacency labelings are also important due to their connection to the notion of induced-universal graphs, interesting combinatorial objects.

Another subject is an optimal labeling scheme for distances in permutation graphs. In the literature, distance is the most often considered function beyond adjacency, and permutation graphs admit particularly efficient schemes for distance. This is also worth attention as it closes an open problem by matching an existing lower bound up to the second-order term.

The dissertation ends with the description of fully dynamic approximation for the longest increasing subsequence (LIS) problem. LIS is one of the oldest algorithmic problems with a classic dynamic programming solution. Existing methods for exact dynamic LIS work in polynomial time, but approximation allows for polylogarithmic time, even worst-case. There are no known (nontrivial) lower bounds for dynamic LIS, thus weighted and 1D query natural variants of the problem are considered, for which conditional polynomial time lower bounds are designed.

Contents

1 Introduction **11**

1.1 Outline . . . 11

1.2 Labeling Schemes . . . 12

1.3 LIS and Dynamic Algorithms . . . 21

1.4 Results . . . 24

1.5 Preliminaries . . . 31

2 Warm-up: Multi-functional Labeling for Trees **33**

2.1 Schemes for Trees . . . 33

2.2 Classic Ancestry Labeling . . . 34

2.3 Ancestry Labeling via Rounding . . . 35

2.4 moveUp Function . . . 38

3 Shorter Labels for Routing in Trees **43**

3.1 Introduction . . . 43

3.2 Preliminaries . . . 44

3.3 Framework for Labeling Schemes . . . 45

3.4 Intermediate Scheme with Double Rounding . . . 50

3.5 Final Scheme with Distribution of Bits . . . 59

3.6 Simple Extensions . . . 65

3.7 Lower Bound of $\log n + \Omega(\log \log n)$ . . . 66

3.8 Conclusions . . . 67

3.9 Appendix: Constant Time Query . . . 67

4 Simpler Adjacency Labeling for Planar Graphs **75**

4.1 Preliminaries . . . 75

4.2 Framework . . . 77

4.3 Maintaining B-trees . . . 80
4.4 Labels for a Strong Product . . . 84
4.5 Sparse Induced-Universal Graphs . . . 90

5 Optimal Distance Labeling for Permutation Graphs **95**
5.1 Overview and Organisation . . . 95
5.2 Preliminaries . . . 98
5.3 Scheme of Size $5\log n$ . . . 98
5.4 Final Scheme of Size $3\log n$ . . . 107
5.5 Conclusion . . . 115

6 Dynamic Approximation of LIS in Polylogarithmic Time **117**
6.1 Preliminaries . . . 120
6.2 Covers and Sparsification . . . 121
6.3 Decremental Structure . . . 128
6.4 Analysis of the Decremental Structure . . . 134
6.5 Erdős-Szekeres Partitioning . . . 137
6.6 Fully Dynamic Worst-Case Structure . . . 138
6.7 Conditional lower bounds . . . 145
6.8 Conclusions and Open Problems . . . 151

Chapter 1

Introduction

1.1 Outline

This dissertation is composed of several research papers written with co-authors, preceded by the chapter introducing their topics and describing areas of research, with a focus on informative labeling schemes for graphs. The introduction begins with a detailed discussion of the notion of labeling schemes, providing basic definitions, examples, existing results, and different directions pursued by researchers in the area. Then, much more briefly, the Longest Increasing Subsequence problem and the context of dynamic algorithms are described. Next, I list publications of which this dissertation consists, each with a summary of the results. Finally, a short technical preliminary section follows.

In Chapter 2, as a warm-up helping to better understand the area, multi-functional labeling for trees is presented. This deals with ancestry, adjacency, parent/siblings, or k-distance queries. Chapter 3 describes a labeling scheme for routing in trees, in the designer-port model. Chapter 4 is a presentation of simplified adjacency labeling for planar graphs, extended to also provide sparse induced-universal graphs for planar graphs. In Chapter 5, I describe an optimal labeling scheme for distance in permutation graphs. In the last Chapter 6 a fully dynamic structure for approximation of the longest increasing subsequence is presented, handling updates and queries in polylogarithmic time. It is followed by conditional lower bounds for dynamic algorithms for specific variants of this problem.

The thesis is based on the following papers:

- SODA 2021 paper *Shorter Labels for Routing in Trees* [GJL21], joint work with Paweł Gawrychowski and Jakub Łopuszański.
- SOSA 2022 paper *Simpler Adjacency Labeling for Planar Graphs with B-Trees* [GJ22], written with Paweł Gawrychowski.
- A preprint *Optimal Distance Labeling for Permutation Graphs* [GJ24], joint work with Paweł Gawrychowski.

- STOC 2021 paper *Fully Dynamic Approximation of LIS in Polylogarithmic Time* [GJ21b], written with Paweł Gawrychowski.
- A preprint *Conditional Lower Bounds for Variants of Dynamic LIS* [GJ21a], joint work with Paweł Gawrychowski.

1.2 Labeling Schemes

Definitions. The notion of informative labeling schemes, formally introduced by Peleg [Pel05], is a framework capturing the idea that for modern scattered networks of huge size, which often need to be accessed from different locations as opposed to being represented in a single place by a centralised data structure, it is desirable to select the identifiers of the vertices as to encode some information about the underlying topology of the network graph.

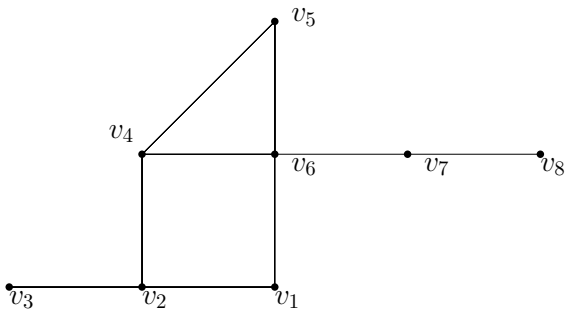
Instead of storing a single global data structure, a labeling scheme assigns each vertex u a binary string $\ell(u)$, called its label. Later, given just the labels of two vertices and no additional information about the graph, we should be able to compute some function defined on those two vertices. In most cases the length of each *individual* label is much smaller than the size of an optimal centralised structure, often by a factor close to $\Theta(n)$, that is, we are able to evenly distribute the information among labels of vertices.

In this context perhaps the most fundamental function is adjacency as considered by Kannan, Naor, and Rudich [KNR92], where we simply want to decide whether two given vertices are neighbors in the graph. Interestingly, as observed by Kannan et al., the question of designing an adjacency labeling scheme for a given class of graphs is in fact equivalent to constructing the so-called induced-universal graph, that is, a larger graph containing each graph from the class as a vertex-induced subgraph. This purely combinatorial problem has been been already studied in 60s by Moon and Rado [Moo65, Rad64], while at the same time, some early variants of labelings were also studied by Breuer and Folkman [BF67].

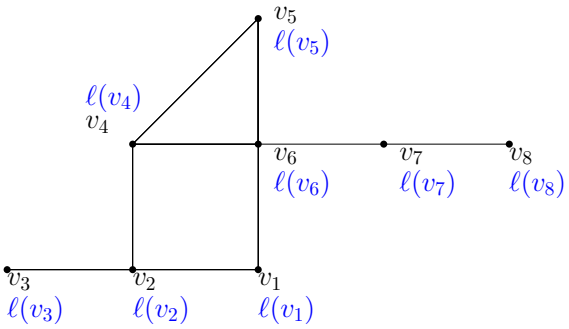
Let $\mathcal{G}$ be a family of graphs. An adjacency labeling scheme for $\mathcal{G}$ consists of an encoder and a decoder. The encoder takes a graph $G \in \mathcal{G}$ and assigns a label (binary string) $\ell_G(u)$ to every vertex $u \in V(G)$. The decoder receives labels $\ell_G(u)$ and $\ell_G(v)$, such that $u, v \in V(G)$ for some $G \in \mathcal{G}$ and $u \neq v$, and should report whether u, v are adjacent in $V(G)$. The decoder is not aware of G and only knows that u and v come from the same graph belonging to $\mathcal{G}$. See Figure 1.1.

An answer of the decoder for labels taken from two different graphs is undefined and can be arbitrary. We are primarily interested in minimizing the maximum length of a label, that is, $\max_{G\in\mathcal{G}} \max_{u\in V(G)} |\ell_G(u)|$, as a function of the number of vertices in a graph. Efficient implementation of the decoder is the secondary goal. Usually, not much attention is paid to optimising running time of the encoder, as long as it is polynomial; in practice, existing schemes often admit near-linear encoders.

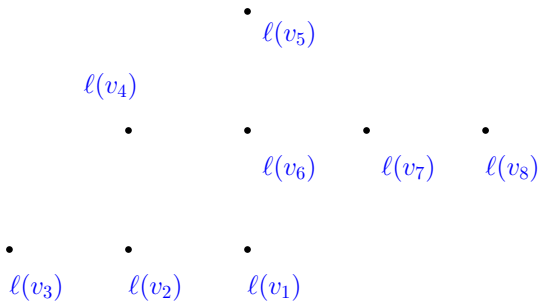
Non-trivial adjacency labeling schemes have been constructed for many classes of graphs, for example undirected, directed, and bipartite graphs, graphs of bounded degree or tree-width,



(a) The encoder receives a graph.



(b) The encoder assigns labels to each vertex.



(c) Only the labels are stored, and the graph is discarded.

(d) The decoder receives the labels of two vertices.

Figure 1.1: A labeling scheme.

trees, planar graphs, comparability graphs, or general families of hereditary graphs. The question of efficient labelings has been considered for many functions beyond adjacency, among others distances, connectivity, flows, ancestry or nearest common ancestor in trees, or routing. In some cases, approaches like randomised labelings or fault-resistant schemes are considered. Depending on the graph class and function being analysed, the size of labels can vary enormously.

Examples. Before reviewing existing bounds, let us see some examples, in order to better grasp the notion of labeling schemes. We use $|V(G)| = n$, and will denote $\lceil \log_2 n \rceil$ as just $\log n$. Moreover, we usually write $\ell(v)$ instead of $\ell_G(v)$, as G is clear from the context.

If we consider adjacency in trees, it is not hard to design a scheme with the labels of size $2 \log n$ bits. Indeed, if vertices of a given tree T are numbered with natural numbers from 1 to n, then we can arbitrarily root the tree and then for any $u \in V(T)$ store as a label a pair of numbers, ID of u and ID of its parent. These two values together clearly take $2 \log n$ bits to store. The decoder just checks for equality, as two vertices are adjacent if and only if one is a parent of the other. Given $\ell(u) = (u_1, u_2)$ and $\ell(v) = (v_1, v_2)$, u is adjacent to v if and only if $u_1 = v_2$ or $u_2 = v_1$. See Figure 1.2. Observe that only the labels are available, if we want to use ID of a vertex it needs to be stored in a label and we need to account for its size, this is not free information for the decoder.

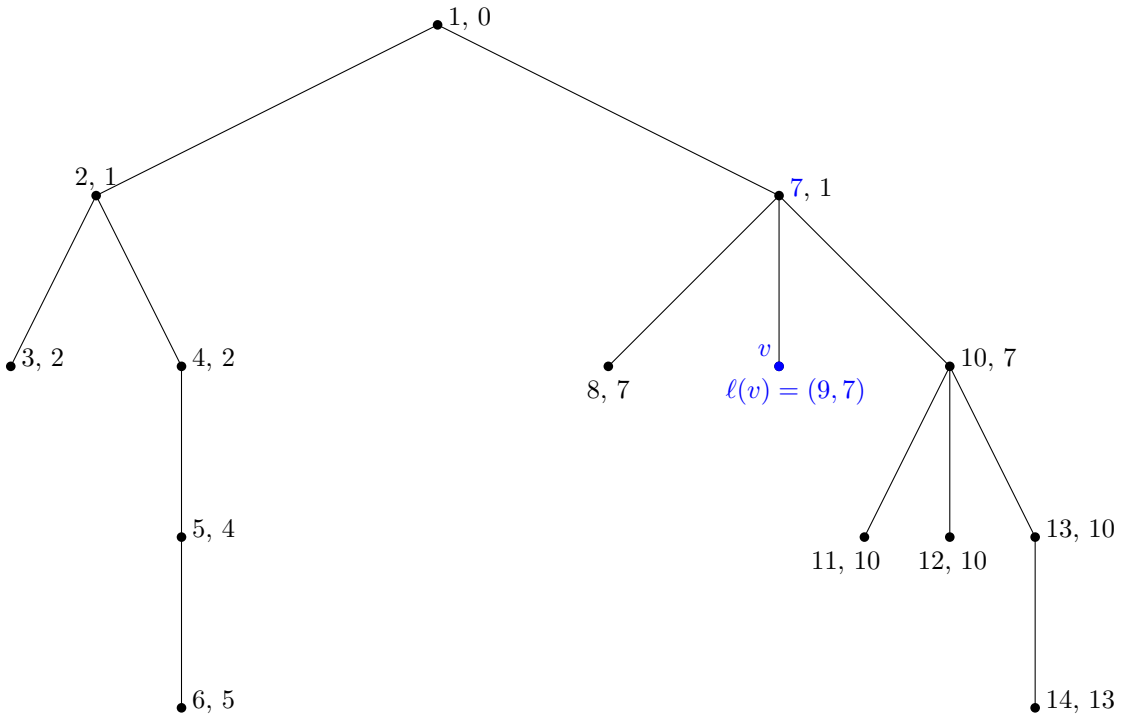


Figure 1.2: Adjacency labelings for trees. For each vertex, the first number is an ID of this vertex, and the second number is the ID of its parent. We have $\ell(v) = (9, 7)$. Here, IDs are assigned in a pre-order traversal, but in fact, they can be arbitrary.

Let us now consider general undirected graphs. We can easily obtain the labels of size $n+\log n$ for adjacency, by storing the ID of a vertex and then a full row of an adjacency matrix of the given graph, so bits indicating adjacency. Reducing this to $n/2+\log n$ is not hard, as for vertex i we can store half of its row, just $n/2$ bit values starting from the one indicating adjacency with vertex $i+1$, up to $i+n/2$ (where we have circular $n+1 \equiv 1$ and so on). When the decoder is given two labels, it can check IDs of vertices and it is guaranteed that the adjacency bit is present in one of the stored half rows.

From Euler's formula, it follows that in a planar graph, there is always a vertex of degree at most 5. Therefore, we can obtain an adjacency labeling scheme for planar graphs with label size $6 \log n$, by repeatedly taking a vertex with the lowest degree, storing as its label its ID and IDs of its neighbours, then removing it from the graph. All edges will be oriented and stored.

From the above examples, we see that it is often quite easy to design asymptotically optimal labeling schemes. As will be described later, due to purely information-theoretic lower bounds based on numbers of different graphs in a given class, there is no adjacency labeling for trees or planar graphs better than $\log n$ bits, and for undirected graphs better than $n/2$ bits. This is one of the reasons why in this area of research we are interested in the bounds optimal up to the lower-order terms, not just asymptotics.

As the final example, to consider some function different than adjacency, let us examine ancestry in rooted trees. The decoder, given the labels of two vertices, should be able to decide whether one is an ancestor of the other. To construct the labels, we assign vertices of a tree

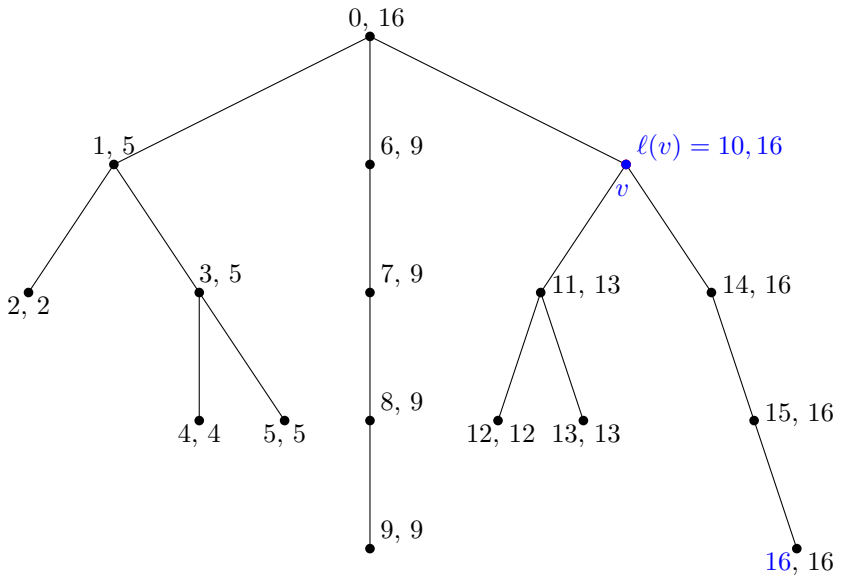


Figure 1.3: For each vertex, the first number is an ID equal to the pre-order number, and the second number is the maximum value occurring in its subtree. We have $\ell(v) = (10, 16)$.

their pre-order numbers in the depth-first search of the tree, so the order in which they are first visited. This is the first half of a label. As the second part of a label, for a vertex u we store the maximum ID occurring in a subtree rooted at u. These two values togeter take a total of $2 \log n$ bits to store. Given $\ell(u) = (u_1, u_2)$ and $\ell(v) = (v_1, v_2)$, u is a proper ancestor of v if and only if $u_1 < v_1 \leq u_2$. See Figure 1.3.

Information-theoretic lower bounds. We have seen some introductory examples, now to present some fundamental lower bounds we need the following observation:

Property 1.1. *If there is a labeling scheme of size $S(n)$ for the function f and family of graphs $\mathcal{G}$, then there also exists labeling for this function and family with size $S(n)+1$ such that for any given graph, the encoder always assigns to all vertices labels of the same length.*

This can be achieved by first using the existing scheme, then checking the size k of the largest label and prepending to each label a bitstring 0^*1 so that after this change all labels are of size exactly $k + 1$ bits. The decoder is able to discard these prefixes.

Now, assume we are given a labeling scheme for the family of undirected graphs on n vertices, with equal label sizes. There are exactly $2^{n(n-1)/2}$ such graphs, as we can consider labeled graphs. Observe that after the encoder is finished, we can concatenate all labels (in order of the vertices) to produce a full description of the graph. Indeed, given the decoder and the concatenation of all labels, we can extract all individual labels and then check for adjacency for all pairs of vertices, reconstructing the whole graph. Therefore, the sum of the labels' sizes must be at least $\log\left(2^{n(n-1)/2}\right)$, which means that the labels must be of size at least $(n-1)/2$ bits. This means that any adjacency labeling for undirected graphs must have the size of at least $n/2 - \mathcal{O}(1)$ bits. Similarly, we can derive $\log n - \mathcal{O}(1)$ lower bound for adjacency in trees, and so also in planar

Class	Lower bound	Upper bound
Trees	$\log n$	$\log n + \mathcal{O}(1)$ [ADK15]
Planar graphs	$\log n$	$\log n + \mathcal{O}(\sqrt{\log n})$ [DEG+21, GJ22]
Graphs of max degree Δ	$(\Delta/2)\log n$	$(\Delta/2)\log n + \mathcal{O}(1)$ [AN17, But09]
Graphs of treewidth k	$\log n + \Omega(k)$	$\log n + \mathcal{O}(k \log\log n/k)$ [GL07]
General graphs	$n/2$	$n/2 + \mathcal{O}(1)$ [Alo17, AKTZ15]
Directed graphs	n	$n + \mathcal{O}(1)$ [Alo17, AKTZ15]
Tournaments	$n/2$	$n/2 + \mathcal{O}(1)$ [Alo17, AKTZ15]
Bipartite graphs	$n/4$	$n/4 + \mathcal{O}(1)$ [Alo17, AKTZ15]

Table 1.1: Selected state-of-the-art bounds for adjacency labeling schemes. In some cases the optimal additive constants were obtained, usually small values like 1 or 4.

graphs. Going back to the previous examples, we see that simple labeling schemes are very close to these bounds, but are not matching in any case. With slightly more careful analysis, we can establish exact additive constants for lower bounds, usually just equal to plus one. While existing labeling schemes almost always assign unique labels, such an assumption is not necessary for this kind of argument, see for example discussion in [AKTZ15].

Adjacency labeling schemes and universal graphs. Without a doubt, adjacency is a function most often considered in the context of labeling schemes. Table 1.1 presents results for some of the most important classes of graphs, although many more subclasses were researched. Notably, Alstrup, Dahlgaard, and Knudsen [ADK15] gave an optimal adjacency labeling for trees with size $\log n + \mathcal{O}(1)$ bits, while Alstrup, Kaplan, Thorup, and Zwick [AKTZ15] presented an optimal labeling for general graphs with size $n/2 + \mathcal{O}(1)$ bits, with constant later improved on by Alon [Alo17]. For planar graphs, the second-order term of lower and upper bounds do not match. In particular, we see that in the case of adjacency, basically all existing lower bounds come from simple counting (information-theoretic) arguments, and we usually can match these.

Adjacency labelings are particularly interesting because of their direct connection to *induced-universal graphs* (or just universal), a very interesting combinatorial object. U is an universal graph for the family $\mathcal{G}$ if every $G \in \mathcal{G}$ is isomorphic to some *induced* subgraph of U. Kannan, Naor, and Rudich [KNR92] in their early work already established a connection between such an object and adjacency labelings. See also Moon [Moo65].

Assume there is an adjacency labeling scheme for $\mathcal{G}$ with the length of the labels bounded by $f(n)$, such that the encoder assigns unique labels to vertices of a given G from $\mathcal{G}$. This means that we can create a universal graph for $\mathcal{G}$ on at most $2^{f(n)}$ vertices, identifying the set of all possible labels with the set of vertices and creating an edge between two vertices if and only if the adjacency query on two respective labels is answered positively by the decoder. Then given any $G \in \mathcal{G}$, we can use the encoder to obtain an induced subgraph isomorphic to G. Conversely, we can use the universal graph to assign the labels, since the encoder can identify an isomorphic subgraph and use it to assign the labels. Therefore, for a given family there is an adjacency

Class	Lower bound	Upper bound
Trees	$1/4\log^2 n - \mathcal{O}(\log n)$ [AGHP16b]	$1/4\log^2 n + o(\log^2 n)$ [FGNW17]
Trees, $(1+\epsilon)$-approximate	$\Omega(\log(1/\epsilon)\log n)$ [FGNW17]	$\mathcal{O}(\log(1/\epsilon)\log n)$ [FGNW17]
Planar graphs	$\Omega(n^{1/3})$ [GPPR04]	$\mathcal{O}(\sqrt{n})$ [GPPR04, GU23]
Graphs of bounded degree	$\Omega(\sqrt{n})$ [GPPR04]	$\mathcal{O}(n\log\log n/\log n)$ [GKU16]
Graphs of bounded treewidth	-	$\mathcal{O}(\log n^2)$ [GPPR04]
Families of cycles	$4/3\log n - \mathcal{O}(1)$ [KPR10]	$3/2\log n$ [SR24]
General graphs	$n/2$	$(\log 3)n/2 + o(n)$ [AGHP16a]
Graphs of max weight W	$1/2(n-1)\lceil W/2+1\rceil$ [AGHP16a]	$1/2n\log(2W+1) + o(n)$ [AGHP16a]
Sparse graphs	-	$o(n)$ [ADKP16]
Interval graphs	$3\log n - \mathcal{O}(\log\log n)$ [GP08]	$3\log n + \mathcal{O}(\log\log n)$ [HW24]
Permutation graphs	$3\log n - \mathcal{O}(\log\log n)$ [BG05]	$3\log n + \mathcal{O}(\log\log n)$ [GJ24]

Table 1.2: Selected state-of-the-art bounds for distance labeling schemes.

labeling scheme of size k if and only if there is an universal graph with 2^k vertices, and these two problems are equivalent. However, we need to note that transforming universal graph into adjacency labeling in a straightforward way is algorithmically unfeasible, as the encoder needs to find an isomorphic subgraph, and both the encoder and the decoder needs to know the (huge) universal graph.

Distance labeling schemes. Adjacency is arguably the most natural function to consider for its theoretical implications, but beyond it another widely researched function for the labeling schemes is distance. Distance schemes have more practical impact, in the context of routing and network communication. In Table 1.2, some of the known results are presented. Of particular note, for general undirected graphs, Alstrup, Gavoille, Halvorsen, and Petersen [AGHP16a] constructed distance labeling of size $(\log 3)n/2 + o(n)$, while the known lower bound is $n/2$ bits (so just coming from the adjacency). In case of planar graphs, scheme of size $\mathcal{O}(\sqrt{n})$ bits is presented by Gawrychowski and Uznanski [GU23], and the known lower bound is $\Omega(n^{1/3})$ bits. For trees, we do not need a polynomial number of bits, as they can be labeled for distances using only $1/4\log^2 n + o(\log^2 n)$ bits as shown by Freedman, Gawrychowski, Nicholson, and Weimann [FGNW17], which is optimal up to the second-order terms [AGHP16b]. Of course, the interesting question is to find natural classes of graphs that admit small logarithmic distance labeling schemes, and we have examples in interval and permutation graphs. There are also some interesting approximate schemes, usually much more efficient than the exact ones. For example, in [AGHP16a] authors present a distance labeling with an additive error of 1 for general graphs with the labels' size of $n/2+o(n)$ bits, which ignoring the second-order term has the same length as just adjacency labeling.

Note that for distances, we have many non-trivial lower bounds, as opposed to adjacency.

Schemes for trees. Trees received large attention in the context of labeling schemes, partly due to simple structure allowing for very low labels' sizes, and partly because of the existence of

Class	Lower bound	Upper bound
Adjacency	$\log n$	$\log n + \mathcal{O}(1)$ [ADK15]
Distance	$1/4 \log^2 n - \mathcal{O}(\log n)$ [AGHP16b]	$1/4 \log^2 n + o(\log^2 n)$ [FGNW17]
$(1+\epsilon)$-approximate distance	$\Omega(\log(1/\epsilon) \log n)$ [FGNW17]	$\mathcal{O}(\log(1/\epsilon) \log n)$ [FGNW17]
Ancestry	$\log n + \Omega(\log \log n)$ [ABR05]	$\log n + \mathcal{O}(\log \log n)$ [FK10b, DKR15]
Nearest common ancestor	$1.008 \log n - \mathcal{O}(1)$ [AHL14]	$2.318 \log n$ [GKo+18]
Siblings/connectivity (unique)	$\log n + \Omega(\log \log n)$ [ABR05]	$\log n + \mathcal{O}(\log \log n)$ [ABR05]
Fixed-port routing	$\Omega(\log^2 n / \log \log n)$ [FG02]	$\mathcal{O}(\log^2 n / \log \log n)$ [FG01]
Designer-port routing	$\log n + \Omega(\log \log n)$ [ABR05]	$\log n + \mathcal{O}((\log \log n)^2)$ [GJL21]
k-distance [FGNW17]	$\log n + \Omega(k \log(\log n / k \log k))$	$\log n + \mathcal{O}(k \log(\log n / k \log k))$

Table 1.3: Selected bounds for different functions on trees (forests). For k-distance the decoder outputs the correct answer only for distances at most k, here are sizes for $k < \log n$.

numerous functions related specifically to them. Table 1.3 summarises some of the results, we already mentioned adjacency, distance, and ancestry, but there are also nearest (lowest) common ancestor, routing (in two models), a sibling or parent relation and k-distance generalising it. For more results see also (slightly outdated) survey by Rotbart [Rot16]. We highlight optimal result of $\log n + \mathcal{O}(\log \log n)$ for ancestry by Dahlgaard, Knudsen and Rotbart [DKR15], as here we see a separation between optimal sizes for adjacency and ancestry. We note that these results hold for family of forests as well.

Often labelings for trees can be extended to other graphs due to specific decompositions. For example, graphs with low arboricity admit efficient adjacency labeling schemes, as we can use concatenation of labelings for individual forests (and adjacency labelings for trees work also for forests). Similarly, routing in general graphs often uses routing on its spanning trees as a subproblem. Ancestry labelings for trees are also useful in the context of XML files [AAK+06, FK10a, CKM02].

Other functions and variants. Here we will shortly note some other functions considered for labeling schemes. Katz, Katz, Korman, and Peleg [KKKP04] described labeling of size $\mathcal{O}(\log n \log W)$ for flow in general graphs with maximum integral capacity W. Bonamy, Esperet, Groenland, and Scott [BEGS21] proved that there is a reachability labeling scheme for directed graphs of size $n/4 + o(n)$, which is optimal up to the second-order terms.

A recently explored direction are fault-tolerant labelings, which can come in different variants, one of which is such that we want the decoder to compute the distance between two vertices given additionally a label of the third forbidden vertex (the shortest path must avoid it). In [BCG+22], fault-tolerant labeling for planar graphs using $\mathcal{O}(n^{2/3})$ bits is shown; this result is to be compared with $\mathcal{O}(\sqrt{n})$ standard distance labeling. Forbidden-set connectivity labelings for planar graphs were already considered in [CGKT08]. Fault-tolerant connectivity schemes for general graphs were developed in [IEWM23], with the labels for vertices and edges where sets of edges can be faulty (forbidden). Also recently, in [PPP24] the authors presented connectivity labeling in the variant of faulty vertices, with additional implications for fault-tolerant routing.

Randomised labeling schemes were studied in the context of universal graphs and hereditary graph families by Harms, Wild, and Zamaraev [Har20, HWZ22], and also in [NP24]. Some fault-tolerant labeling schemes are also randomised, or have smaller sizes in the randomised variants, especially connectivity and routing schemes by Dory and Parter [DP21].

In [TWZ23], authors consider semi-distributed graph representations, which are meant to bridge labeling schemes and centralised oracles, in a scenario when a relatively small global data structure is allowed in order to reduce the labels' sizes below conventional lower bounds.

On constants. At this point, after seeing a multitude of results for labelings across many variants of graph classes and functions, we are better positioned to touch on the subject of optimising constants. At the very beginning, we have seen that it is often very simple to obtain asymptotically optimal labeling schemes, like $\mathcal{O}(\log n)$ for adjacency in trees or $\mathcal{O}(n)$ bits for general graphs. Yet, many researchers were interested in investigating optimal first-order terms or even second-order terms, not simply asymptotics.

This has been successfully done for multiple classes, e.g. distance labeling for trees, where $\mathcal{O}(\log^2 n)$ [Pel00] was improved to $(1/2)\log^2 n$ [AGHP16b] and $(1/4)\log^2 n + o(\log^2 n)$ [FGNW17], optimal up to the second-order term. Adjacency labeling for planar graphs is a particularly good example, with the first scheme having the size of $6\log n$ based on [Mul88], then $4\log n$ [KNR92], $(2+o(1))\log n$ [GL07], $(4/3+o(1))\log n$ [BGP20], finally $\log n + \mathcal{O}(\sqrt{\log n \log\log n})$ [DEG+21] and $\log n + \mathcal{O}(\sqrt{\log n})$ [GJ22] schemes were presented, the last two being optimal up to the second-order terms. See also Table 1.4.

In the case of adjacency labelings for general undirected graphs, starting with the classical result presenting the labels of size $n/2 + \mathcal{O}(\log n)$ [Moo65], $n/2 + \mathcal{O}(1)$ [AKTZ15] and $n/2 + 1$ [Alo17] labelings were constructed. Similar sharply optimal labelings are also shown for directed graphs, tournaments, bipartite graphs, and oriented graphs. Finally, the first described ancestry labeling schemes for trees was of size $2\log n$ [KNR92], and subsequently $1.5\log n$ [AAK+06], $\log n + \mathcal{O}(\log n/\log\log n)$ [TZ01], $\log n + \mathcal{O}(\sqrt{\log n})$ [AR02a], $\log n + 4\log\log n + \mathcal{O}(1)$ [FK10b], and $\log n + 2\log\log n + \mathcal{O}(1)$ [DKR15] schemes were provided, achieving optimality up to second-order terms. There are, of course, more examples, but hopefully, we made a point that obtaining tight bounds is usually not a trivial task.

We can find several reasons to search for optimal results, beyond scientific curiosity and the fact that in other areas we also might not be satisfied in, say, stating that some problem is in P and has solution working in time $\mathcal{O}(n^9)$. In the context of adjacency labeling schemes and universal graphs, recall that the difference between $2\log n$ and $\log n$ labeling is a difference between universal graphs on n^2 and n vertices, so multiplicative change becomes much worse exponential difference. As universal graphs are interesting combinatorial objects, this encouraged a better understanding of adjacency labeling schemes, even optimising the second-order terms.

While considering second-order terms, at some point in history the best known results were $\log n + \mathcal{O}(\log\log n)$ labelings for adjacency and ancestry in trees, but lower bounds were respectively $\log n$ and $\log n + \Omega(\log\log n)$. Obtaining an optimal $\log n + \mathcal{O}(1)$ adjacency labeling

Adjacency in planar graphs
$6 \log n$, for example [Mul88]
$4 \log n$ [KNR92]
$(2 + o(1)) \log n$ [GL07]
$(4/3 + o(1)) \log n$ [BGP20]
$\log n + \mathcal{O}(\sqrt{\log n \log\log n})$ [DEG+21]
$\log n + \mathcal{O}(\sqrt{\log n})$, small improvement of the above in [GJ22]

Adjacency in trees
$2 \log n$ (trivial)
$\log n + \mathcal{O}(\log\log n)$ [Chu90]
$\log n + \mathcal{O}(\log^* n)$ [AR02b]
$\log n + \mathcal{O}(1)$ [ADK15]

Ancestry in trees
$2 \log n$ (trivial)
$1.5 \log n + \mathcal{O}(\log\log n)$ [AAK+06]
$\log n + \mathcal{O}(\log n \cdot \log\log\log n / \log\log n)$ [TZ01]
$\log n + \mathcal{O}(\sqrt{\log n})$ [AR02a]
$\log n + 4 \log\log n + \mathcal{O}(1)$ [FK10b]
$\log n + 2 \log\log n + 3$ [DKR15]

Routing in trees, the designer-port model
$\mathcal{O}(\log n)$ [FG01]
$\log n + \mathcal{O}(\log n \cdot \log\log\log n / \log\log n)$ [TZ01]
$\log n + \mathcal{O}((\log\log n)^2)$ [GJL21]

Table 1.4: Historical note on progress in improving selected labeling schemes.

allowed us to separate the difficulty of these two functions in our model, which was not possible with just first-order terms.

Finally, focus of labeling schemes is much more on memory than computation complexity, and memory is usually more scarce resource, both in theory and practice. There is a very rich literature on *succinct* data structures, which match information-theoretic memory lower bounds for multiple problems, while there is less interest in improving running times of algorithms from, say, $3n$ to $1.7n$.

Generalised approach to labeling. There is also a branch of research on generalised approach, focusing on broad graph families, not specific classes. This comes with different techniques and flavour, although definitions remain the same. Arguably this started with a question about implicit representation of graphs. We say that a graph family is hereditary if an induced subgraph of any graph in the family is also in the family. Considering hereditary

families is necessary due to concentration anomalies, for example, subgraphs of sparse graphs are not necessarily sparse. Recall that we have adjacency labeling for general graphs with the labels using $n/2+\mathcal{O}(1)$ bits, but for trees $\log n+\mathcal{O}(1)$ bits is sufficient, so a much smaller value. Kannan et al. [KNR92] already wondered about whether all sufficiently small graph families admit such an efficient representation. Let $\mathcal{G}_n$ be the set of n-vertex graphs from the family $\mathcal{G}$. If for all n we have $|\mathcal{G}_n| \leq 2^{\mathcal{O}(n \log n)}$, then we say $\mathcal{G}$ have at most factorial speed of growth. Spinrad [Spi03] stated the following conjecture:

Conjecture 1.2. *(The Implicit Graph Conjecture) Every hereditary family of graphs of at most factorial speed has an adjacency labeling scheme of size* $\mathcal{O}(\log n)$.

It was unresolved for almost two decades, but recently disproved by a wide margin in a work of Hatami and Hatami [HH22], which shows the existence of a hereditary family of graphs of required size but for which the labels of size $\Omega(n^{1/2-\epsilon})$ are necessary. An even weaker take on the conjecture was also disproved by Bonnet, Duron, Sylvester, Zamaraev, and Zhukovskii [BDS+24a].

This prompted research into some other more detailed cases, for example, Bonamy, Esperet, Groenland, and Scott proved [BEGS21] that every hereditary class of graphs $\mathcal{G}$ containing $2^{\Omega(n^2)}$ graphs (so close to the maximum) have asymptotically optimal adjacency labeling scheme of size $(\log |\mathcal{G}_n|)/n + o(n)$. In [BDS+24b], authors consider monotone (closed under taking not necessarily induced subgraphs) classes, and show tight bounds for adjacency labelings for many subcases. See the discussion there for a general overview of results depending on the growth functions of a given class. Bonnet, Duron, Sylvester, and Zamaraev [BDSZ24] proved efficient adjacency labeling schemes for small hereditary graph classes, containing at most $n!c^n$ n-vertex graphs, although results are not yet tight.

1.3 LIS and Dynamic Algorithms

Computing the length of a longest increasing subsequence (LIS) is one of the basic algorithmic problems. Given a sequence $(a_1, \ldots, a_n)$ with a linear order on the elements, an increasing subsequence is a sequence of indices $1 \leq i_1 < i_2 < \ldots < i_\ell \leq n$ such that $a_{i_1} < a_{i_2} < \ldots < a_{i_\ell}$. We seek the largest ℓ for which such a sequence exists. It is well known that ℓ can be computed in $\mathcal{O}(n \log n)$ time using either dynamic programming and an appropriate data structure or by computing the first row of the Young tableaux (see e.g. [Fre75], we will refer to this procedure as Fredman's algorithm even though Fredman himself attributes it to Knuth). The latter method admits an elegant formulation as a card game called patience sorting, see [AD99]. For comparison-based algorithms, a lower bound of $\Theta(n \log n)$ was shown by Fredman [Fre75], and later modified to work in the more powerful algebraic decision tree model by Ramanan [Ram97]. However, under the natural assumption that the elements in the array belong to $[n]$, an $\mathcal{O}(n \log \log n)$ time Word RAM algorithm can be obtained using a faster data structure such as van Emde Boas trees [vEB77]. This solution has been further refined to work in $\mathcal{O}(n \log \log k)$ time, where k is the answer, by Crochemore and Porat [CP10].

Dynamic algorithms. Even linear (or almost-linear) algorithms can be too slow to run repeatedly when dealing with large data. This is particularly relevant when we need to query frequently such data that undergoes continuous updates. One possible approach is to design a dynamic algorithm capable of maintaining the answer during such modifications. The updates could be simply appending new items, or inserting/substituting/deleting an arbitrary item. We note that LIS problem itself is not connected to the labeling schemes, although dynamic schemes were sometimes considered and designed. Nevertheless, as labelings can be used to represent huge networks in a decentralised way and efficiently provide some information about them on a distributed basis, dynamic algorithms are similarly important in dealing with massive data.

There exists a long line of work on dynamic algorithms for graph problems, in which the updates consist of adding or removing edges. Here, *fully dynamic* means handling edges deletions and insertions at the same time, as opposed to only decremental or only incremental changes. Examples of questions considered in this setting include fully dynamic maximal independent set [AOSS18, AOSS19, BDH+19], fully dynamic minimum spanning forest [NSW17, Wul17, NS17, HdLT01], fully dynamic matching [BCH20, BHR19, BS16], fully dynamic APSP [GW20c], incremental/decremental reachability and single-source shortest paths [BPW19, GWW20, GW20a, GW20b], or fully dynamic Steiner tree [ŁOP+15]. We stress that many of these papers resort to maintaining an approximate solution, and that in most cases the goal is to achieve polylogarithmic update/query time.

While by now we have nontrivial and surprising dynamic algorithms for many problems, in some cases even allowing randomisation and amortisation does not seem to help. Abboud and Dahlgaard [AD16] showed how to use the popular conjectures to provide an evidence that, for some problems, subpolynomial update/query solutions are unlikely, even for very restricted classes of graphs. [HKNS15] introduced a new conjecture tailored to showing that, for multiple natural dynamic problems, subpolynomial update/query time would be surprising. It has been later shown that the online Boolean matrix-vector problem considered in this conjecture does admit a very efficient cell probe algorithm [LW17, CKL18], so one cannot hope to prove it by purely information-theoretical methods. Therefore, it seems that by now we have both a number of nontrivial algorithms and some tools for proving that, in some cases, such an algorithm would be very surprising.

Fredman's algorithm can be used to maintain the length of LIS under appending elements to the sequence in $\mathcal{O}(\log n)$ time (or, by symmetry, prepending). However, it is already unclear if we can support both appending and prepending elements at the same time complexity. Chen, Chu and Pinsker [CCP13] considered the more general question of maintaining the length of LIS under insertions and deletions of elements in a sequence of length n, and showed how to implement both operations in $\mathcal{O}(\ell \log(n/\ell))$ time, where ℓ is the current length of LIS. In the worst case, this could be linear in n, so only slightly better than recomputing from scratch. In a recent breakthrough, Mitzenmacher and Seddighin [MS20a] overcame this obstacle by relaxing the problem and maintaining an approximation of LIS. For any constant $\epsilon > 0$, their algorithm maintains an $\mathcal{O}((1/\epsilon)^{\mathcal{O}(1/\epsilon)})$-approximation of LIS in $\tilde{\mathcal{O}}(n^{\epsilon})$ time per an insertion or deletion. Their solution is of course a nontrivial improvement on the worst-case $\Theta(n)$ time complexity,

but comes at the expense of returning an approximate solution. This also brings the challenge of determining if we can improve the update time to polylogarithmic (which seems to be the natural complexity for a dynamic algorithm), and determining the dependency on ϵ. Recently, Kociumaka and Seddighin [KS21] presented the first exact fully dynamic LIS algorithm with sublinear update time $\tilde{\mathcal{O}}(n^{2/3})$.

Different models of computation. In the streaming model, it is usual to mostly focus on the working space of an algorithm instead of its running time. The distance to monotonicity (DTM) of a sequence is the minimum number of edit operations required to make it sorted. This is easily seen to be the length of the sequence minus the length of its LIS. In the streaming model, computing both LIS and DTM requires $\Omega(n)$ bits of space, so it is natural to resort to approximation algorithms. For DTM, [GJKK07] gave a randomised $(4+\epsilon)$-approximation working in $\mathcal{O}(\log^2 n)$ bits of space, and [SS13] improved this to $(1+\epsilon)$-approximation in $\mathcal{O}(1/\epsilon \cdot \log^2 n)$ bits of space. [GJKK07] also provided a deterministic $(1+\epsilon)$-approximation in $\mathcal{O}(\sqrt{n})$ space, and [EJ08] gave a deterministic $(2+o(1))$-approximation in polylogarithmic space. Finally, [NS15] provided a deterministic $(1+\epsilon)$-approximation in polylogarithmic space. For LIS, [GJKK07] provided a deterministic $(1+\epsilon)$-approximation in $\mathcal{O}(\sqrt{n})$ space, and this was later proved to be essentially the best possible [EJ08, GG10].

In the property testing model, [SS17] showed how to approximate LIS to within an additive error of $\epsilon \cdot n$, for an arbitrary $\epsilon \in (0,1)$, with $\tilde{\mathcal{O}}((1/\epsilon)^{1/\epsilon})$ queries. Denoting the length of LIS by ℓ, [RSSS19] designed a nonadaptive $\mathcal{O}(\lambda^{-3})$-approximation algorithm, where $\lambda = \ell/n$, with $\tilde{\mathcal{O}}(\lambda^{-7}\sqrt{n})$ queries (and also obtained different tradeoffs between the dependency on λ and n). Recently, [NV20] proved that adaptivity is essential in obtaining polylogarithmic query complexity (with the exponent independent of ϵ) for this problem.

In the read-only random access model, [KOO+18] showed how to find the length of LIS in $\mathcal{O}(n^2/s \cdot \log n)$ time and only $\mathcal{O}(s)$ space, for any parameter $\sqrt{n} \le s \le n$. Investigating the time complexity for smaller values of s remains an intriguing open problem.

Related work. LIS can be seen as a special case of the longest common subsequence. LCS is another fundamental algorithmic problem, and it has received significant attention. A textbook dynamic programming solution allows calculating LCS of two sequences of length n in $\mathcal{O}(n^2)$ time, and the so-called "Four Russians" technique brings this down to $\mathcal{O}(n^2/\log^2 n)$ for constant alphabets [MP80] and $\mathcal{O}(n^2(\log\log n)^2/\log^2 n)$ [BF08] or even $\mathcal{O}(n^2 \log\log n/\log^2 n)$ [Gra16] for general alphabets. Recently, there was some progress in explaining why a strongly subquadratic $\mathcal{O}(n^{2-\epsilon})$ time algorithm is unlikely [ABW15, BK15], and in fact even achieving $\mathcal{O}(n^2/\log^{7+\epsilon} n)$ would have some exciting unexpected consequences [AB18]. This in particular implies that one cannot hope for a strongly sublinear time dynamic algorithm, even if only appending letters to one of the strings is allowed (unless the Strongly Exponential Time Hypothesis is false). Very recently, Charalampopoulos, Kociumaka, and Mozes [CKM20] matched this conditional lower bound, providing a fully dynamic algorithm with $\tilde{\mathcal{O}}(n)$ update time.

1.4 Results

All chapters beyond the first two are based on published papers, some of them further modified or extended. Here I shortly describe these results in the proper context.

1.4.1 Routing in Trees

Chapter 3 is a modified SODA 2021 paper *Shorter Labels for Routing in Trees* [GJL21], joint work with Paweł Gawrychowski and Jakub Łopuszański. It reuses methods from introductory Chapter 2, and is extended by a description of the decoder working in constant time.

A fundamental challenge related to networks is that of designing efficient routing mechanisms that are able to transfer information between any two nodes of the network along the shortest (or, at least, a reasonably short) path. A routing scheme formalises this as follows. Each node of the network receive packets of information and needs to decide whether they have already reached their destination or should be forwarded to one of its neighbours. This decision is made based on a header attached to the packet and a local routing table. The edges outgoing from each node have their associated port numbers and, if the decision is to forward the packet, the node needs to determine a port number and forward the packet along the corresponding edge. The stretch of a routing scheme is the maximum ratio between the length of a path determined by the scheme and the length of the shortest path. In most routing schemes the header is simply the label of the destination node chosen by the designer of the network.

In this model, Cowen designed a stretch-3 routing scheme with headers of size $\mathcal{O}(\log n)$ and local routing tables of size $\tilde{\mathcal{O}}(n^{2/3})$ [Cow01]. Eilam et al. [EGP03] decreased the size of the local routing tables to $\tilde{\mathcal{O}}(n^{1/2})$ at the expense of increasing the stretch to 5. Then, in their seminal paper Thorup and Zwick [TZ01] designed a stretch-3 routing scheme with headers of size $(1+o(1))\log n$ and routing tables of size $\tilde{\mathcal{O}}(n^{1/2})$ (essentially optimal for such a stretch). An important technical ingredient introduced in their paper is a routing scheme for trees of unbounded degree that assigns a label of length $(1+o(1))\log n$ bits to every node in such a way that, given only the labels of a source node and a destination node, it is possible to determine in constant time the port number of the first edge on the unique path from the source to the destination. According to the authors, this ingredient "may be of independent practical and theoretical interests". Indeed, follow-up works on this topic also reduce routing in a general graph to routing in (multiple) trees [Che13].

Routing labeling schemes in trees. There are two models for routing labeling schemes in trees: fixed-port and designer-port. In both versions, each undirected edge is replaced with two directed edges and, for every node u of degree $\mathsf{deg}(u)$, the edges outgoing from u have their assigned port numbers that form a permutation of $\{1,\ldots,\mathsf{deg}(u)\}$. Given the labels of u and w we should output the port number corresponding to the first edge on the path from u to w. The difference between the models is that in the fixed-port model, the port numbers are predetermined and cannot be modified, while in the designer-port model, we are free to select

them while assigning the labels, as long as the port numbers assigned to the edges outgoing from u form a permutation of $\{1, \ldots, \mathsf{deg}(u)\}$. Intuitively, the additional degree of freedom could allow for shorter labels. For trees on n nodes it is known that the labels of length $\mathcal{O}(\log^2 n/\log\log n)$ are enough in the fixed-port model [FG01], and this is asymptotically tight [FG02]. In the designer-port model, Fraigniaud and Gavoille [FG01] showed that the labels of length $4.8\log n$ are enough. Then, Thorup and Zwick [TZ01] achieved the labels of length $\log n + \mathcal{O}(\log n \cdot \log\log\log n/\log\log n)$. For the designer-port model the only known lower bound of $\log n + \Omega(\log\log n)$ bits follows by a simple reduction from ancestry labeling[1], for which an upper bound of $\log n + \mathcal{O}(\log\log n)$ does exist [FK10b], thus such a reduction cannot result in a better lower bound.

Our results. In the designer-port model, we construct a labeling scheme for routing in trees on n nodes with the labels of length $\log n + \mathcal{O}((\log\log n)^2)$. Thus, we make significant progress in determining the correct second-order term, exponentially improving on the previous results. Furthermore, our scheme is canonical, meaning that the ports are assigned in the natural order corresponding to the sizes of the subtrees. Additionally, we describe how our schemes work for trees of bounded degree or depth, and present some trade-offs if local tables at nodes are allowed to be larger than single headers.

The decoder works in polylogarithmic time. In the appendix, we modify the scheme to allow for computing the port number in constant time at the expense of slightly increasing the length to $\log n + \mathcal{O}((\log\log n)^3)$, assuming the standard word RAM model.

1.4.2 Adjacency in Planar Graphs

Chapter 4 is a modified SOSA 2022 paper *Simpler Adjacency Labeling for Planar Graphs with B-Trees* [GJ22], written with Paweł Gawrychowski. It is extended to also provide sparse induced-universal graphs for planar graphs.

Adjacency in planar graphs. We already stated some results, but let us review them in more detail. Muller [Mul88] 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. Kannan, Naor, and Rudich [KNR92] 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 [ADK15] gives us a scheme for planar graphs with the labels of length $3\log n + \mathcal{O}(1)$. Furthermore, Gavoille and Labourel [GL07] used the fact that any planar graph can be edge-partitioned into two outerplanar graphs (which have constant treewidth) to design a $(2 + o(1))\log n$-bit labeling. Recently, Bonamy, Gavoille,

[1] [Rot16] explicitly mentions that such a lower bound holds, but does not describe the (simple) necessary modification. For completeness, we provide a self-contained description in Section 3.7.

and Pilipczuk [BGP20] were able to leverage another structural theorem for planar graphs, the product structure theorem [DJM+20], to give a $(4/3 + o(1)) \log n$-bit labeling scheme.

Finally, Dujmović, Esperet, Gavoille, Joret, Micek, and Morin [DEG+21] proved the following:

Theorem 1.3 (from [DEG+21]). *There is an adjacency labeling scheme for planar graphs on n vertices with the 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 [AKTZ15, ADK15, FK10b]. 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 [DEG+21] 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 [DEG+21] is designing a 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-vertex path changes.

Our contribution to improving this result 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 result is that there is an adjacency labeling scheme for planar graphs on n vertices with the labels of length $\log n + \mathcal{O}(\sqrt{\log n})$ and the decoder working in constant time.

Universal graphs. By the standard connection between adjacency labeling schemes and induced-universal graphs we also have that:

Corollary 1.4. *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 contain every planar graph on n vertices as an induced subgraph.*

Beyond minimizing the number of vertices in a universal graph, sometimes an additional goal is to reduce the number of edges, as done for example in [AC07, AA02] for the case of graphs of bounded degree. As stated in Corollary 4.2, the adjacency labeling from [DEG+21] leads directly to an induced-universal graph with $n^{1+o(1)}$ vertices, but that graph can have n^2 or more edges. Very recently, Esperet, Joret, and Morin [EJM20] further refined the labeling scheme to obtain an induced-universal graph on just $n^{1+o(1)}$ edges:

Theorem 1.5. *(from [EJM20]) For every n, there is a universal graph U_n with $n^{1+o(1)}$ vertices and edges that contains every planar graph on n vertices as an induced subgraph.*

It turns out that also there using B-trees instead of Bulk Trees might have some benefits: their methods can be applied to our modified scheme without significant change, and the proof becomes simpler as we do not need to use additional properties of graphs of bounded treewidth. This is presented in the last section of the chapter.

1.4.3 Distance in Permutation Graphs

Chapter 5 is a preprint *Optimal Distance Labeling for Permutation Graphs* [GJ24], joint work with Paweł Gawrychowski. It describes an optimal labeling scheme for distance in permutation graphs.

Permutation graph G_π is a graph with vertices representing elements of permutation π, where there is an edge between two vertices if and only if the elements they represent form an inversion in the permutation. See Figure 1.4. In [MS99] McConnell and Spinrad show that it is possible to test in linear time whether a given graph is a permutation graph, and also construct the corresponding permutation.

A geometric intersection graph is a graph where each vertex corresponds to an object in the plane, and two such vertices are adjacent when their corresponding objects have non-empty intersection. Usually, one puts some restrictions on the objects, for example, they should be unit disks. The motivation for such a setup is twofold. First, it allows for modeling many practical problems. Second, it leads to nice combinatorial questions. This is a large research area, and multiple books/surveys are available [MM99, Pal13, Ell22, HK01], to name just a few.

In this chapter, we are interested in one of the most basic classes of geometric intersection graphs, namely permutation graphs. A permutation graph is the intersection graph of a set of segments between two parallel lines. An alternative (and more formal) definition is as follows. A graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is a permutation graph if there exists a permutation π on n elements, such that u and v are adjacent exactly when $u < v$ but $\pi(u) > \pi(v)$.

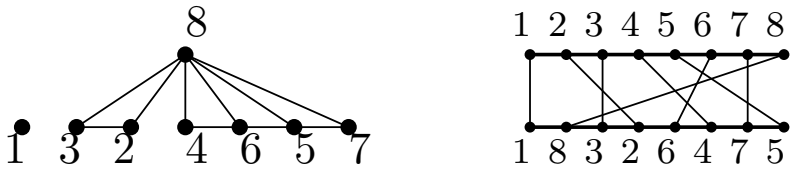


Figure 1.4: On the left there is a permutation graph described by $\pi = 18326475$. To the right we have its representation as an intersection graph of line segments whose endpoints lie on two parallel lines.

Permutation graphs admit a few alternative definitions. For example, G is a permutation graph if and only if both G and its complement are comparability graphs [EPL72]. Alternatively, they can be defined as comparability graphs of two-dimensional posets [BFR72]. Permutation graphs can be recognised in linear time [MS99], and multiple problems that are computationally difficult on general graphs admit efficient algorithms on permutation graphs [Möh85, CHL00, Col81]. In the context of labeling schemes, we seek natural classes of graphs which admits

distance labeling schemes of size $\mathcal{O}(\log n)$. In Chapter 5, we consider constructing a labeling scheme capable of efficiently reporting the distance between two given vertices of a permutation graph.

Distance labeling for permutation graphs. Katz, Katz and Peleg [KKP05] presented distance labeling scheme of size $\mathcal{O}(\log^2 n)$ for interval and permutation graphs. This was improved by Gavoille and Paul to $5 \log n$ labeling for interval graphs [GP08], with a lower bound of $3 \log n - \mathcal{O}(\log \log n)$. Very recently, He and Wu [HW24] presented tight $3 \log n + \mathcal{O}(\log \log n)$ distance labeling for interval graphs. For connected permutation graphs, Bazzaro and Gavoille in [BG05] showed a distance labeling scheme of size $9 \log n + \mathcal{O}(1)$ bits, and a lower bound of $3 \log n - \mathcal{O}(\log \log n)$. As noted in their work, this is especially interesting as there are very few hereditary graph classes that admit distance labeling schemes of size $o(\log^2 n)$. Our main result closes the gap between the lower and upper bounds on the size of distance labeling for permutation graph, by improving the upper bound to $3 \log n + \mathcal{O}(\log \log n)$.

Related works. The challenge of designing labeling schemes with short labels is related to that of designing succinct data structures, where we want to store the whole information about the input (say, a graph) using very few bits, ideally at most the information theoretical minimum. This is a rather large research area, and we only briefly describe the recent results on succinct data structures for the interval and permutation graphs. Tsakalidis, Wild, and Zamaraev [TWZ23] described a structure using only $n \log n + o(n \log n)$ bits (which is optimal) capable of answering many types of queries for permutation graphs. They also introduce the concept of semi-distributed representation, showing that for distances in permutation graphs it is possible to store global array of size $\mathcal{O}(n)$ bits and the labels on only $2 \log n$ bits, offering a mixed approach which can overcome $3 \log n$ lower bound for distance labeling. For interval graphs, a structure using $n \log n + \mathcal{O}(n)$ bits (which is again optimal) is known ([ACJS21] and [HMN+20]).

1.4.4 Dynamic Longest Increasing Subsequence

Chapter 6 is a STOC 2021 paper *Fully Dynamic Approximation of LIS in Polylogarithmic Time* [GJ21b], written with Paweł Gawrychowski. It is extended with the content of a preprint *Conditional Lower Bounds for Variants of Dynamic LIS* [GJ21a].

Our results. We consider maintaining an approximation of LIS under insertions and deletions of elements anywhere in a sequence. Let us denote the current sequence by $(a_1, a_2, \ldots, a_n)$. Then an insertion of an element x at position i transforms the sequence into $(a_1, \ldots, a_{i-1}, x, a_i, \ldots, a_n)$, while a deletion of an element at position i transforms it into $(a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$. Denoting by ℓ the length of LIS of the current sequence, we seek an algorithm that returns its $(1+\epsilon)$-approximation, that is, a number from $[\ell/(1+\epsilon), \ell]$. Our main result is that for any $\epsilon > 0$, there is a fully dynamic algorithm maintaining an $(1+\epsilon)$-approximation of LIS with insertions and deletions working in $\mathcal{O}(\epsilon^{-5} \log^{11} n)$ worst-case time.

In fact, our algorithm allows for more general queries, namely approximating LIS of any continuous subsequence $(a_i, \ldots, a_j)$, in $\mathcal{O}(\log^2 n)$ time. Furthermore, if the returned approximation is k, then in time $\mathcal{O}(k)$ the algorithm can also provide an increasing subsequence of length k. Finally, the algorithm can be initialised with a sequence of length n in time $\mathcal{O}(n\epsilon^{-2}\log^6 n)$.

The time complexities of our algorithm should be compared with that of Mitzenmacher and Seddighin [MS20a], who introduced an $\mathcal{O}((1/\epsilon)^{\mathcal{O}(1/\epsilon)})$-approximation in $\tilde{\mathcal{O}}(n^\epsilon)$ worst-case time per an insertion or deletion; this result was later improved to $(1-o(1))$-approximation in $\tilde{\mathcal{O}}(n^{o(1)})$ time [KS21]. In this context, we provide an exponential improvement: instead of providing a constant approximation in $\mathcal{O}(n^\epsilon)$ time per update, for an arbitrarily small but constant $\epsilon > 0$, we are able to provide $(1+\epsilon)$-approximation in polylogarithmic time. This is obtained by introducing a new tool, called LIS sparsification, and taking a different approach than the one based on grid packing described by Mitzenmacher and Seddighin. As for fully dynamic *exact* LIS, Kociumaka and Seddighin [KS21] presented the first algorithm with sublinear update time $\tilde{\mathcal{O}}(n^{2/3})$.

Erdős-Szekeres partitioning. The well-known theorem of Erdős and Szekeres states that any sequence consisting of distinct elements with length at least $(r-1)(s-1)+1$ contains a monotonically increasing subsequence of length r or a monotonically decreasing subsequence of length s [ES35]. In particular, any sequence consisting of n distinct elements contains a monotonically increasing or decreasing subsequence of length $\sqrt{n}$, and as a consequence, any such permutation can be partitioned into $\mathcal{O}(\sqrt{n})$ monotone subsequences (it is easy to see that this bound is asymptotically tight). The algorithmic problem of partitioning such a sequence into $\mathcal{O}(\sqrt{n})$ monotone subsequences is known as Erdős-Szekeres partitioning. A straightforward application of Fredman's algorithm gives an $\mathcal{O}(n^{1.5}\log n)$ time solution for this problem, and this has been improved to $\mathcal{O}(n^{1.5})$ by Bar-Yehuda and Fogel [BF98].

However, in the (nonuniform) decision tree model we have a trivial solution that uses $\mathcal{O}(n\log n)$ comparisons: we only need to identify the sorting permutation and then no further comparisons are required. In their breakthrough paper on the decision tree complexity of 3SUM, Grønlund and Pettie [GP18] mention Erdős-Szekeres partitioning as one of the natural examples of a problem with a large gap between the (nonuniform) decision tree complexity and the (uniform) algorithmic complexity. Another examples include 3SUM, for which the decision tree complexity was first decreased to $\mathcal{O}(n^{1.5}\sqrt{\log n})$ [GP18] and then further to $\mathcal{O}(n\log^2 n)$ [KLM19], and APSP, for which the decision tree complexity is known to be $\mathcal{O}(n^{2.5})$ [Fre76].

Closing the gap between the decision and the algorithmic complexity of Erdős-Szekeres partitioning is interesting not only as an intriguing puzzle, but also due to its potential applications. Namely, we hope to use it as a preprocessing step and achieve a speedup by operating on the obtained monotone subsequences. Recently, Grandoni, Italiano, Łukasiewicz, Parotsidis, and Uznański [GIu+21] successfully applied such a strategy to design a faster solution for the all-pairs LCA problem. The crux of their approach is a preprocessing step that partitions a given poset into $\mathcal{O}(\ell)$ chains and $\mathcal{O}(n/\ell)$ antichains, for a given parameter ℓ, in $\mathcal{O}(n^2)$ time. Of course, this can be directly applied to partition a sequence into $\mathcal{O}(\sqrt{n})$ monotone subsequences by setting

$\ell = \sqrt{n}$, but this does not constitute an improvement in the time complexity, and it is not clear whether similar techniques can help here. However, we can apply the recent result of Mitzenmacher and Seddighin [MS20a] to obtain such a partition in $\mathcal{O}(n^{1+\epsilon})$ time, for any constant $\epsilon > 0$, as follows.

We maintain a constant approximation of LIS in $\mathcal{O}(n^{\epsilon})$ time per deletion. As long as the approximated length of LIS is at least $\sqrt{n}$, we extract the corresponding increasing subsequence and delete all of its elements. This takes $\mathcal{O}(n^{1+\epsilon})$ time overall and creates no more than $\sqrt{n}$ increasing subsequences. When the approximated length drops below $\sqrt{n}$, we know that the exact length is $\mathcal{O}(\sqrt{n})$. A byproduct of Fredman's algorithm for computing the length k of LIS is a partition of the elements into k decreasing subsequences. Thus, by spending additional $\mathcal{O}(n \log n)$ time we obtain the desired partition into $\mathcal{O}(\sqrt{n})$ monotone subsequences in $\mathcal{O}(n^{1+\epsilon})$ total time. Plugging in our algorithm for maintaining $(1+\epsilon)$-approximation of LIS in $\tilde{\mathcal{O}}(\epsilon^{-5})$ time per deletion with, say, $\epsilon = 1$, we significantly improve this time complexity to $\tilde{\mathcal{O}}(n)$.

Parallel and independent work. Shortly after a preliminary version of our paper appeared on arXiv, two other relevant papers were made public. First, Seddighin and Mitzenmacher [MS20b] independently observed that their approximation algorithm can be applied to obtain Erdős-Szekeres partition (in particular, they provide a detailed description of how to modify their solution to extract the elements of LIS). Second, Kociumaka and Seddighin [KS21] modified the grid packing technique to obtain an algorithm with update time $\mathcal{O}(n^{o(1)})$ and approximation factor $1 + o(1)$. While their modification allows for more general queries, it is not able to provide a constant approximation in polylogarithmic time.

LIS variants. In the variant of LIS with 1D queries, the algorithm needs to answer queries about the length of LIS in a subarray $(a_i, a_{i+1}, \ldots, a_j)$, given any i, j. For static LIS, it is known how to build a structure of size $\mathcal{O}(n \log n)$ capable of providing exact answers to such queries in $\mathcal{O}(\log n)$ time using the so-called unit-Monge matrices [Tis07, Chapter 8].

In the weighted variant of LIS, elements have assigned nonnegative weights and we seek an increasing subsequence with the maximum sum of weights of elements. Note that in a dynamic weighted LIS changing the weight of an element can be simply done by first deleting and then inserting a new element. For the static LIS, the weighted case is also solvable in $\mathcal{O}(n \log n)$ time with a simple algorithm.

Any algorithm for weighted LIS can easily handle 1D queries. Let M be the maximum weight of any element of the array of n elements. Then to answer (i, j) 1D query, we can insert immediately to the left of a_i an element with weight $M(n+1)$ and being minimum in the linear order of elements, and insert immediately to the right of a_j an element with weight $M(n+1)$ and being maximum in the linear order of elements. The global maximum weight of an increasing subsequence in this modified array is then the weight of an increasing subsequence in the original array on elements between a_i and a_j only, increased by $2M(n+1)$.

Lower bounds. In the second part of the chapter, we apply the construction of Abboud and Dahlgaard [AD16], originally designed for distances in planar graphs [GPPR04], to provide conditional polynomial lower bounds for dynamic LIS with 1D queries (so also for dynamic weighted LIS). The main idea is to simulate their grid gadgets by an appropriately designed set of points in the plane and arrange the points in an array in the natural left-to-right order. Then, we are able to extract the original distance by querying for the LIS in a subarray. The conditional lower bounds are based on APSP and OMv conjectures.

The recent algorithms for dynamic $(1+\epsilon)$-approximation of (global) LIS [MS20a,GJ21b,KS21] support 1D queries without any modifications. On the other hand, the recent sublinear algorithm for dynamic *exact* LIS [KS21] does not. We hope that our polynomial lower bound for this natural generalisation helps to understand why there is such a large gap between the polylogarithmic complexity time for dynamic $(1+\epsilon)$-approximate LIS [GJ21b] and $\tilde{\mathcal{O}}(n^{2/3})$ time complexity for dynamic exact LIS [KS21].

1.5 Preliminaries

We denote $\lceil \log_2 n \rceil$ as $\log n$. In many calculations, we will usually skip ceil or floor notation.

The subtree rooted at vertex $u \in T$ of tree T is denoted by T_u. $\mathsf{root}(T)$ is the root of T. If s is a binary string, $|s|$ denotes its length. For binary strings s_1, s_2 we denote their concatenation by $s_1 \circ s_2$. s^k is k copies of s concatenated together.

Our labeling schemes often use labels composed of a constant number of parts. The length of each part will be $\mathcal{O}(\log n)$, so we can organise them internally using pointers. That is, at the beginning of the label we store a constant number of pointers, each taking up to $\mathcal{O}(\log \log n)$ bits and indicating location of some fixed part inside the label.

We need to note that there are some subtleties in the definitions of labelings, which might be confusing, but most often does not matter much. For example:

- Do we require the encoder to assign unique labels to vertices of any given graph? Almost all encoders do that, in a natural way, with a few exceptions of very small labels.
- When we consider a scheme for some class $\mathcal{G}_n$ of graphs on n vertices, should it work given graphs with less than n vertices, so actually $\mathcal{G}_{\leq n}$? This is usually the case.
- Similarly, when we consider $\mathcal{G}_n$, is n known to the decoder? It is best if the decoder is not provided with such knowledge for free.

Chapter 2

Warm-up: Multi-functional Labeling for Trees

2.1 Schemes for Trees

In this chapter, we provide examples of labeling schemes and along the way present some popular methods used in constructing algorithms in this area. We will consider rooted trees and ancestry labeling queries, as well as connectivity, parenthood, and sibling relations. The purpose of this chapter is introductory, thus a reader experienced with labeling schemes might want to skip some of the details. Our approach will be based on methods of Dahlgaard, Knudsen, and Rotbart [DKR15, DKR14], and though Theorem 2.1 provides (to our knowledge) new results, they are not especially involved. Nevertheless, this should be a useful introduction to labeling schemes on trees, and methods are directly reused in a more complicated Chapter 3 on routing.

Let us consider the following functions for pairs of vertices in trees:

- Adjacency, in which given the labels of two vertices $\ell(u), \ell(v)$ the decoder should decide whether u, v are adjacent in a tree.
- Siblings, where we want to check whether u, v share the parent in a tree.
- k-distance for fixed k, in which the decoder needs to decide if the distance between u, v is at most k and if so, output this distance exactly.
- Ancestry, where we want to check wheter u is an ancestor of v.
- Connectivity, if we consider a family of forests instead of trees.

As enumerated in the previous chapter, there are labeling schemes of size $\log n + \mathcal{O}(\log \log n)$ for all these functions in trees, and even $\log n + \mathcal{O}(1)$ for adjacency. Therefore a natural question is whether there are *multi-functional* labeling schemes supporting multiple types of queries at once and also of size $\log n + \mathcal{O}(\log \log n)$, instead of trivial $\mathcal{O}(\log n)$ concatenation. This is in contrast with labeling for distances in trees, where one needs $\Omega(\log^2 n)$ bits, and so it is straightforward to

enhance distance labeling with all of the above queries while worsening only second-order terms in the lengths of the labels.

k-distance can be seen as a multi-functional scheme, as the right approach [ABR05,FGNW17] can also answer sibling and parenthood queries. In [DKR14], authors consider other combinations. In particular, they explain that it is easy to add connectivity queries to any labeling scheme, with the additional cost of $\log\log n$ bits. Then, a multi-functional labeling scheme for adjacency, siblings, and connectivity is presented, using just $\log n + 3\log\log n$ bits. Finally, in the conclusion of the paper, the authors state that extending ancestry queries (with any other query besides connectivity) remains open. In this chapter, we will describe a labeling scheme of size $\log n + \mathcal{O}(\log\log n)$ for all the mentioned queries:

Theorem 2.1. *There is an ancestry labeling scheme for rooted trees such that the label of each vertex u consists of two parts, unique ID of length $\log n + \mathcal{O}(\log\log n)$ bits, and auxiliary $\mathcal{O}(\log\log n)$ bits. The decoder can answer ancestry queries and also, given label $\ell(u)$, it can compute ID of the parent of vertex u.*

It should be clear that such a scheme allows us to answer ancestry, adjacency, siblings, and parenthood queries. By expanding auxiliary parts (storing several of them from immediate ancestors), k-distance queries can also be answered, which will be described later. Finally, if we change the family of graphs to forests, we can also deal with connectivity queries as observed in [DKR14]; this is also explained in the Chapter 5, Section 5.4.1.

2.2 Classic Ancestry Labeling

Let $\mathcal{T}$ be a family of rooted trees. For a rooted tree T, $|T|$ denotes its size (the number of nodes). This will be denoted by n for the whole input tree.

It turns out that there are effective ancestry labeling schemes for trees using only one or two DFS traversals. We will start with a very simple scheme using labels of length $2\log n$, described in one of the first papers in the field [AR02b], though this method was already known earlier. This is an approach based on interval inclusion, in which nodes are assigned intervals and for node u its interval lies inside intervals of its ancestors.

To achieve this goal, we assign nodes of a rooted tree IDs equal to their pre-order numbers in the DFS traversal of the tree; this is the first half of a label. As the second part of a label, for node u we store the maximum ID occurring in its subtree T_u. These two values together clearly take $2\log n$ bits to store. Given $\ell(u) = (u_1, u_2)$ and $\ell(w) = (w_1, w_2)$, u is a proper ancestor of w if and only if $u_1 < w_1 \leq u_2$. Here we even have $[w_1, w_2] \subset [u_1, u_2]$, but later we will relax this property. See Figure 2.1 for an example, and Algorithm 2.1 for pseudocode. We refer to the counter assigning pre-order numbers as *accumulator*.

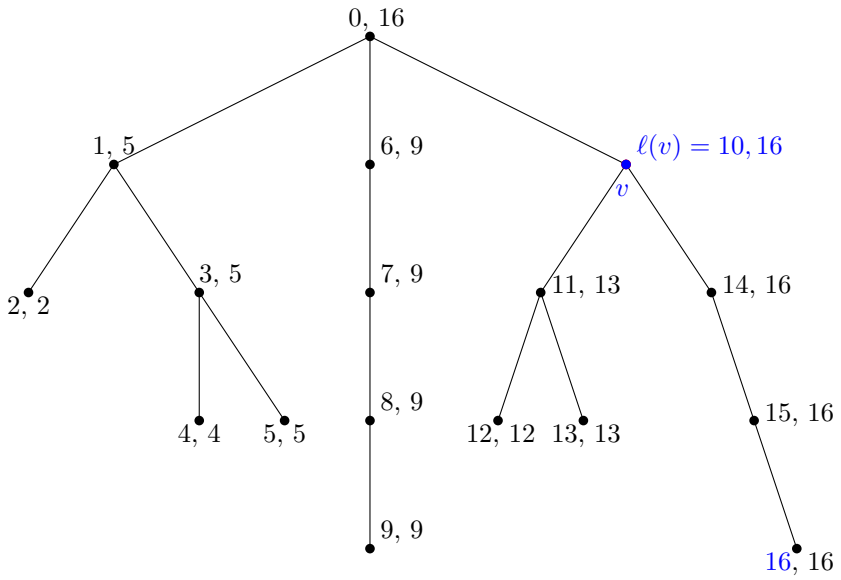


Figure 2.1: For each node, the first number is an ID equal to the pre-order number, and the second number is the maximum value occurring in its subtree. We have $\ell(v) = (10, 16)$.

Algorithm 2.1 The encoder for classic simple ancestry labeling scheme, recursively assigning IDs and subtree maximums.

1: **function** ASSIGN-IDS(u, A)
Input: node u, accumulator A. ▷ Encoder start at the root with $A = 0$
Output: $\ell(u) = (u_1, u_2)$. ▷ u_1 is preorder number, u_2 maximum in the subtree

2: $u_1 \leftarrow A$, $u_2 \leftarrow A$
3: **for** v child of u **do**
4: ASSIGN-IDS(v, $u_2 + 1$)
5: $u_2 \leftarrow v_2$ ▷ Updating the maximum

2.3 Ancestry Labeling via Rounding

As the next step, we would like to modify this classic algorithm according to the scheme by Dahlgaard, Knudsen and Rotbart [DKR15]. The main tool is reducing the number of stored bits by *rounding* numbers. Say that we are to store some values of form $\lfloor 2^{t/b} \rfloor$, where b is a fixed integer parameter and t is some nonnegative integer. For values in the range up to n, this requires just $\mathcal{O}(\log(b \cdot \log n))$ bits, instead of $\log n$, since there are only $\mathcal{O}(b \cdot \log n)$ possible values of this form. For $b = \mathcal{O}(\log n)$, the resulting size is $\mathcal{O}(\log \log n)$ bits.

We need to consider which numbers could be rounded. If we were to round the accumulator multiplicatively in each node, then a range of IDs would quickly explode, and since storing huge numbers takes many bits, this would beat the purpose of rounding. Nevertheless, we will see that for the accumulator, *additive* shifts are feasible. To facilitate rounding without extending the domain of IDs too much, we need three things. First, let us change the way intervals are

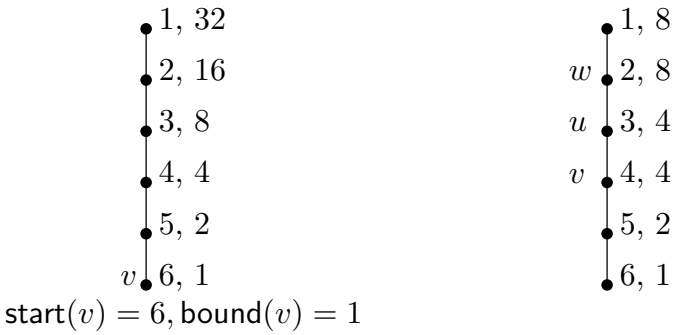


Figure 2.2: Next to nodes are noted their start, bound values, and here we assume the bound values are rounded to the powers of 2. To the left, we insist on interval inclusion; a node with ID 5 can have bound equal to 1, but then we see that bound values increase exponentially. To the right, we only require ID to be inside the ancestor's interval. For v, u we see that $4+4 > 3+4$. Moreover, the right end of the interval of w is also larger than the right end of the interval of the head of the heavy path.

stored in the labels; in the classic scheme an interval was represented simply by its beginning and end. Here, we will assign an interval of integers $\mathsf{I}(u) = [\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$ to every node $u \in T$, and $\mathsf{start}(u)$ can still be thought of as the unique ID of u and we will refer to it as such. But now the right end of the interval is not stored directly, instead it is obtained by adding the value of $\mathsf{bound}(u)$ to the ID, thus our error will depend on the range of the interval, not the accumulator itself. Still, we need to avoid the cumulative effects of multiplicative rounding as much as possible. This is where arguably the most known tree decomposition can help us, that is heavy path decomposition (or heavy-light decomposition) [ST83].

We will use the standard notion of heavy path decomposition of a tree T. Each non-leaf node $u \in T$ selects its child $\mathsf{heavy}(u)$ with the largest subtree, by the number of vertices (if there is a tie, we choose any largest subtree). All other children are called light. The edge from u to $\mathsf{heavy}(u)$ is called heavy, and all other edges outgoing from u are called light. A heavy path is a maximal path P consisting of heavy edges, with the node closest to root denoted by $\mathsf{head}(P)$. Heavy paths form a partition of the nodes of T. The light depth of a node $u \in T$ is the number of light edges on the path from u to the root and is known to be at most $\log n$. The whole decomposition (just marking the heavy child of each node) can be computed with one DFS traversal of the tree.

As the last change to the classic algorithm, we slightly modify the idea of interval inclusions. Observe the problem presented by Figure 2.2. To avoid it, we will not demand that $[\mathsf{start}(v), \mathsf{start}(v) + \mathsf{bound}(v)) \subset [\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$ for v being descendant of u, but only that $\mathsf{start}(v) \in [\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$. This requires more caution, as now some numbers (potential IDs) outside of the interval of u can be inside intervals of its children.

Summing up, we change the definition of intervals and the relation between ancestry and interval inclusion, then use rounding on ends of intervals, all on heavy path decomposition of the tree. Now, we will present the algorithm of [DKR15] in detail, though our description differ

a bit from the original. Then, we will be able to apply changes allowing us to extend it to a multi-functional scheme.

Properties. Let b be an integer parameter to be chosen later. We shall guarantee the following properties:

1. If u and w belong to different heavy paths, then $\mathsf{I}(u) \cap \mathsf{I}(w) = \emptyset$, $\mathsf{I}(u) \subset \mathsf{I}(w)$, or $\mathsf{I}(w) \subset \mathsf{I}(u)$. Moreover, $\mathsf{I}(u) \subset \mathsf{I}(w)$ iff w is an ancestor of u.
2. If u and w belong to the same heavy path and u is closer to the root of T than w, then $\mathsf{start}(w) \in \mathsf{I}(u)$.
3. $\mathsf{bound}(u) = \lfloor 2^{t/b} \rfloor$ for some nonnegative integer t, depending on u.

The first property states that for disjoint heavy paths, we still maintain exact inclusion, that is interval of the descendant is inside the interval of an ancestor. The second property says that for two nodes on the same heavy path, we require only ID to be included in ancestor's interval. From the all properties stated above we have that w is a proper ancestor of u iff $\mathsf{start}(u) \in \mathsf{I}(w) \backslash \{\mathsf{start}(w)\}$.

Encoding. The intervals are assigned with a recursive procedure that receives a havy path and the accumulator value. Consider a heavy path $P = u_1 - u_2 - \ldots - u_p$, where $\mathsf{head}(P) = u_1$. After reaching a node u_i, we set $\mathsf{start}(u_i)$ to be the current accumulator, increase the accumulator by 1, and iterate over the light children $v_{i,1}, v_{i,2}, \ldots$ of u_i. Let $\mathsf{span}(v) = \max_{v' \in T_v} \mathsf{start}(v') + \mathsf{bound}(v') - \mathsf{start}(v)$. For each light node $v_{i,j}$, we call the procedure recursively, and then increase the accumulator by $\mathsf{span}(v_{i,j})$. See Algorithm 2.2.

Algorithm 2.2 The encoder for optimal ancestry labeling scheme, recursively handling heavy paths. It is initiated with the root's heavy path and $A = 0$.

function ASSIGN-INTERVALS(P, A)
Input: heavy path $P = u_1, \ldots, u_p$, accumulator A. b is a fixed parameter.
Output: labels for every node $u \in T_{u_1}$.

  for $i = 1 \ldots p$ **do**
    $\mathsf{start}(u_i) \leftarrow A, A \leftarrow A + 1$
    for $v_{i,j}$ light child of u_i **do**
      ASSIGN-INTERVALS(P', A), where P' is a heavy path with $v_{i,j}$ as the head
      $A \leftarrow A + \mathsf{span}(v_{i,j})$
  for $i = 1 \ldots p$ **do**
    Let t be the smallest natural number such that $\lfloor 2^{t/b} \rfloor + \mathsf{start}(u_i) \geq A$
    $\mathsf{bound}(u_i) \leftarrow \lfloor 2^{t/b} \rfloor$

Finally, we need to set $\mathsf{bound}(u_i)$ for every $i = 1, 2, \ldots, p$. It can be seen that we only need that $\mathsf{start}(u_i) + \mathsf{bound}(u_i)$ is at least as large as the final accumulator, by span definition. Furthermore,

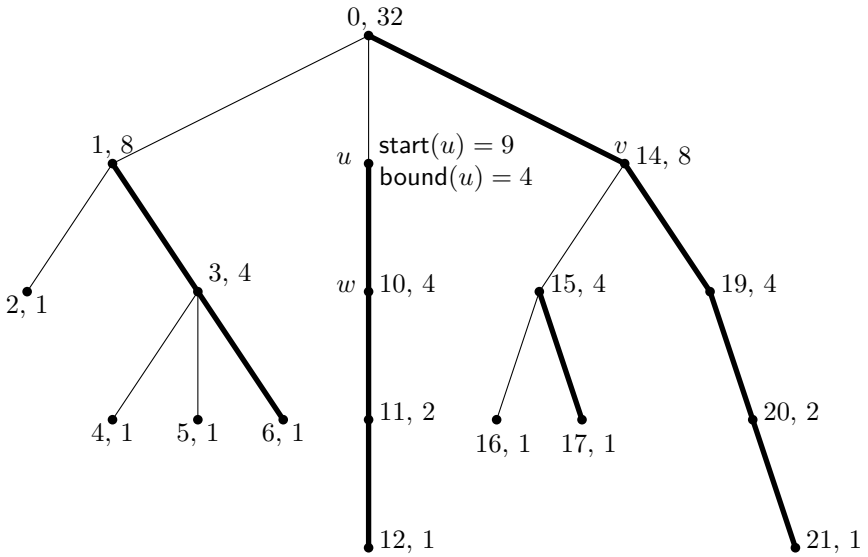


Figure 2.3: start and bound values for a given tree, with $b = 1$ for simplicity. Thicker lines represent heavy paths. Note that $\mathsf{start}(u)+\mathsf{bound}(u) = 13$ is smaller than $\mathsf{start}(w)+\mathsf{bound}(w) = 14$, even though w is a (heavy) child of u, and that $\mathsf{start}(v)$ is set to 14 due to $\mathsf{span}(u)$ being 5, not 4. span of the root is 32, even though the tree has 18 nodes, but such an increase is acceptable.

the final accumulator is $A = \mathsf{start}(u_1)+\sum_{i=1}^{p}(1+\sum_j \mathsf{span}(v_{i,j}))$, and we have $\mathsf{start}(u_i) \geq \mathsf{start}(u_1)$ for all i. Then, because possible values of $\mathsf{bound}(u_i)$ are all numbers of the form $\lfloor 2^{t/b} \rfloor$, by taking the smallest t such that $\mathsf{start}(u_i) + \mathsf{bound}(u_i) \geq A$ we have $\mathsf{start}(u_i) + \mathsf{bound}(u_i) \leq \mathsf{start}(u_i) + 2^{1/b}(A - \mathsf{start}(u_1))$. Thus, we obtain that $\mathsf{span}(u_1) \leq 2^{1/b}\sum_{i=1}^{p}(1 + \sum_j \mathsf{span}(v_{i,j}))$. Note that $2^{1/b}$ is larger than 1 and this is our multiplicative rounding effect at each heavy path. Consult Figure 2.3 for an example with assigned start and bound values.

Property 1 is maintained by line 6, where we shift by the whole span to ensure interval disjointness. Other properties hold because of the last two lines.

Analysis. By unwinding the recurrence and using the fact that the light depth of any node is at most $\log n$, we conclude that $\mathsf{span}(\mathsf{root}(T)) \leq 2^{(\log n)/b} n$, and by construction $\mathsf{start}(u) + \mathsf{bound}(u) \leq \mathsf{span}(\mathsf{root}(T))$ for any node u. Consequently, storing any $\mathsf{start}(u)$ takes only $\log n + (\log n)/b$ bits. Storing $\mathsf{bound}(u)$ requires $\mathcal{O}(\log(b \cdot \log n))$ bits. This results in an ancestry labeling scheme with labels of length $\log n + \mathcal{O}(\log \log n)$, for example by taking $b = \log n$. In fact, if we pay attention to constants, we can see that the length can be $\log n + 2 \log \log n$ (it is possible to store such two parts of the labels without additional overhead for marking where the second part begins).

2.4 moveUp Function

To prove Theorem 2.1, we would like to enhance the ancestry labeling scheme from the previous section with function $\mathsf{moveUp}(u)$, which returns ID (start value) of the parent of u, or nothing for the root. As for now, IDs were rather arbitrary numbers and there was no clear connection

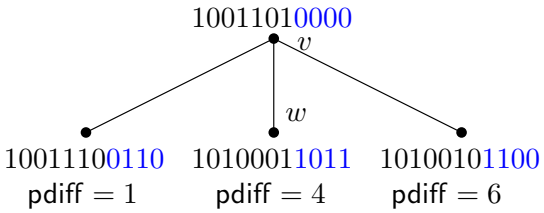


Figure 2.4: Children of v store that v has four trailing zeroes, so $\mathsf{pzero}(w) = 4$. Then, we are only interested in the difference between the remaining prefixes, thus $\mathsf{pdiff}(w) = 1010001_2 - 1001101_2 = 4$.

between IDs of v and its children, except the very first one. With more additive shifts applied to the accumulator, though, we can ensure that the parent's ID is easy to store. Observe that for any two natural numbers a, b we have in the range $[a, a + b)$ some number with at least $\log b$ (ignoring floor) trailing zeroes in binary representation, that is, a number divisible by $2^{\log b}$. Such a number is easier to store, since we do not need to store zeroes explicitly, instead storing prefix of a number and then how many zeroes need to be appended at the end. For example, 1011001000000000_2 could be stored in two parts, the prefix of seven bits and then the number of trailing zeroes (nine); we deal with labels consisting of multiple parts in a way described in preliminaries.

Let us for a moment ignore how moveUp works for heavy children. By $\mathsf{lw}(u)$ we denote the *light weight* of a node u, that is the sum of subtrees' sizes of its light children. When the encoder arrives at node u with accumulator A, it does not always have to assign $\mathsf{start}(u) = A$ as previously. Instead, we consider the range $[A, A + \mathsf{lw}(u)]$ and choose the number with the most trailing zeroes in binary representation. Intuitively, it should not increase the domain too much, as the total amount of additive shifts is no more than $n \log n$. Then, if for w a child of u we have $\mathsf{start}(w) - \mathsf{start}(u) \leq 2^k \mathsf{lw}(u)$ for not too large k, it is easy to store how to transform $\mathsf{start}(w)$ into $\mathsf{start}(u)$, that is, implement $\mathsf{moveUp}(w)$.

If u is parent of w, let $\mathsf{pzero}(w)$ denote the number of trailing zeroes in $\mathsf{start}(u)$; this information takes $\mathcal{O}(\log \log n)$ bits to store. Then denote $\mathsf{pdiff}(w) = (\mathsf{start}(w) - \mathsf{start}(u))/2^{\mathsf{pzero}(w)}$, which we later prove to also be possible to store on $\mathcal{O}(\log \log n)$ bits. To obtain $\mathsf{start}(u)$ from $\ell(w)$, first we set a suffix of $\mathsf{start}(w)$ of length $\mathsf{pzero}(w)$ to zeros. Second, we subtract $\mathsf{pdiff}(w)$ from the remaining prefix. See Figure 2.4, Algorithm 2.3 (with marked changes from the previous encoder), and Algorithm 2.4 for the decoder.

Analysis. We define the level of a node u, denoted $\mathsf{level}(u)$, to be the unique integer l such that $|T_u| \in [2^l, 2^{l+1})$. Thus, $\mathsf{level}(u) \in \{0, 1, \ldots, \log n\}$. Let u_1 be the head of its heavy path. We can show by induction on levels that $\mathsf{span}(u_1) \leq |T_{u_1}|(1 + \mathsf{level}(u_1))2^{\mathsf{level}(u_1)/b}$. Indeed, this

Algorithm 2.3 The encoder for multi-functional labeling scheme.

1: **function** ASSIGN-LABELS(P, A)
Input: heavy path $P = u_1, \ldots, u_p$, accumulator A. b is a fixed parameter.
Output: labels for every node $u \in T_{u_1}$.

2: **for** $i = 1 \ldots p$ **do**
3: Set A to number with largest number of trailing zeroes in $[A, A + \mathsf{lw}(u_i)]$
4: $\mathsf{start}(u_i) \leftarrow A, A \leftarrow A + 1$
5: **for** $j = 1 \ldots \deg(u_i) - 1$ **do** ▷ Light children of u_i
6: ASSIGN-LABELS(P', A), where P' is a heavy path with $v_{i,j}$ as the head
7: $A \leftarrow A + \mathsf{span}(v_{i,j})$
8: **for** $i = 1 \ldots p$ **do**
9: Let t be the smallest natural number such that $\lfloor 2^{t/b} \rfloor + \mathsf{start}(u_i) \geq A$
10: $\mathsf{bound}(u_i) \leftarrow \lfloor 2^{t/b} \rfloor$
11: Store as $\mathsf{pzero}(u_i)$ the number of trailing zeroes from the parent of u_i
12: Store $\mathsf{pdiff}(u_i) = (\mathsf{start}(u_i) - \mathsf{start}(v))/2^{\mathsf{pzero}(u_i)}$

holds for leaves on level 0, and then:

$$\begin{aligned}
\mathsf{span}(u_1) &\leq 2^{1/b}\Big(\sum_{i=1}^{p} 1 + \mathsf{lw}(u_i) + \sum_j \mathsf{span}(v_{i,j})\Big) \\
&\leq 2^{1/b}\Big(|T(u_1)| + \sum_{i=1}^{p}\sum_j |T(v_{i,j})| \cdot \mathsf{level}(u_1) 2^{(\mathsf{level}(u_1)-1)/b}\Big) \\
&\leq 2^{1/b}\big(|T(u_1)| + |T(u_1)| \cdot \mathsf{level}(u_1) 2^{(\mathsf{level}(u_1)-1)/b}\big) \\
&\leq |T_{u_1}|(1 + \mathsf{level}(u_1)) 2^{\mathsf{level}(u_1)/b}.
\end{aligned}$$

Let us again set $b = \log n$. It holds that $\mathsf{span}(\mathsf{root}(T)) \leq n(1 + \log n)2^{\log n/b}$, so storing any start value takes $\log n + \mathcal{O}(\log \log n)$ bits. We also see that indeed pdiff values can be stored on $\mathcal{O}(\log \log n)$ bits, as for w a light child of u it holds that $\mathsf{span}(u) \leq 2\mathsf{lw}(u) \log n$, and thus

$$\mathsf{start}(w) - \mathsf{start}(u) \leq \mathsf{lw}(u) 2^{\log \log n + 1} \leq 2^{\mathsf{pzero}(w)} 2^{\log \log n + 1}.$$

The running time of both encoder and decoder depends on how quickly we can operate on rounded numbers of the form $2^{t/b}$, but even with simple iterations over all values the encoder works in near-linear time, ancestry query works in logarithmic time, and moveUp in constant time.

Heavy children. Now we can revisit the case of heavy children. In our approach, the problem is that w heavy child of u can have arbitrarily bigger light weight, which means that after choosing $\mathsf{start}(w)$ we cannot always have small $\mathsf{pdiff}(w)$; in other words, $\mathsf{start}(w) - \mathsf{start}(u)$ can be $\Omega(n)$ while $\mathsf{pzero}(w)$ is even zero. Then $(\mathsf{start}(w) - \mathsf{start}(u))/2^{\mathsf{pzero}(w)}$ can be significantly larger than $2^{\log \log n}$.

Algorithm 2.4 Functions of the decoder: ancestry query and moveUp.

function IsAncestor($\ell(u), \ell(v)$)
Input: labels of two nodes u, v.
Output: True if u is proper ancestor of v.
    Extract start and bound values from the labels
    return $\mathsf{start}(v) \in (\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$

function moveUp($\ell(u)$)
Input: label of node u.
Output: ID of the parent of u.
    Split label into parts
    $s \leftarrow \mathsf{start}(u)$
    Set $\mathsf{pzero}(u)$ trailing bits of s to zeros
    $s \leftarrow s - \mathsf{pdiff}(u) \cdot 2^{\mathsf{pzero}(u)}$
    return s

This is not a big issue and we have in fact many ways to deal with it. For example, if $\mathsf{start}(w)$ contains a lot of trailing zeroes, we do not actually need to store them explicitly, instead we can just store their number and then use saved bits for some other information, for example parts of parent's ID. But this would make both structure of labels and decoding slightly more complicated, so instead we opt for approach modifying encoding.

We make an additional preprocessing step during the encoding phase, after computing heavy-path decomposition. For each heavy path, first we set $\mathsf{lw}'(u_j) = \mathsf{lw}(u_j)$ for all nodes on a path. Then we iteratively consider unprocessed u_j with the largest $\mathsf{lw}'(u_j)$ and set, for $d \in \{-1, 1\}$, $\mathsf{lw}'(u_{j+d}) = \max(\mathsf{lw}'(u_{j+d}), \mathsf{lw}'(u_j)/2)$. That is, we simply ensure that the light weights of two adjacent nodes on a heavy path differ by a factor of at most 2. The cost of this action is negligible, as it is easy to see that the sum of the final lw' values of all nodes is at most three times the sum of the original lw, from the geometric sequence in two directions. In other words, if we count potential increases caused by u_j for all nodes u_i with $i < j$, this is at most $\sum_k \mathsf{lw}(u_j)/2^k < \mathsf{lw}(u_j)$.

Therefore, using lw' changes only additive constant in the label size, and then w heavy child of u can be treated just as light children, since $(\mathsf{start}(w) - \mathsf{start}(u))/2^{\mathsf{pzero}(w)} = \mathcal{O}(2^{\log\log n})$.

Conclusion. As decoding is straightforward, with the above we have proved Theorem 2.1. Indeed, $\mathsf{start}(u)$ is unique ID of node u and can be stored on $\log n + \mathcal{O}(\log\log n)$ bits. Then, the auxiliary part of the label is composed of three parts, $\mathsf{bound}(u)$, $\mathsf{pzero}(u)$, and $\mathsf{pdiff}(u)$, each stored on $\log\log n$ bits. Given two labels $\ell(u), \ell(v)$, u being proper ancestor of v is equivalent with $\mathsf{start}(v) \in (\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$. Given just $\ell(u)$, using $\mathsf{start}(u), \mathsf{pzero}(u), \mathsf{pdiff}(u)$ we can compute ID (start value) of the parent of u. Now, say w is parent of u, then if in label u we store also $\mathsf{pzero}(w), \mathsf{pdiff}(w)$ using additional $\mathcal{O}(\log\log n)$ bits, we can compute ID of the grandparent (parent of w) just from the label of u, by repeating the process. Therefore, for fixed k, we can answer ancestry and k-distance queries using labels of size $\log n + \mathcal{O}(\log\log n)$ bits.

Chapter 3

Shorter Labels for Routing in Trees

In this chapter, the following main theorem is proven:

Theorem 3.1. *There exists a canonical labeling scheme for routing in trees on n nodes with labels of length $\log n + \mathcal{O}((\log\log n)^2)$ bits, the decoder answering queries in polylogarithmic time, and the encoder working in near-linear time.*

3.1 Introduction

We consider a family of rooted trees $\mathcal{T}$. A routing labeling scheme for $\mathcal{T}$ consists of two parts.

Definition 3.2. *The encoder takes a tree $T \in \mathcal{T}$ and assigns a label (binary string) $\ell(u)$ to every node $u \in T$. Additionally, every edge from u to its child is labeled with a distinct integer from $\{1, 2, \ldots, \mathsf{deg}(u)\}$, called the port number.*

Definition 3.3. *The decoder receives labels $\ell(u)$ and $\ell(w)$, such that $u, w \in T$ for some $T \in \mathcal{T}$ and $u \neq w$. If the next node on the path from u to w is the parent of u, the decoder should return 0[1]. Otherwise, it should return the port number corresponding to the first edge on the path.*

Our schemes will always create unique labels, so in fact they do not need the assumption that $u \neq w$. We are interested in minimising the maximum length of a label, that is, $\max_{T\in\mathcal{T}} \max_{u\in T} |\ell(u)|$. For convenience, we will assume that the value of $\lceil \log n \rceil$ is known to the decoder, where n is the size of the tree. This is without loss of generality, as we can simply include it in every label. To avoid clutter in the description of our schemes, we will sometimes ignore floors and ceilings.

Overview of the methods. Extending ideas from Chapter 2, we use rounding to the powers of $2^{1/b}$. For integer parameter b, any integer $x \in [0, n-1]$ can be rounded up to $x' = \lfloor 2^{t/b} \rfloor$, for the smallest integer t such that $x' \geq x$ holds. Assuming that b is known, only the value

[1] We might also allow assigning a port number to the edge from u to its parent. However, we find this formulation cleaner and sufficient for our purposes.

of t needs to be stored using only $\log(b \log n) = \log b + \log\log n$ bits instead of $\log n$. This is useful when combined with the notion of an interval-based scheme, in which we assign a range $[\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$ to every node, and then round up $\mathsf{bound}(u)$.

We follow the particularly clean version of such an approach used by Dahlgaard, Knudsen, and Rotbart [DKR15] in their ancestry labeling scheme. This allows us to decide if we should route up, but the real challenge is to route down. In Section 3.3, we overcome this difficulty by partitioning the children of u into small and big. The intuition is that the former can be rounded up more aggressively, and we can afford to store the number of children for every possible value of bound corresponding to the latter. By appropriately adjusting the parameters in this construction, we obtain a canonical routing labeling scheme with labels of length $\log n + \mathcal{O}((\log^{3/4} n)\sqrt{\log\log n})$. This is already a significant improvement on the state-of-the-art, and serves as an introduction to the better scheme presented in Section 3.4. This section introduces our main technical contribution, which is a compact encoding of the sizes of the subtrees rooted at the children, using two-step process of rounding more complex than simple division into small/big children. This results in a canonical routing labeling scheme with labels of length $\log n + \mathcal{O}(\sqrt{\log n \cdot \log\log n})$.

Finally, in Section 3.5 we combine this with an additional tool used for adjacency labeling [ADK15]: we are able to guarantee that, for some nodes u, there are many trailing zeroes in $\mathsf{start}(u)$, which allows us to hide some additional information there. This in turn makes possible using varying parameter b depending on subtrees sizes of nodes. With that we obtain a canonical routing labeling scheme with labels of length $\log n + \mathcal{O}((\log\log n)^2)$. While we are mostly concerned with determining the asymptotics of the second-order term, we observe that the decoder for this scheme can be implemented in polylogarithmic time.

3.2 Preliminaries

As in the previous chapter, we define the level of a node u, denoted $\mathsf{level}(u)$, to be the unique integer l such that $|T_u| \in [2^l, 2^{l+1})$.

Important to us is a notion of canonical assignments:

Definition 3.4. *(as in [TZ01]) We say that the assignment of port numbers to the edges of T is* canonical *if it is obtained in the following way. Let v be a node of T whose parent, if any, is v_0, and whose children arranged in non-increasing order of size are $v_1, v_2, \ldots v_i$, i.e. $|T_{v_1}| \geq |T_{v_2}| \geq \ldots \geq |T_{v_i}|$. Then, for every j, the edge from v to v_j should be assigned port j. There might be many canonical assignments if subtrees of some children are of equal sizes.*

Computational model. We assume the standard word RAM model with words consisting of $w = \Omega(\log n)$ bits. Basic arithmetical operations (addition, subtraction, bitwise operations, multiplication and division) on such words are assumed to work in constant time. A label consisting of b bits is given to the decoder packed in an array consisting of $\lceil b/w \rceil$ words.

3.3 Framework for Labeling Schemes

3.3.1 Routing in Trees of Bounded Degree

The ancestry labeling scheme from Chapter 2 extends easily to a routing labeling scheme for bounded degree trees with labels of length $\log n + \mathcal{O}(\log \log n)$. As explained in Section 3.7, this is optimal up to the asymptotics of the smaller-order term, even for very simple trees.

Algorithm 3.1 The encoder for trees of bounded degree, with marked changes from Algorithm 2.2.

function CREATE-LABELS(P, A)
Input: heavy path $P = u_1, \ldots, u_p$, accumulator A. b is a fixed parameter.
Output: labels for every node $u \in T_{u_1}$.

for $i = 1 \ldots p$ **do**
  $\mathsf{start}(u_i) \leftarrow A, A \leftarrow A + 1$
  for $v_{i,j}$ light child of u_i **do** ▷ Sorted by size
    CREATE-LABELS(P', A), where P' is a heavy path with $v_{i,j}$ as the head
    $\mathsf{rt}(u_i)[j] \leftarrow \mathsf{span}(v_{i,j})$ ▷ Routing table
    $A \leftarrow A + \mathsf{span}(v_{i,j})$
for $i = 1 \ldots p$ **do**
  Let t be the smallest natural number such that $\lfloor 2^{t/b} \rfloor + \mathsf{start}(u_i) \geq A$
  $\mathsf{bound}(u_i) \leftarrow \lfloor 2^{t/b} \rfloor$
Let t be the smallest natural number such that $\lfloor 2^{t/b} \rfloor \geq \mathsf{span}(u_1)$
$\mathsf{bound}(u_1) \leftarrow \lfloor 2^{t/b} \rfloor$ ▷ Enforce head of the path to define span

Algorithm 3.2 The decoder for trees of bounded degree.

function GET-PORT($\ell(u), \ell(w)$)
  Input: labels of u and w.
  Output: port number corresponding to the first edge on the path from u to w.

  Unpack $\mathsf{start}(w), \mathsf{start}(u), \mathsf{bound}(u), \mathsf{rt}(u)$ from the labels
  if $\mathsf{start}(w) \notin (\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$ **then**
    return 0 ▷ Not an ancestor, routing up
  $S \leftarrow 0$
  for $j = 1 \ldots \deg(u) - 1$ **do** ▷ Iterating through the routing table
    $S \leftarrow S + \mathsf{rt}(u)[j]$
    if $\mathsf{start}(w) - \mathsf{start}(u) < S$ **then return** $j + 1$
  return 1 ▷ Routing to the heavy child

Theorem 3.5. *There exists a canonical labeling scheme for routing in bounded degree trees on n nodes with labels of length $\log n + \mathcal{O}(\log \log n)$ bits.*

Proof. (sketch) We apply the ancestry scheme described in Chapter 2. Observe that for a head of a heavy path u_1, by increasing $\mathsf{bound}(u_1)$, we can make sure that $\mathsf{span}(u_1) = \mathsf{bound}(u_1)$ and so $\mathsf{span}(u_1) = \lfloor 2^{t/b} \rfloor$ for some t, meaning that it can be stored on small number of bits. Such an operation introduces rounding by at most $2^{1/b}$ for every light edge. With this property, every node u_i can save in its label a *routing table* $\mathsf{rt}(u_i)$, storing a constant number of (rounded) numbers $\mathsf{rt}(u_i)[j] = \mathsf{span}(v_{i,j})$ that represent ranges necessary for subtrees rooted in the light children of u_i. Additionally, we assign number $j+1$ to the port leading to $v_{i,j}$, and number 1 to the port leading to the heavy child. Then, the decoder can answer a query for routing from u to w in the following way. Firstly, it checks whether u is an ancestor of w, and if not port number 0 leading to the parent of u is chosen. Secondly, if $\mathsf{start}(w) - \mathsf{start}(u) \geq \sum_{j=1} \mathsf{rt}(u)[j]$, port number 1 leading to the heavy child is chosen. In the last case, the decoder finds the smallest k such that $\mathsf{start}(w) - \mathsf{start}(u) < \sum_{j=1}^{k} \mathsf{rt}(u)[j]$ holds and then port number $k+1$ is chosen. Pseudocode for the encoder is presented in Algorithm 3.1, and for the decoder in Algorithm 3.2.

Again, by a straightforward induction on the light depth of a node, it can be shown that $\mathsf{span}(u) \leq |T_u| 2^{2\mathsf{level}(u)/b}$. Using $b = \log n$, we get that the final length of a label is as stated, since $\mathsf{start}(u)$ is stored on $\log n + \mathcal{O}(1)$ bits and there is constant number of other parts, each stored on $\mathcal{O}(\log \log n)$ bits. The scheme can be made canonical by simply ordering light children by the size of their subtrees. □

In an example presented in Figure 2.3, we have $\mathsf{span}(u) = 5$, so (for $b = 1$) it would be rounded to $\mathsf{span}(u) = 8$ just by setting $\mathsf{bound}(u) = 8$. Then the accumulator would carry that shift further, and for example $\mathsf{start}(v)$ would be set to 17.

3.3.2 Preliminary Routing Scheme

The goal of this subsection will be to prove the following theorem:

Theorem 3.6. *There exists a canonical labeling scheme for routing in trees on n nodes with labels of length $\log n + \mathcal{O}((\log^{3/4} n)\sqrt{\log \log n})$.*

Although already better than previously known results, it can be seen as an extended warm-up and methods used in this subsection are not essential for the rest of the chapter. Therefore the reader comfortable enough with the framework can safely skip this part.

We start with a general overview of a routing labeling scheme. Given the value of start, we need to be able to determine the child whose subtree contains the corresponding node. With this goal in mind, we partition the light children of every node into small and big. To define the partition, we say the light child v of u is *big* when $\mathsf{level}(v) \in (\mathsf{level}(u) - c, \mathsf{level}(u))$, where c is a parameter to be chosen later. Otherwise light child v is *small*. We arrange the light children of u in non-increasing order by the sizes of their subtrees, so that we first have all big children and then all small children. The intervals of big children are processed exactly as in an ancestry scheme, but for the intervals of all small children we introduce an intermediate step.

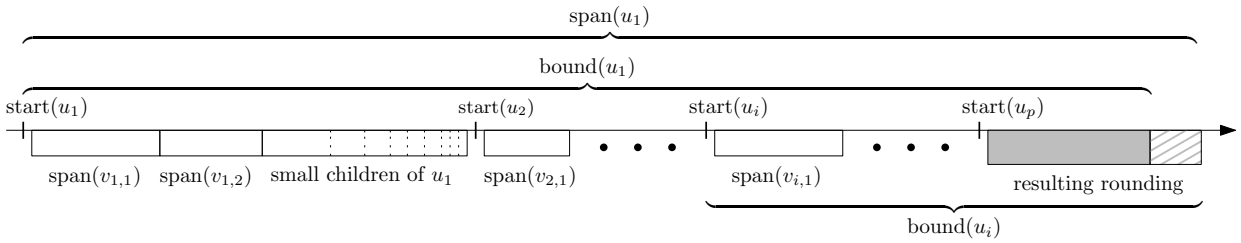


Figure 3.1: Intervals corresponding to the nodes of a single heavy path and the subtrees that hang off this path, individual big children first and then clustered small children. $\mathsf{span}(u)$ might be larger than $\mathsf{bound}(u)$, although for the head of the heavy path u_1 we will make sure that $\mathsf{span}(u_1) = 2^{t/b}$.

Observe that on any path from the root to a leaf there are at most $(\log n)/c$ small children, because every small child implies that the level decreases by at least c. This allows us to deal with small children via a costly but simple approach. For a node u_i with small children $v_{i,k+1}, v_{i,k+2}, \ldots, v_{i,k+s}$ let $\mathsf{small}(u_i) = \sum_{j=1}^{s} \mathsf{span}(v_{i,k+j})$. As this value will be stored in a label, we need to round it up to $\mathsf{small}'(u_i)$, the smallest number $x = \lfloor 2^{t/b} \rfloor$ such that $x \geq \mathsf{small}(u_i)$. Then, to handle small children of u_i, we reserve a single interval of length $\mathsf{small}'(u_i) \log(\mathsf{small}'(u_i))$. This interval is partitioned according to a harmonic sequence, so at the beginning there is a subinterval of size $\mathsf{small}'(u_i)$, then subinterval $\lfloor \mathsf{small}'(u_i)/2 \rfloor$, in general the j-th subinterval has size $\lfloor \mathsf{small}'(u_i)/j \rfloor$ for $j = 1, 2, \ldots, \mathsf{small}'(u_i)$. Clearly, due to the ordering of the children, $\mathsf{span}(v_{i,k+j})$ can fit into the j-th subinterval. The final size of the single interval reserved for storing all small children is at most $\mathsf{small}(u_i) 2^{1/b} (\log \mathsf{small}(u_i) + 1)$.

Knowing $\mathsf{small}'(u_i)$, the decoder can compute the length $\mathsf{small}'(u_i) \log(\mathsf{small}'(u_i))$ of the created interval and its exact division according to the harmonic sequence. Then, given any number from such an interval, the decoder can determine which subinterval contains this number, so also which child contains the corresponding node in its subtree. The encoder can first bound spans of small children as in the claim below, compute small' value and then recurse on the small children, with successive accumulator values shifted according to the harmonic sequence.

Consult Figure 3.1 for an illustration of these intervals on the number axis. We also make sure that $\mathsf{span}(u_1) = \lfloor 2^{t/b} \rfloor$ for some t by increasing $\mathsf{bound}(u_1)$ if necessary. This is to guarantee that there is a small number of the possible values of span for the heads of heavy paths, which will be useful later. By the same argument as before, the procedure guarantees the properties stated in Subsection 2.3, but now we need to analyze the resulting $\mathsf{span}(\mathsf{root}(T))$ more carefully.

Claim 3.7. *Let u_1 be the head of its heavy path. For n large enough, after having processed u_1 the following holds:*

$$\mathsf{span}(u_1) \leq |T_{u_1}| 2^{2\mathsf{level}(u_1)/b} (2 \log n)^{\lfloor \mathsf{level}(u_1)/c \rfloor}.$$

Proof. Observe that the above expression can be upper bounded as follows:

$$\begin{aligned}\mathsf{span}(u_1) &\leq |T_{u_1}|2^{2\mathsf{level}(u_1)/b}(2\log n)^{\lfloor \mathsf{level}(u_1)/c\rfloor}\\ &\leq n2^{2\log n/b}(2\log n)^{\log n/c}\\ &= n2^{2\log n/b+(1+\log\log n)\log n/c}.\end{aligned}$$

As long as $b = \omega(1)$ and $c = \omega(\log\log n)$, for n large enough this is at most $n^{1.5}$. We will use this crude bound on $\mathsf{span}(u)$ to simplify some of the calculations.

Claim 3.7 is proved by induction on the light depth of a node. The induction basis is a heavy path with no light children hanging off it, for which we have just $\mathsf{span}(u_1) = 1$ if $|T_{u_1}| = 1$ and $\mathsf{span}(u_1) \leq |T_{u_1}|2^{1/b}$ otherwise. Now let us consider a heavy path $P = u_1 - \ldots - u_p$ and assume that the inequality holds for the root $v_{i,j}$ of every subtree hanging off the path:

$$\mathsf{span}(v_{i,j}) \leq |T_{v_{i,j}}|2^{2\mathsf{level}(v_{i,j})/b}(2\log n)^{\lfloor \mathsf{level}(v_{i,j})/c\rfloor}.$$

Then, replacing the intervals of all small children of u_i with a single interval increases their total length by a factor of $2^{1/b}(\log \mathsf{small}(u_i) + 1)$, where for n large enough $\mathsf{small}(u_i) \leq n^{1.5}$ and $\log n^{1.5} + 1 \leq 2\log n$, so the increase is by at most a factor of $2^{1/b}2\log n$. If $v_{i,j}$ is a small child, then $\lfloor \mathsf{level}(v_{i,j})/c\rfloor \leq \lfloor \mathsf{level}(u_1)/c\rfloor - 1$, so we have:

$$\begin{aligned}&\mathsf{small}'(u_i)\log \mathsf{small}'(u_i)\\ &\leq 2^{1/b}2\log n \sum_{j:v_{i,j}\text{ is small}} \mathsf{span}(v_{i,j})\\ &\leq (2\log n)^{\lfloor \mathsf{level}(u_1)/c\rfloor} \sum_{j:v_{i,j}\text{ is small}} |T_{v_{i,j}}|2^{(2\mathsf{level}(v_{i,j})+1)/b}\\ &\leq 2^{(2\mathsf{level}(u_1)-1)/b}(2\log n)^{\lfloor \mathsf{level}(u_1)/c\rfloor} \sum_{j:v_{i,j}\text{ is small}} |T_{v_{i,j}}|.\end{aligned}$$

For any big child $v_{i,j}$, we have $\mathsf{span}(v_{i,j}) \leq 2^{(2\mathsf{level}(u_1)-1)/b}(2\log n)^{\lfloor \mathsf{level}(u_1)/c\rfloor}|T_{v_{i,j}}|$. After that, bound rounding for head of heavy paths adds another factor of $2^{1/b}$, and at this moment:

$$\begin{aligned}&\mathsf{span}(u_1)\\ &\leq 2^{(2\mathsf{level}(u_1)-1)/b}(2\log n)^{\lfloor \mathsf{level}(u_1)/c\rfloor}\sum_{i}^{p}(1+\sum_{j}|T_{v_{i,j}}|)\\ &= |T_{u_1}|2^{(2\mathsf{level}(u_1)-1)/b}(2\log n)^{\lfloor \mathsf{level}(u_1)/c\rfloor}.\end{aligned}$$

But we might increase $\mathsf{bound}(u_1)$ by another factor of $2^{1/b}$, so indeed finally we have: $\mathsf{span}(u_1) \leq |T_{u_1}|2^{2\mathsf{level}(u_1)/b}(2\log n)^{\lfloor \mathsf{level}(u_1)/c\rfloor}$. □

Thus:

$$\begin{aligned}\mathsf{span}(\mathsf{root}(T)) &\leq n2^{2\log n/b}(2\log n)^{\log n/c}\\ &\leq 2^{\log n+\mathcal{O}(\log n/b+\log\log n\cdot\log n/c)},\end{aligned}$$

and storing the final $\mathsf{start}(u)$ for any $u \in T$ takes $\log n + \mathcal{O}(\log n/b + \log\log n \cdot \log n/c)$ bits, while storing $\mathsf{bound}(u)$ takes only $\mathcal{O}(\log(b \cdot \log n))$ bits, under earlier assumptions on b and c. Furthermore, now by inspecting the proof we obtain that, for every head of a heavy path u_1, $\mathsf{bound}(u_1)$ value can be always modified so that $\mathsf{span}(u_1)/(|T_{u_1}|2^{2\mathsf{level}(u_1)/b}(2\log n)^{\lfloor \mathsf{level}(u_1)/c \rfloor}) \in (2^{-1/b}, 1]$, increasing it if necessary.

Encoder. We are now ready to present the details of our routing scheme. Ports are numbered just according to the non-increasing size of subtrees, with port 1 leading to the heavy child. $\mathsf{start}(u)$ and $\mathsf{bound}(u)$ are stored inside $\ell(u)$. Additionally, every node $u \in T$ stores information about its big children that we call its routing table. For any big child v_j, we have that $\mathsf{level}(v_j) \in (\mathsf{level}(u) - c, \mathsf{level}(u))$. We also made sure that $\mathsf{span}(v_j) = 2^{t/b}$ for some t, as v_j is a head of its heavy path. Finally, $\mathsf{span}(v_j)/(|T_{v_j}|2^{2\mathsf{level}(v_j)/b}(2\log n)^{\lfloor \mathsf{level}(v_j)/c \rfloor}) \in (2^{-1/b}, 1]$. Thus, the number of possible distinct values of $\mathsf{span}(v_j)$ is not larger than $b \cdot \log(2^{c+1}2^{2c/b}\log n)$, which is $\mathcal{O}(b \cdot c)$ assuming $c = \Omega(\log\log n)$. For each of these $\mathcal{O}(b \cdot c)$ distinct values, we store how many big children v_j of u have such $\mathsf{span}(v_j)$. This number is at most 2^c, so all these values take $\mathcal{O}(b \cdot c^2)$ bits in total.

Final structure of a label. Label $\ell(u)$ of node u is composed of several parts:

1. $\mathsf{start}(u)$,
2. $\mathsf{bound}(u) = \lfloor 2^{t/b} \rfloor$ stored as t in binary,
3. $\mathsf{small}'(u)$,
4. routing table $\mathsf{rt}(u)$ storing the number of big children of every possible size,
5. $\mathsf{level}(u)$.

Constants b and c will be computable from the value of $\lceil \log n \rceil$. All labels are distinct because start values are distinct.

Decoder. As for decoding, with $\ell(u)$ and $\ell(w)$ we first check whether we should choose port 0 to the parent of u (when $\mathsf{start}(w) \notin [\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$). In the other case, u is an ancestor of w, and we need to route down. Entries of the routing table storing the number of big children of given size are checked one by one, while maintaining the sum of the values of span for the already considered children, until this sum is at least $\mathsf{start}(w) - \mathsf{start}(u)$, indicating the correct port number. If this does not happen, known value of $\mathsf{small}'(u)\log\mathsf{small}'(u)$ allows us to check whether w is in the subtree of some small child. If so, the correct port can be computed from a harmonic sequence. The last remaining option is port 1 leading to the heavy child, which is chosen when $\mathsf{start}(w) - \mathsf{start}(u)$ is greater than the sum of spans of all big children and the interval of light children.

Length of a label. Now we can analyse the total length of a label:

$$\begin{aligned}&\log n + \mathcal{O}(\log n/b + \log\log n \cdot \log n/c)\\&+ \mathcal{O}(\log(b \cdot \log n)) + \mathcal{O}(b \cdot c^2)\\&= \log n + \mathcal{O}(\log n/b + b \cdot c^2 + \log\log n \cdot \log n/c)\end{aligned}$$

We minimise the above expression by setting $b = (\log^{1/4} n)/\sqrt{\log\log n}$ and $c = (\log^{1/4} n)\sqrt{\log\log n}$. This results in labels of total length $\log n + \mathcal{O}((\log^{3/4} n)\sqrt{\log\log n})$. The obtained labeling scheme is canonical, as the encoder assigns port 1 to the heavy child even though it is processed after all the other children.

3.4 Intermediate Scheme with Double Rounding

In this section we will prove the following theorem:

Theorem 3.8. *There exists a canonical labeling scheme for routing in trees on n nodes with labels of length* $\log n + \mathcal{O}(\sqrt{\log n \log\log n})$.

The method will be further refined in the next section to arrive at our final result. From now on we slightly change the high-level way in which assignment of labels works, splitting it into two phases for the encoder. The main first phase works in a bottom-up manner and provides the necessary statistics for creating labels. Every node in a tree is assigned the size of its *reserved segment*, and the *routing tables* are created. Then the second top-down phase deals with assigning proper values of $\mathsf{start}(u)$ and $\mathsf{bound}(u)$, with the sizes of the reserved segments guaranteeing such assignment to be possible. For this section, we could do with just one phase, but this distinction will be useful later.

We use a positive integer parameter b to be fixed later. The main concept is that, like before, we assign to nodes intervals of numbers from range $[1, n2^{\mathcal{O}(\log n/b)}]$, so that the IDs can be stored on $\log n + \mathcal{O}(\log n/b)$ bits, and we allow span to be rounded up by a factor of $2^{\mathcal{O}(1)/b}$ for every light edge. We will use about $b \log\log n$ bits in each node to store a routing table. Moreover, we denote by $\mathsf{sl}(u_1)$ the length of the reserved segment for u_1 being the head of its heavy path. Our intention is that $\mathsf{sl}(u_1)$ should be guaranteed to be at least as big as $\mathsf{span}(u_1)$, so given any interval of length at least $\mathsf{sl}(u_1)$ we should be able to assign unique start and proper bound values in the whole subtree T_{u_1}. To facilitate efficient encoding, $\mathsf{sl}(u_1)$ will be rounded up two times when processing the parent of u_1. Firstly, there is rounding to $\mathsf{sl}'(u_1)$, when we ensure there is only a small set of possible values for sizes of the children's segments. Secondly, rounding to $\mathsf{sl}''(u_1)$ happens, when we gather children in groups and make the segment size of everyone in a group equal. $\mathsf{rt}(u)$ will denote a binary string representing the routing table for u used to route down to the children of u.

3.4.1 Encoder

First phase — segments assignment. First phase of the encoder generates rounded segment sizes and routing tables. The respective values are assigned in a bottom-up manner, guided by the heavy path decomposition. Consider a heavy path $P = u_1 - u_2 - \ldots - u_p$, where $\mathsf{head}(P) = u_1$, and assume that we have already visited all nodes in the subtrees hanging off P. By $\mathsf{lw}(u)$ we denote the *light weight* of a node u, that is the sum of subtrees' sizes of its light children. To make calculations less technical and reduce the number of cases, in the following we will assume that for every node u $\mathsf{lw}(u) \geq c'$, for some constant c'. This is obviously not true for the input tree, but as explained later it can be achieved for example by adding c' leaf children to every node of the input tree and making some simple adjustments to the encoder to handle these leaves. For now, we assume that $\mathsf{lw}(u) \geq 4$, so that $\lfloor \log \log \mathsf{lw}(u) \rfloor$ is non-zero. With a scheme from this section we are able to achieve the following:

Claim 3.9. *Suppose that* lw *of every node is at least* 4. *Then, for* $b \geq 6$, *using* $\mathsf{sl}(u_1) = |T_{u_1}| 2^{12\mathsf{level}(u_1)/b}$ *for every* u_1 *being the head of a heavy path is sufficient for the encoder to be able to store* $\mathsf{rt}(u)$ *on only* $\mathcal{O}(b \log \log \mathsf{lw}(u))$ *bits for every node* u.

This will be proved inductively for the procedure described below. The procedure traverses the path from its head, one node at a time. Let $v_{i,1}, v_{i,2}, \ldots, v_{i,\mathsf{deg}(u_i)-1}$ be the light children of u_i, sorted in the non-increasing order by the values of $\mathsf{sl}(v_{i,j})$ (note that this is also non-increasing by the values of $|T_{v_{i,j}}|$). In the following, we will often refer to light children as just children, since the heavy child u_{i+1} will be handled separately later and it does not introduce rounding. Important observation is that often we can afford more rounding than just by a factor of $2^{\mathcal{O}(1)/b}$. In fact, if $\mathsf{level}(v_{i,j}) = \mathsf{level}(u_i) - k$, then rounding by $2^{\mathcal{O}(k)/b}$ still guarantees a good final bound, as we allow rounding by $2^{\mathcal{O}(1)/b}$ for every skipped level, and thus we can agressively round smaller children, especially when using partition more refined than just into small or big children as in the previous section.

Rounding in classes. Let $l = \min(\lfloor \log \mathsf{lw}(u_i) \rfloor + 1, \mathsf{level}(u_i))$, so that level of any light child of u_i is less than l. Define preclasses as (possibly empty) sets of children of u_i with the same value of level, with children from the first preclass having level equal to $l-1$, from the second preclass level $l-2$ and so on. Then classes are defined as follows: for the first $b-1$ preclasses, the k-th preclass is evenly divided into $\lceil b/k \rceil$ classes. So if the k-th preclass consists of children with level equal to x, then the sizes of their subtrees are in $[2^x, 2^{x+1})$, and the first class created by dividing this preclass consists of children with the size of the subtree in $[2^x, 2^{x+k/b})$, the second class in $[2^{x+k/b}, 2^{x+2k/b})$, and so on, the last one possibly having smaller interval. This process of subdividing the first $b-1$ preclasses creates at most $\sum_{i=1}^{b-1} \lceil b/i \rceil \leq b(\ln b + 1) + b \leq b(\log b + 2)$ classes. Preclasses with rank at least b are not divided but merged into classes. Preclasses with ranks from b to $2b-1$ are just left as b separate classes, then preclasses $2b, \ldots, 4b-1$ are merged in pairs into b classes, the next $4b$ preclasses are merged in quadruples into b classes, and so on. The last class might be composed of a number of preclasses not being power of two. It can

be seen that at most $b\lceil \log(\log \mathsf{lw}(u_i)/b)\rceil$ classes are created. In total we have no more than $b(\log b + \log(\log \mathsf{lw}(u_i)/b) + 3) \leq b(\log\log \mathsf{lw}(u_i) + 3)$ classes. Pseudocode for this procedure is presented in Algorithm 3.3.

Algorithm 3.3 Dividing light children of a node into classes according to their sizes.

function CONSTRUCT-CLASSES(u_i, b)
  Input: node u_i and its light children $v_{i,1}, \ldots, v_{i,\mathsf{deg}(u_i)-1}$, parameter b.
  Output: partition of light children into classes.

  $PC \leftarrow \emptyset$ ▷ Division into preclasses
  $l \leftarrow \min(\lfloor \log \mathsf{lw}(u_i)\rfloor + 1, \mathsf{level}(u_i))$
  for $k = 1 \ldots l$ **do**
    $pc_k \leftarrow \{v_{i,j} : \mathsf{level}(v_{i,j}) = l - k\}$ ▷ Preclass is the set of children with the same level
    $PC \leftarrow PC \cup \{pc_k\}$

  $C \leftarrow \emptyset$ ▷ Division into classes
  for $k = 1 \ldots b-1$ **do** ▷ First $b-1$ preclasses are subdivided
    for $p = 1 \ldots b/k$ **do**
      $cl \leftarrow \{v_{i,j} : |T_{v_{i,j}}| \in [2^{l-k+(p-1)k/b}, 2^{l-k+pk/b})\}$
      Set $2^{l-k+pk/b}2^{12(l-k)/b}$ as the boundary value of class cl
      $C \leftarrow C \cup \{cl\}$
  $j \leftarrow b, z \leftarrow 1$
  $cap \leftarrow l - b + 1$
  while $j \leq l$ **do** ▷ The remaining preclasses are merged into classes
    for $k = 1 \ldots b$ **do**
      $cl \leftarrow \bigcup_{m=j+(k-1)z}^{\min(j+kz-1,l)} pc_m$
      Set $2^{cap(1+12/b)}$ as the boundary value of class cl
      $C \leftarrow C \cup \{cl\}$
      $cap \leftarrow cap - z$
    $j \leftarrow 2j, z \leftarrow 2z$
  return C

A *boundary value* of a class is an upper bound on the possible sl value in this class. If a class consists of every light child $v_{i,j}$ with $|T_{v_{i,j}}| \in [x_1, x_2 + 1)$, then the boundary value of this class is $x_2 2^{12\lfloor \log x_2\rfloor/b}$, as $\mathsf{level}(u) = \lfloor \log |T_u|\rfloor$ and $\mathsf{sl}(u) = |T_u|2^{12\mathsf{level}(u)/b}$. With division into classes as described, we achieve the promised property that rounding factor depends heavily on the level of a child:

Lemma 3.10. *Let $l = \min(\lfloor \log \mathsf{lw}(u_i)\rfloor + 1, \mathsf{level}(u_i))$. Assuming $b \geq 6$, if a node $v_{i,j}$ with $\mathsf{level}(v_{i,j}) = l - k$ is part of class C, then the boundary value of this class is no larger than $\mathsf{sl}(v_{i,j})2^{3k/b}$.*

Proof. If C is one of the classes subdividing some preclass of rank at most $b-1$, then we have that the boundary value of this class is actually no larger than $\mathsf{sl}(v_{i,j})2^{k/b}$, as interval of C spans inside just a single level subdivided into b/k classes. Now assume that C is a class constructed by merging r preclasses of rank between rb and $2rb-1$. Then level of two children in C must differ by less than r, which means the sizes of their subtrees differ by less than a factor of 2^r. Thus, the boundary value of C is less than $\mathsf{sl}(v_{i,j})2^r 2^{12r/b}$. Since $b \geq 6$, this is at most $\mathsf{sl}(v_{i,j})2^{3r}$. As C is constructed by merging preclasses of rank at least rb, then $\mathsf{level}(v_{i,j}) \leq l - rb$, so we have that $k \geq rb$ and finally the boundary value of C is at most $\mathsf{sl}(v_{i,j})2^{3k/b}$. □

For every class C, the encoder rounds the segment length of each child in C up to the boundary value of C. This way, rounding in classes will increase the size of a reserved segment of $v_{i,j}$ to $\mathsf{sl}'(v_{i,j})$, with

$$\begin{aligned}\mathsf{sl}'(v_{i,j}) &\leq \mathsf{sl}(v_{i,j})2^{3(l-\mathsf{level}(v_{i,j}))/b} \\ &\leq \mathsf{sl}(v_{i,j})2^{3(\mathsf{level}(u_i)-\mathsf{level}(v_{i,j}))/b},\end{aligned}$$

and the last inequality being sufficient for this section. By rounding in classes, we achieved a small number of different possible segment lengths without too much rounding.

Note that the set of possible values of $\mathsf{sl}'(v_{i,j})$ (boundary values for classes) depends only on b and l, not the actual children sizes. This is crucial, as it allows for compact encoding of the routing table if we know these two values. Still, there can be as many as $\Omega(n)$ children with a given rounded segment length, which prohibits us from just storing their cardinality directly in $\mathsf{rt}(u_i)$. This problem can be once again solved by grouping and rounding up.

Rounding in groups. After rounding in classes, we have at most $z = b(\log\log \mathsf{lw}(u_i) + 3)$ different possible sizes of segments (sl' values). Our second goal is to divide children into at most z groups of fixed sizes and make their segment lengths equal. Then we will be able to use the following:

Proposition 3.11. *A non-increasing sequence of at most z integers from $[0, z]$ can be encoded on $2z$ bits.*

Proof. The decoder starts with a counter set to z. Then a sequence is stored as a bit string in which 0 tells the decoder to decrease the counter by one, and 1 tells the decoder that the next element of the sequence is equal to the current counter. □

Elements of the sequence will correspond to consecutive groups of children, and their values to the set of boundary values defined by the classes. Thus, these values correspond to the rounded sizes of reserved segments in groups. To achieve the stated goal, we define pregroups and groups, similarly to preclasses and classes. Children of u_i are divided into at most l pregroups, the k-th one containing 2^k consecutive children, sorted in the non-increasing order by the subtree size, or equivalently sl'. The last pregroup is padded with dummy children of size 0 if needed, as we cannot afford storing the exact degree of a node. Then, as before, groups are defined in the following

way: for the $b-1$ first pregroups, the k-th pregroup is divided into $\lceil b/k \rceil$ groups with roughly equal number of children (some are possibly empty). Pregroups with rank at least b are not divided but merged into groups. Pregroups with ranks from b to $2b-1$ are just left as b separate groups, then pregroups $2b \dots 4b-1$ are merged in pairs into b groups, the next $4b$ pregroups are merged in quadruples for another b groups, and so on. Total number of created groups is not greater than the number of classes. The encoder, for every group, rounds the reserved segment length of every child in this group up to the length of the largest one. Pseudocode for rounding in groups is presented in Algorithm 3.4, and Figure 3.2 depicts rounding in classes and groups.

Algorithm 3.4 Dividing light children of a node into groups in which the segment lengths are rounded up to the same value.

function CONSTRUCT-GROUPS(u_i, b)
  Input: node u_i and its light children $v_{i,1}, \dots, v_{i,\mathsf{deg}(u_i)-1}$, parameter b.
  Requirement: light children in the non-increasing order by their size.
  Output: partition of light children into groups.

  $PG \leftarrow \emptyset$ ▷ Division into pregroups
  $j \leftarrow 1, c \leftarrow 0$
  while $j < \mathsf{deg}(u_i)$ **do**
    $c \leftarrow c + 1$
    $pg_c \leftarrow \{v_{i,j}, \dots, v_{i,\min(\mathsf{deg}(u_i)-1, 2j)}\}$
    $PG \leftarrow PG \cup \{pg_c\}$ ▷ Pregroups of first 2, then 4, then 8... children
    $j \leftarrow 2j + 1$
  if $|pg_c|$ is not power of two **then**
    Add to pg_c sufficiently many dummy children of size 0

  $G \leftarrow \emptyset$ ▷ Division into groups
  for $j = 1 \dots b-1$ **do** ▷ First $b-1$ pregroups are subdivided
    Order the children in pg_j by size
    Divide pg_j into b/j consecutive groups of roughly equal number of children, add them to G
  $j \leftarrow b, z \leftarrow 1$
  while $j \leq c$ **do** ▷ The remaining pregroups are merged into groups
    for $k = 1 \dots \min(j+kz-1, c)$ **do**
      $g \leftarrow \bigcup_{m=j+(k-1)z} pg_m$
      $G \leftarrow G \cup \{g\}$
    $j \leftarrow 2j, z \leftarrow 2z$
  return G

Note that the number of children in any group depends only on b, and the number of pregroups depends only on $\log(\mathsf{deg}(u_i))$, which is crucial for encoding of the routing table. Only the number

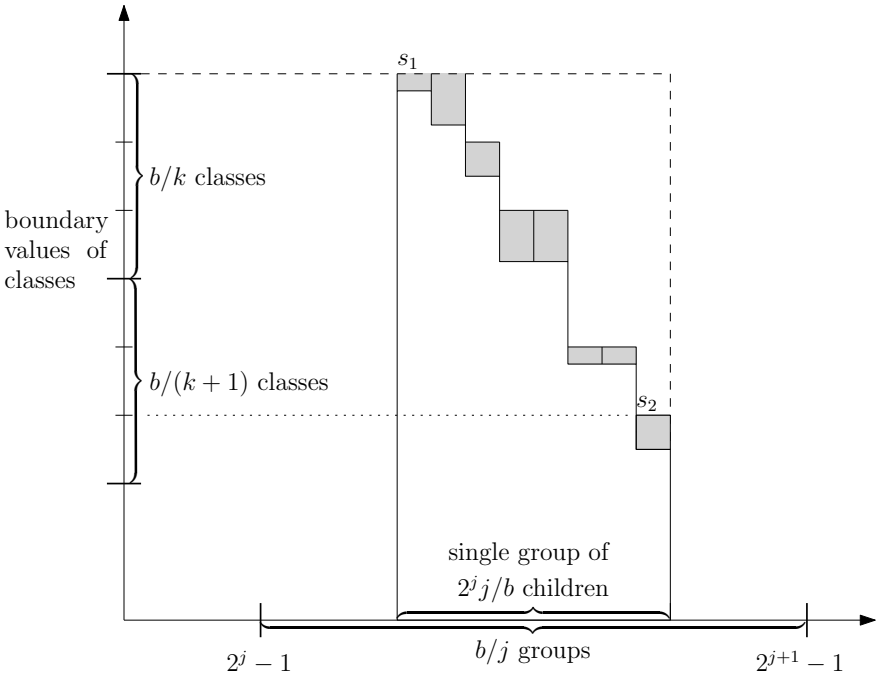


Figure 3.2: Rounding of one group, with children on the horizontal axis and their segment lengths on the vertical axis, $k < b$. Children are rounded up to the boundary value of their class, here for example the rightmost child is rounded to s_2, and then the whole group is rounded up to maximum sl' value in this group, here s_1.

of dummy children cannot be reproduced from these two values. Moreover, double rounding still preserves the monotonicity of sizes of the children subtrees, so our routing scheme will be canonical. Now we need to prove a bound on the total increase in lengths of segments.

Lemma 3.12. *Let* $\mathsf{sl}'(v_{i,j})$ *be the size of the reserved segment of* $v_{i,j}$ *after rounding in classes and* $l = \min(\lfloor \log \mathsf{lw}(u_i) \rfloor + 1, \mathsf{level}(u_i))$. *Then the total size of segments after rounding in groups for light children of* u_i, *including dummy ones, is bounded by* $\sum_{j \geq 1} \mathsf{sl}'(v_{i,j}) 2^{8(l - \mathsf{level}(v_{i,j}))/b}$, *provided that* $b \geq 6$.

Proof. Observe that for child $v_{i,j}$ in the k-th pregroup we have $\mathsf{level}(v_{i,j}) \leq l - k$. We assign potentials to children, child $v_{i,j}$ in the k-th pregroup is assigned value $\mathsf{sl}'(v_{i,j})(2^{8k/b} - 1)$, so if we think about the plot (as in Figure 3.2) it corresponds to $(2^{8k/b} - 1)$ units of potential assigned to every unit length of the column representing sl' value of a child from the k-th pregroup. Then we claim that the sum of children's sizes increases during the rounding is bounded by the sum of all assigned potentials.

To see this, first consider the case of a group inside the k-th pregroup, for $k < b$. It consists of at most $2^k k/b$ children. Let s_1 be the size of the first (the largest) of them, and s_2 the last (the smallest) one. Then the size of every child is rounded up to s_1. Note that in previous groups the size of every child was at least s_1 even before rounding in these groups, and that children from further groups will not be rounded to more than s_2. Thus for rounding in the considered group we can charge unit lengths from s_1 to s_2 for children in all prior groups. Charging only

the $(k-1)$-th pregroup will be enough for us. In the first pregroup there is no rounding at all, provided that $b > 1$. As $1 + x \leq e^x$ and thus $2^{2x} - 1 \geq x$, sum of the potentials of unit lengths from s_1 to s_2 for elements from the $(k-1)$-th pregroup is equal to

$$\begin{aligned}(s_1 - s_2)2^{k-1}(2^{8(k-1)/b} - 1) &\geq (s_1 - s_2)2^{k-1}4(k-1)/b \\ &\geq (s_1 - s_2)2^k k/b.\end{aligned}$$

Each of the remaining groups is created by merging some pregroups. Let g be the number of groups created from the first $b-1$ pregroups (we already dealt with them). For the increase of segments in a group of rank $g+r$, we can similarly charge only the previous group $g+r-1$, except for the special case of $r = 1$ where we would charge two groups g and $g-1$ (which together are just the $(b-1)$-th pregroup). Let $r \in (zb, zb+b]$ for $z \geq 0$, then the group of rank $g+r$ is created by merging 2^z pregroups. It can be seen that the children from group $g+r$ are from pregroups of rank at least $b2^z$, and that the children from group $g+r-1$ are from pregroups of rank at least $b2^z - 2^{z-1} - 1$. Thus, we will have a large reserve of potentials, as due to $b \geq 6$ every unit length for a child in group $g+r-1$ has potential at least

$$\begin{aligned}2^{8(b2^z - 2^{z-1} - 1)/b} - 1 &= 2^{8\cdot 2^z - 8(2^{z-1}+1)/b} - 1 \\ &\geq 2^{8\cdot 2^z - 8(2^{z-1}+1)/6} - 1 \\ &\geq 2^{8\cdot 2^z - 2^z - 2} - 1 \\ &\geq 2^{5\cdot 2^z} - 1 \\ &\geq 2^{2^z+1}.\end{aligned}$$

Since the number of children in group $g+r$ is at most 2^{2^z+1} times larger than the number of children in group $g+r-1$, an increase in the size of segments in group $g+k$ can be bounded by the sum of potentials of respective unit lengths from s_1 to s_2 from group $g+k-1$, as before. □

Bound on the length of a segment. After rounding in classes and groups for a node u_i, $\mathsf{rt}(u_i)$ is set to be description of the rounded groups as discussed in Proposition 3.11, consisting of $\mathcal{O}(b \log\log \mathsf{lw}(u_i))$ bits. Let $\mathsf{sl}''(v_{i,j})$ be the size of the reserved segment of child $v_{i,j}$ after rounding in groups. Then it can be seen that segment of length $1 + \sum_{j\geq 1} \mathsf{sl}''(v_{i,j})$ will allow us to set the IDs for u_i and subtrees rooted at its light children. As the length of the considered heavy path is p, then similarly with a segment of length $p + \sum_{i=1}^{p} \sum_{j\geq 1} \mathsf{sl}''(v_{i,j})$ it will be possible to assign IDs in the whole T_{u_1}.

Lemma 3.13. *$\mathsf{sl}(u_1) = |T_{u_1}|2^{12\mathsf{level}(u_1)/b}$ is enough for the encoder to properly assign IDs in the whole T_{u_1}.*

Algorithm 3.5 First phase of the encoder, assigning sl'' and rt on a heavy path.

function ASSIGN-$\mathsf{sl}(P)$
  Input: heavy path $P = u_1, \ldots, u_p$. b is a fixed parameter.
  Requirement: all nodes in the subtrees hanging off P are already processed.
  Output: $\mathsf{rt}(u_i), \mathsf{sl}''(v_{i,j})$ values for every u_i.

  $r \leftarrow 0$
  for $i = 1 \ldots p$ **do**
    $C \leftarrow$ CONSTRUCT-CLASSES(u_i, b)
    for $j = 1 \ldots \mathsf{deg}(u_i) - 1$ **do** ▷ Light children of u_i
      Set $\mathsf{sl}'(v_{i,j})$ to be the boundary value of the class of $v_{i,j}$ in C ▷ First rounding
    $G \leftarrow$ CONSTRUCT-GROUPS(u_i, b) ▷ Note that dummy nodes are added here
    $\mathsf{segment}_i = 1$
    for $j = 1 \ldots \mathsf{deg}(u_i) - 1$ **do**
      Set $\mathsf{sl}''(v_{i,j})$ to be sl' of the largest child in group of $v_{i,j}$ in G ▷ Second rounding
      $\mathsf{segment}_i \leftarrow \mathsf{segment}_i + \mathsf{sl}''(v_{i,j})$
    $r \leftarrow r + \mathsf{segment}_i$
    Set $\mathsf{rt}(u_i)$ as a description of rounded groups ▷ Create the routing table
  Guaranteed property: $r \leq |T_{u_1}| 2^{(12\mathsf{level}(u_1)-1)/b}$
  $\mathsf{sl}(u_1) \leftarrow |T_{u_1}| 2^{12\mathsf{level}(u_1)/b}$

Proof. By Lemma 3.10, Lemma 3.12 and induction on the light depth of a node, we have:

$$\begin{aligned} & p + \sum_{i=1}^{p} \sum_{j \geq 1} \mathsf{sl}''(v_{i,j}) \\ & \leq p + \sum_{i=1}^{p} \sum_{j \geq 1} \mathsf{sl}(v_{i,j}) 2^{11(\mathsf{level}(u_1) - \mathsf{level}(v_{i,j}))/b} \\ & \leq p + \sum_{i=1}^{p} \sum_{j \geq 1} |T_{v_{i,j}}| 2^{12\mathsf{level}(v_{i,j})/b} 2^{11(\mathsf{level}(u_1) - \mathsf{level}(v_{i,j}))/b} \\ & \leq |T_{u_1}| 2^{(12\mathsf{level}(u_1)-1)/b}. \end{aligned}$$

Then, another factor of $2^{1/b}$ factor is needed for rounding the value of bound. □

Thus in the first phase, at the end of processing of a heavy path, $\mathsf{sl}(u_1)$ is set to be just $|T_{u_1}| 2^{12\mathsf{level}(u_1)/b}$. This way the bound from Claim 3.9 is satisfied. Pseudocode for the first phase of the encoder is presented in Algorithm 3.5.

Second phase — creating labels. The second phase of the encoder only unfolds the assigned sl'' into concrete values of start and bound for every node. It works in a top-down manner, using a recursive procedure scanning one heavy path in the tree at a time. We start in the root with the accumulator A set to 0. At node u_i on a path, the procedure sets $\mathsf{start}(u_i)$

Algorithm 3.6 Second phase of the encoder, recursively assigning start and bound values.

function ASSIGN-IDS(P, s)
  Input: heavy path $P = u_1, \ldots, u_p$. b is a fixed parameter.
  Output: $\mathsf{start}(u)$, $\mathsf{bound}(u)$ for every $u \in T_{u_1}$, $\mathsf{start}(u) \in [s, s+\mathsf{sl}(u_1))$, $\mathsf{span}(u_1) \leq \mathsf{sl}(u_1)$.

  $A \leftarrow s$ ▷ Accumulator
  for $i = 1 \ldots p$ **do**
    $\mathsf{start}(u_i) \leftarrow A$
    $A \leftarrow A + 1$
    for $j = 1 \ldots \mathsf{deg}(u_i) - 1$ **do**
      ASSIGN-IDS(P', A), where P' is a heavy path with $v_{i,j}$ as the head
      $A \leftarrow A + \mathsf{sl}''(v_{i,j})$
  for $i = 1 \ldots p$ **do**
    Let t be the smallest natural number such that $\lfloor 2^{t/b} \rfloor + \mathsf{start}(u_i) \geq A$
    $\mathsf{bound}(u_i) \leftarrow \lfloor 2^{t/b} \rfloor$

to be the current value of the accumulator and then increases the accumulator by 1. Then it iterates over the light children $v_{i,1}, v_{i,2}, \ldots$ of u_i. For each $v_{i,j}$ we call the recursive procedure with a copy of the current accumulator as an argument, and then increase the accumulator by $\mathsf{sl}''(v_{i,j})$. Finally, we need to set the proper $\mathsf{bound}(u_i)$. It can be seen that we only need that $\mathsf{start}(u_i) + \mathsf{bound}(u_i)$ is at least as large as the final accumulator. Further, if the starting value of accumulator for a heavy path P was s, the final accumulator is $A = s + d$, where $d = \sum_{i=1}^{p}(1 + \sum_{j \geq 1} \mathsf{sl}''(v_{i,j}))$. Then, because $\mathsf{start}(u_i) \geq s$ and possible values of $\mathsf{bound}(u_i)$ are all numbers of the form $\lfloor 2^{t/b} \rfloor$, we can always choose t so that $\mathsf{start}(u_i) + \mathsf{bound}(u_i) \leq s + d2^{1/b}$. At the end, by construction we obtain that $\mathsf{span}(u_1) \leq \mathsf{sl}(u_1)$. Note that in total we round the size of a given subtree twelve times for each passed level: three times for rounding in classes to sl', (amortized) eight times for rounding in groups to sl'' and once when bound is increased to be power of $2^{1/b}$. Pseudocode for the second phase of the encoder is presented in Algorithm 3.6.

3.4.2 Resulting Labels

Final structure of a label. Label $\ell(u)$ of node u is composed of six parts:

1. $\mathsf{start}(u)$,
2. $\mathsf{bound}(u) = \lfloor 2^{t/b} \rfloor$ stored as just t,
3. an encoding of $\mathsf{rt}(u)$,
4. $\lfloor \log \mathsf{lw}(u) \rfloor$,
5. $\mathsf{level}(u)$,
6. $\lceil \log (\deg(u) - 1) \rceil$.

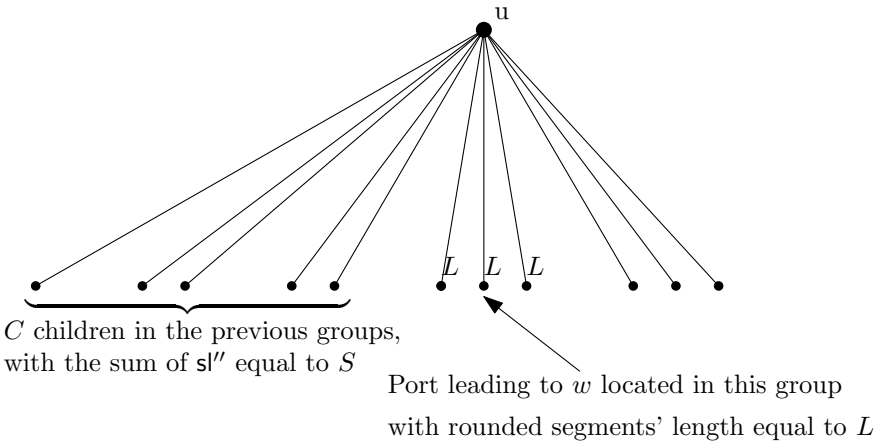


Figure 3.3: Information needed during a query: locating the correct group, the rounded size of a reserved segment there, and prefix sums of the number of ports and segment sizes in all previous groups. $\mathsf{start}(w) = \mathsf{start}(u) + S + kL + x$, and port $C + k + 2$ is chosen.

b will be computable from the known value of $\lceil \log n \rceil$. The labels are distinct because start values are distinct.

Length of a label. From Lemma 3.13 we have $\mathsf{span}(\mathsf{root}(T)) \leq \mathsf{sl}(\mathsf{root}(T)) = n2^{12 \log n/b}$. Then every start value can be stored on $\log n + \mathcal{O}(\log n/b)$ bits, bound on $\mathcal{O}(\log(b \log n))$ bits and rt on $\mathcal{O}(b \log\log \mathsf{lw}(u_i)) = \mathcal{O}(b \log\log n)$ bits. Three last elements of a label can be stored on $\mathcal{O}(\log\log n)$ bits. By taking $b = \sqrt{\log n / \log\log n}$, which is at least 6 for n large enough, we obtain labels of length $\log n + \mathcal{O}(\sqrt{\log n \log\log n})$.

3.4.3 Decoder

The decoder can answer routing queries in polylogarithmic time, since the number of groups is polylogarithmic. From $\ell(u)$ and $\ell(w)$ it can easily extract rt, start and bound of both nodes. If $\mathsf{start}(w) \notin [\mathsf{start}(u), \mathsf{start}(u) + \mathsf{bound}(u))$, then $w \notin T_u$ and port number 0 is chosen. Otherwise $\mathsf{rt}(u)$ is read and rounded sl'' of every light child of u is retrieved. We have groups of children of predefined sizes (depending only on b and three last elements of a label), with every child in a single group with the same sl''. Let $v_1, v_2, \ldots$ be the light children of u sorted by non-increasing size. Then port 1 leading to heavy child of u is chosen if $\mathsf{start}(w) > \mathsf{start}(u) + \sum_{j=1} \mathsf{sl}''(v_j)$. In the remaining case, port $i+1$ is chosen such that i is smallest number for which $\mathsf{start}(w) \leq \mathsf{start}(u) + \sum_{j=1}^{i} \mathsf{sl}''(v_j)$ holds. Port with a number greater than $\deg(u)$ will never be chosen. Consult Figure 3.3 for the situation during a query.

3.5 Final Scheme with Distribution of Bits

In this section we design our final routing labeling scheme and prove Theorem 3.1.

The overall approach is as in the previous section, but now we choose b to not be the same value in every node, and move start values a little to make this idea work efficiently. First, notice that it might be excessive for $\mathsf{rt}(u)$ to always have the same amount of reserved bits. Indeed, in most cases the light weight of a node is small, and then we can get away with only a few bits for the routing table. On the other hand, for nodes with a very significant light weight, it would be beneficial to use much more bits to reduce the impact of rounding, since this rounding affects a huge part of the tree. Thus, we use a simple method for compressing start values depending on the light weight of a given node, known from other works in this area.

The idea is that if the binary expansion of $\mathsf{start}(u)$ happens to have l trailing zeroes, then we can save l bits by removing these trailing zeroes, assuming that we are additionally storing their number. Thus, we add $\mathcal{O}(\log\log n)$ bits to every label, but then are able to substantially shorten some of them. The saved space can be used to store a relatively big routing table. As we cannot be sure that binary expansions have a lot of trailing zeroes, we will enforce such a property. If the light weight of u is $\mathsf{lw}(u) = l$, we reserve an interval of length $2^{\lceil \log l \rceil}$ exclusively for choosing the value $start(u)$. In such an interval surely there is always a number with at least $\log l$ trailing zeroes, so using that many bits for storing $rt(u)$ will be possible, which translates into $b = \Omega(\log l / \log\log l)$ for node u. Moreover, the total size of intervals reserved this way is not too big, as every node contributes to the light weight of at most $\log n$ ancestors, so the sum of intervals reserved for shifting start values will be at most $2n \log n$. Note that equivalently we could store $\mathsf{rt}(u)$ as a suffix of the shifted value of start, instead of explicitly creating additional space by enforcing the existence of $\log l$ trailing zeroes, but we find the latter solution cleaner for our purposes.

3.5.1 Encoder

First phase. For the first phase, we need to modify the previous bound for sl.

Claim 3.14. *Suppose* lw *of every node is bigger than some constant* c'*. Then, using for every* u_1 *being the head of a heavy path* $\mathsf{sl}(u_1) = 2|T_{u_1}|\mathsf{level}(u_1)2^{\mathsf{level}(u_1)/\log n}\prod_{k=1}^{\mathsf{level}(u_1)} 2^{14c\log k/k}$*, for some big enough constant* c*, is sufficient for the encoder to achieve both* $|\mathsf{rt}(u)| \leq \log \mathsf{lw}(u)$ *and* $\mathsf{start}(u)$ *having* $\log \mathsf{lw}(u)$ *trailing zeroes for every node* u*.*

Claim 3.14 will be proved inductively with the modified procedure described below. We need to make several changes to the encoder from Section 3.4. Firstly, b is no longer a global parameter, but rather a possibly different value for every node. For a node u we use $b = \log \mathsf{lw}(u)/c \log\log \mathsf{lw}(u)$, for some constant c. The product in the bound for $\mathsf{sl}(u)$ reflects such a choice of b, going over all levels up to $\mathsf{level}(u)$. Note that rounding factors are decreasing, as the values of b are increasing and thus $2^{1/b}$ are decreasing. We require that $b \geq 6$, meaning that for every node $\log \mathsf{lw}(u)/\log\log \mathsf{lw}(u) \geq 6c$, which is realised if we assume that $\mathsf{lw}(u)$ is big enough. Secondly, we increase the range for start value from just 1 to $2\mathsf{lw}(u)$, ensuring that the assignment of $\mathsf{start}(u)$ with sufficient number of trailing zeroes will be possible.

Thirdly, we need to adjust bound rounding. As the value of b is now set locally for a node u, depending solely on its light weight, we can no longer round up $\mathsf{bound}(u)$ by a factor of $2^{1/b}$, since this rounding applies to more than just the subtrees of light children of u. Because of that, now for every node we just round bound by a factor $2^{1/\log n}$. This is a minor issue, as storing bound was never main factor in the length of a label, and such modified values of bound will still be stored on just $\mathcal{O}(\log\log n)$ bits. Lastly, we need to revisit rounding in classes. Rounding in groups works exactly as before. However, for classes, because of the new formula for length of segments, now we have different boundary values than in Section 3.4. Namely, if a class consists of every light child $v_{i,j}$ with $|T_{v_{i,j}}| \in [x_1, x_2+1)$, then the boundary value of this class will be $2x_2 \log x_2 2^{\log x_2/\log n} \prod_{k=1}^{\log x_2} 2^{14c\log k/k}$. After this change we have a bound analogous to Lemma 3.10:

Lemma 3.15. *Consider node u_i for which we have $b_i = \log \mathsf{lw}(u_i)/c \log\log \mathsf{lw}(u_i)$, and let $l = \min(\lfloor \log \mathsf{lw}(u_i)\rfloor + 1, \mathsf{level}(u_i))$. If $v_{i,j}$, a light child of u_i with $\mathsf{level}(v_{i,j}) = l - k$, is part of class C, then the boundary value of this class is no larger than $\mathsf{sl}(v_{i,j})2^{6k/b_i}$.*

Proof. If C is one of the classes subdividing some preclass of rank at most $b_i - 1$, then as before the boundary value of this class is actually no larger than $\mathsf{sl}(v_{i,j})2^{k/b_i}$, as interval of C spans inside just a single level, and this level was subdivided into b_i/k classes. Now assume that C is a class constructed by merging r preclasses of rank between rb_i and $2rb_i - 1$. Then level of two children in C must differ by less than r, which means that their subtrees' sizes differ by less than a factor 2^r. Thus, the boundary value of C is less than $\mathsf{sl}(v_{i,j})2^r(1+1/b_i)2^{r/\log n}\prod_{z=\mathsf{level}(v_{i,j})}^{\mathsf{level}(v_{i,j})+r-1} 2^{14c\log z/z}$. By assumption on the light weight of nodes, which guarantees that b is always at least 6, and as $\mathsf{level}(u)$ is not less than $\log \mathsf{lw}(u)$, $\prod_{z=\mathsf{level}(v_{i,j})}^{\mathsf{level}(v_{i,j})+r-1} 2^{14c\log z/z} \leq \prod_{z=\mathsf{level}(v_{i,j})}^{\mathsf{level}(v_{i,j})+r-1} 2^{14/6} \leq 2^{14r/6}$. Furthermore, $(1+1/b_i) < 2$ and $2^{r/\log n} < 2$ as $r < \log n$. Therefore the boundary value of C is less than $\mathsf{sl}(v_{i,j})2^{r+2+14r/6} < \mathsf{sl}(v_{i,j})2^{6r}$. As C is constructed by merging preclasses of rank at least rb_i, then $\mathsf{level}(v_{i,j}) \leq l - rb_i$, so we have that $k \geq rb_i$ and finally the boundary value of C is at most $\mathsf{sl}(v_{i,j})2^{6k/b_i}$. □

Pseudocode for the first phase of the modified encoder is presented in Algorithm 3.7. It can be seen that with a segment of length $\sum_{i=1}^{p}(2\mathsf{lw}(u_i) + \sum_{j\geq 1}\mathsf{sl}''(v_{i,j}))$, it will be possible for the encoder to properly assign IDs in the whole T_{u_1}. We need a bound on this value.

Lemma 3.16.

$$\mathsf{sl}(u_1) = 2|T_{u_1}|\mathsf{level}(u_1)2^{\mathsf{level}(u_1)/\log n}\prod_{k=1}^{\mathsf{level}(u_1)} 2^{14c\log k/k}$$

is enough for the encoder to properly assign IDs in the whole T_{u_1}.

Proof. By Lemma 3.15, Lemma 3.12 and induction on the light depth of a node. Let $l_i = \min(\lfloor \log \mathsf{lw}(u_i) \rfloor + 1, \mathsf{level}(u_i))$, and $b_i = \log \mathsf{lw}(u_i)/c \log\log \mathsf{lw}(u_i)$, then we have:

$$\begin{aligned}
&\sum_{i=1}^{p}(2\mathsf{lw}(u_i) + \sum_{j\geq 1}\mathsf{sl}''(v_{i,j})) \\
&\leq \sum_{i=1}^{p}(2\mathsf{lw}(u_i) + \sum_{j\geq 1}\mathsf{sl}(v_{i,j})2^{14(l_i - \mathsf{level}(v_{i,j}))/b_i}) \\
&\leq 2|T_{u_1}| + \sum_{i=1}^{p}\sum_{j\geq 1}(2|T_{v_{i,j}}|\mathsf{level}(v_{i,j})\cdot \\
&2^{\mathsf{level}(v_{i,j})/\log n}2^{14(l_i - \mathsf{level}(v_{i,j}))/b_i}\prod_{k=1}^{\mathsf{level}(v_{i,j})} 2^{14c\log k/k}) \\
&\leq 2|T_{u_1}| + (\mathsf{level}(u_1) - 1)2^{(\mathsf{level}(u_1)-1)/\log n}\cdot \\
&\sum_{i=1}^{p}\sum_{j\geq 1} 2|T_{v_{i,j}}|\prod_{k=1}^{l_i} 2^{14c\log k/k} \\
&\leq 2|T_{u_1}|\mathsf{level}(u_1)2^{(\mathsf{level}(u_1)-1)/\log n}\prod_{k=1}^{\mathsf{level}(u_1)} 2^{14c\log k/k}.
\end{aligned}$$

Then, increase by a single factor of $2^{1/\log n}$ is needed for bound rounding. □

Thus in the first phase, at the end of processing of a heavy path, $\mathsf{sl}(u_1)$ is set as follows as to satisfy the bound from Claim 3.14 :

$$2|T_{u_1}|\mathsf{level}(u_1)2^{\mathsf{level}(u_1)/\log n}\prod_{k=1}^{\mathsf{level}(u_1)} 2^{14c\log k/k}.$$

Second phase — creating labels. The second phase is very similar to the one from Section 3.4. We start in the root with the accumulator set to 0, and then process heavy paths in a top-down manner. At node u_i on a heavy path and with the accumulator being A, the procedure sets $\mathsf{start}(u_i)$ to be the greatest value with at least $\lceil \log \mathsf{lw}(u_i) \rceil$ trailing zeroes not bigger than $A + 2\mathsf{lw}(u_i) - 1$, and then sets $A = \mathsf{start}(u_i) + 1$. Next, we iterate over the light children $v_{i,1}, v_{i,2}, \ldots$ of u_i. For each $v_{i,j}$ we call the recursive procedure with a copy of the current accumulator as an argument, and then increase the accumulator by $\mathsf{sl}''(v_{i,j})$. Finally, we need to set proper $\mathsf{bound}(u_i)$ for every $i = 1, 2, \ldots, p$, so that $\mathsf{start}(u_i) + \mathsf{bound}(u_i)$ is at least as large as the final accumulator. If the initial value of the accumulator for a heavy path P is s, then the final accumulator is $A = s + d$, where $d \leq \sum_{i=1}^{p}(2\mathsf{lw}(u_i) + \sum_{j\geq 1}\mathsf{sl}''(v_{i,j}))$. Finally, because $\mathsf{start}(u_i) \geq s$ and the possible values of $\mathsf{bound}(u_i)$ are all numbers of the form $\lfloor 2^{t/\log n} \rfloor$, by taking the smallest viable t we have $\mathsf{start}(u_i) + \mathsf{bound}(u_i) \leq s + d2^{1/\log n}$. Thus, by construction we obtain that $\mathsf{span}(u_1) \leq \mathsf{sl}(u_1)$. See Algorithm 3.8.

Algorithm 3.7 First phase of the modified encoder, assigning sl'' and rt on a heavy path. We do not include the change to the boundary values in function CONSTRUCT-CLASSES there.

function ASSIGN-$\mathsf{sl}(P)$
  Input: heavy path $P = u_1, \ldots, u_p$.
  Requirement: all nodes in the subtrees hanging off P are already processed.
  Output: $\mathsf{rt}(u_i), \mathsf{sl}''(v_{i,j})$ values for every u_i.
  c — fixed constant

  $r \leftarrow 0$
  for $i = 1 \ldots p$ **do**
    $b_i \leftarrow \log \mathsf{lw}(u_i)/c \log\log \mathsf{lw}(u_i)$
    $C \leftarrow$ CONSTRUCT-CLASSES(u_i, b_i)
    for $j = 1 \ldots \deg(u_i) - 1$ **do**
      Set $\mathsf{sl}'(v_{i,j})$ to be the boundary value of the class of $v_{i,j}$ in C
    $\mathsf{segment}_i \leftarrow 2\mathsf{lw}(u_i)$ ▷ Interval reserved for $\mathsf{start}(u_i)$
    $G \leftarrow$ CONSTRUCT-GROUPS(u_i, b_i)
    for $j = 1 \ldots \deg(u_i) - 1$ **do**
      Set $\mathsf{sl}''(v_{i,j})$ to be size of the largest child in group of $v_{i,j}$ in G
      $\mathsf{segment}_i \leftarrow \mathsf{segment}_i + \mathsf{sl}''(v_{i,j})$
    $r \leftarrow r + \mathsf{segment}_i$
    Set $\mathsf{rt}(u_i)$ as description of rounded groups
  Guaranteed property: $r \le 2|T_{u_1}|\mathsf{level}(u_1)2^{(\mathsf{level}(u_1)-1)/\log n} \prod_{k=1}^{\mathsf{level}(u_1)} 2^{14c\log k/k}$
  $\mathsf{sl}(u_1) \leftarrow 2|T_{u_1}|\mathsf{level}(u_1)2^{\mathsf{level}(u_1)/\log n} \cdot \prod_{k=1}^{\mathsf{level}(u_1)} 2^{14c\log k/k}$

3.5.2 Resulting labels

Final structure of a label. As in Section 3.4. Note that the stored $\log \mathsf{lw}(u_i)$ allows recovering b. The number of trailing zeroes in $\mathsf{start}(u)$ is saved separately, and these zeroes are cut off from the binary representation.

Length of a label. By Lemma 3.16 we have $\mathsf{span}(\mathsf{root}(T)) \le \mathsf{sl}(\mathsf{root}(T)) = 4n \log n \prod_{k=1}^{\log n} 2^{14c\log k/k}$. As we are interested in the logarithm of this value that affects the length of start, we calculate:

$$\log \prod_{k=1}^{\log n} 2^{14c\log k/k} = \mathcal{O}(\sum_{k=1}^{\log n} \log k/k) = \mathcal{O}((\log\log n)^2).$$

Note that the rounding is not evenly distributed among levels. Inside small subtrees, where we cannot provide many trailing zeroes, the values of b are relatively small and thus roundings are quite large deep in the tree. Regardless of that, every start could be stored on $\log n + \mathcal{O}((\log\log n)^2)$ bits, but every $\mathsf{start}(u)$ is guaranteed to have $\log \mathsf{lw}(u)$ trailing zeroes, so we actually store this value on $\log n + \mathcal{O}((\log\log n)^2) - \log \mathsf{lw}(u)$ bits. Clearly, $|\mathsf{bound}(u)| = \mathcal{O}(\log\log n)$.

Algorithm 3.8 Second phase of the modified encoder, recursively assigning start and bound values.

function ASSIGN-IDS(P, s)
  Input: heavy path $P = u_1, \ldots, u_p$. c is a fixed constant, $\log n$ is known.
  Output: $\mathsf{start}(u)$, $\mathsf{bound}(u)$ for every $u \in T_{u_1}$, $\mathsf{start}(u) \in [s, s+\mathsf{sl}(u_1))$, $\mathsf{span}(u_1) \leq \mathsf{sl}(u_1)$.

  $A \leftarrow s$
  for $i = 1 \ldots p$ **do**
    $b \leftarrow \log \mathsf{lw}(u_i)/c \log\log \mathsf{lw}(u_i)$
    $A \leftarrow A + 2\mathsf{lw}(u_i) - 1$ ▷ Reserving an additional interval
    $l \leftarrow 2^{\lceil \log \mathsf{lw}(u_i) \rceil}$
    $\mathsf{start}(u_i) \leftarrow \lfloor A/l \rfloor \cdot l$ ▷ start value gets enough trailing zeroes
    $A \leftarrow \mathsf{start}(u_i) + 1$
    for $j = 1 \ldots \deg(u_i) - 1$ **do**
      ASSIGN-IDS(P', A), where P' is a heavy path with $v_{i,j}$ as the head
      $A \leftarrow A + \mathsf{sl}''(v_{i,j})$
  for $i = 1 \ldots p$ **do**
    Let t be the smallest natural number such that $\lfloor 2^{t/\log n} \rfloor + \mathsf{start}(u_i) \geq A$
    $\mathsf{bound}(u_i) \leftarrow \lfloor 2^{t/\log n} \rfloor$

For b values, we use $c = 2$, or rather exactly $b = \log \mathsf{lw}(u)/2(\log\log \mathsf{lw}(u) + 3)$. As the number of bits used to describe the routing table is at most twice the number of groups, and there are at most $b(\log\log \mathsf{lw}(u)+3)$ of them, $|\mathsf{rt}(u)| \leq \log \mathsf{lw}(u)$. Overall, the labels consist of $\log n + \mathcal{O}((\log\log n)^2)$ bits.

3.5.3 Decoder

Decoder. Decoding proceeds as in Section 3.4, after retrieving the value of b.

Light weight assumption. At the beginning of Section 3.4, we assumed that for every node u $\mathsf{lw}(u) \geq c'$ for some constant c'. As hinted, it could be realised in the following way. Firstly, the encoder adds $c' + 1$ artificial leaves to every node of the input tree. Then the assumption holds for every original node, but the encoder needs to handle these additional nodes. In a heavy path decomposition, artificial nodes fall into two categories. Most of them are heads of heavy paths of length one, and remaining ones are bottom nodes at the ends of some longer heavy paths. Both can be handled very similarly, as they have no children to process. For leaves, in the Algorithm 3.7, $\mathsf{rt}(u)$ is set to null (we can use additional single bit in the label indicating whether u does have any children), and if an artificial node u is a head of a heavy path then the encoder sets $\mathsf{sl}(u) = 1$. In Algorithm 3.8, ID of an artificial node u is set to be the current value of the accumulator ($\mathsf{start}(u) \leftarrow A$), $\mathsf{bound}(u)$ is set to 1, and the accumulator is increased by one.

As we increase size of an input tree by a constant factor and our labeling scheme uses labels of length $\log n + \mathcal{O}((\log \log n)^2)$, adding artificial children increases the length of a label only by $\mathcal{O}(1)$ bits. Moreover, as a labeling scheme is canonical, ports leading to the artificial children of a node u are $\deg(u)+1, \ldots, \deg(u)+c'$. Thus, if as the last step the encoder disregards the labels of artificial nodes and edges, then we are left with a correct labeling for the input tree. The decoder does not change. Note that the concept of adding artificial leaves is used purely for simplicity of the analysis.

3.6 Simple Extensions

In Section 3.3 we presented a simple routing labeling scheme for bounded degree trees with better lengths than general scheme. Another extension is the case of a small depth.

Corollary 3.17. *For trees of bounded depth there is a routing labeling scheme with labels of length* $\log n + \mathcal{O}(\log \log n)$ *bits.*

Proof. (sketch) Only cause of label size exceeding $\log n + \mathcal{O}(\log \log n)$ for the scheme from Section 3.5 is length of $start(u)$, enforced by rounding accumulated over levels. But for trees of bounded depth, as b is always at least 1, segments' lengths in the described scheme can be adjusted to constraint overall roundings to a factor of $\mathcal{O}(2^d)$, where d is the depth of a tree. □

Routing with larger local memory. A labeling scheme provides a symmetric solution for the routing problem while using little space, in which every node of the network has an assigned label and during a routing query the decoder gets just labels of the current node and the destination node. A label of the destination node serves as its address and travels through the network, in the header attached to a packet, and the label of a node is stored in its a local memory as its identifier. Typically, relaxation of this setting is used, allowing nodes to store locally not only a small label, but (much) more bits. In such case, a further decrease in length of a label (address) attached to a packet is possible.

Definition 3.18. *A routing labeling scheme* with local tables *for a family of rooted trees* $\mathcal{T}$ *consists of an encoder and a decoder. The encoder takes a tree* $T \in \mathcal{T}$*, then assigns a label* $\ell(u)$ *and a local table* $\mathsf{local}(u)$ *to every node* $u \in T$*. Edges are numbered with port numbers as in the regular designer-port routing labeling scheme. The decoder receives* $\mathsf{local}(u)$ *and* $\ell(w)$*, such that* $u, w \in T$ *for some* $T \in \mathcal{T}$ *and* $u \neq w$*. The decoder should return the port number corresponding to the first edge on the path from* u *to* w*. We are interested in minimising the maximum length of a label, and minimising the maximum length of the local table as a secondary objective.*

Our main result gives a routing labeling scheme with both local tables and labels of size $\log n + \mathcal{O}((\log \log n)^2)$. Here we give two other trade-offs further decreasing size of a label at the expense of increasing local space used.

Note that routing labeling scheme from Section 3.4 possibly makes use of every part of label $\ell(u)$, but checks only the value of $\mathsf{start}(w)$ from $\ell(w)$. Thus we can construct a routing labeling

scheme with local tables by first running the encoder described in Section 3.4 and then, for every node u, setting $\mathsf{local}(u)$ to be returned label of u and storing as shrunk $\ell(u)$ only $\mathsf{start}(u)$. Recall that this labeling scheme does not use hiding information by compressing trailing zeroes. Additionally, there are no parts of the new label to be separated, so we can just set $\ell(u) = \mathsf{start}(u)$. This way, setting $b = \log n / \log\log n$, we achieve $\mathsf{span}(\mathsf{root}(T)) \leq n2^{12\log n/b}$, and the size of rt is $\mathcal{O}(\log n)$.

Corollary 3.19. *There is a routing labeling scheme with local tables having label length bounded by* $\log n + \mathcal{O}(\log\log n)$ *and local tables size bounded by* $\mathcal{O}(\log n)$.

Similarly, setting $b = \log n$ we get the following:

Corollary 3.20. *There is a routing labeling scheme with local tables having label length bounded by* $\log n + \mathcal{O}(1)$ *and local tables size bounded by* $\mathcal{O}(\log n \log\log n)$.

3.7 Lower Bound of $\log n + \Omega(\log\log n)$

We first describe the well-known lower bound for ancestry labeling schemes, as the lower bound for routing labeling schemes is just its minor adjustment. We fix number of nodes, n, and define T_i to be a tree consisting of i paths of roughly equal length hanging off a root.

Firstly, observe that all labels must be different, because every node is its own ancestor, and $u = w$ is the only case where the answer for both queries $(\ell(u), \ell(w))$ and $(\ell(w), \ell(u))$ is positive. We want to consider the assignment of labels by the encoder for $T_1, T_2, \ldots$, keeping track of the number of distinct labels that have already been used on some nodes. For T_1 there is one path, and every node has a different label. We count and mark all these labels. For T_2, we want to argue that already marked labels may appear only on one of the two paths. Indeed, assume that two marked (and necessarily different) labels ℓ_1 and ℓ_2 appeared on different paths. But as they are marked, they appeared on one path in T_1, which means that the decoder has to answer positively ancestry query for these two labels, in one or another order. This is a contradiction, as the decoder must not answer positively ancestry query for two nodes from different paths. Thus, one of the paths in T_2 must have been assigned only unmarked labels. We count and mark all these new labels. In general, considering T_i, there were labels from $i-1$ paths counted and marked in the previous trees. This means that the encoder has to assign unmarked labels to at least one of the paths in T_i. We choose one such a path, then count and mark labels from it. At the end there are $\sum_{i=1}^{n} \lfloor n/i \rfloor = \Omega(n \log n)$ different marked labels, which leads to the desired lower bound on length of a label.

To move to a lower bound for routing, we just note that in considered trees every node except the root has degree of at most 2. Thus, we can artificially augment the labels created by a labeling scheme for routing in such trees, adding a single bit denoting which of at most two ports leads to a parent of a node. This effectively allows answering ancestry queries, as u is an ancestor of w iff decoder is answering a query $(\ell(u), \ell(w))$ with a port not leading to the parent of u. If the decoder for a routing labeling scheme does not have to provide answer to queries in

form of (u, u), we need some other way to enforce uniqueness of labels on paths. It can be done by attaching a single dummy node to every node on a path. Finally, we notice that we could use trees with degree two, by expanding high-degree root into a binary tree, adding only a linear number of new vertices. This gives us the following:

Theorem 3.21. *Any labeling scheme for routing in trees on n nodes, even with degree bounded by 2, needs labels consisting of $\log n + \Omega(\log \log n)$ bits.*

3.8 Conclusions

We have designed a labeling scheme for routing in trees on n nodes with labels of length $\log n + \mathcal{O}((\log \log n)^2)$. While this is a major step in determining the asymptotically correct second-order term, we still have a gap between the (simple) lower bound of $\log n + \Omega(\log \log n)$ and our upper bound of $\log n + \mathcal{O}((\log \log n)^2)$. It does not seem possible to bridge this gap by working in our framework, as it seems to suffer from a multiplicative penalty of $\log \log n'$, for every $n' = n, n^{1/2}, n^{1/4}, \ldots$, in the second-order term. It seems to us that routing is harder than ancestry, and so $\Omega((\log \log n)^2)$ might be the right answer. However, showing this requires fixing a family of trees that are complicated but at the same time possible to analyse. We have shown that trees of bounded degree or depth admit a scheme with the second-order being $\mathcal{O}(\log \log n)$, which suggests that this family should consists of trees with logarithmic depth and many nodes having large "entropy" of (rounded) sizes of their subtrees. This seems hard to quantify and analyze, and the bound of $\log n + \mathcal{O}(\log \log n)$ in the model with local tables of size $\mathcal{O}(\log n)$ shows that some natural approaches cannot work.

3.9 Appendix: Constant Time Query

Technical aspects of roundings. In our labeling schemes we frequently used rounding up to $\lfloor 2^{t/b} \rfloor$, for integer t and some fixed integer parameter b. Numbers from range $[0, n-1]$ can be stored, after such rounding, on $\log(b \log n) = \log b + \log \log n$ bits. This is convenient for the analysis, but not necessarily so for implementing operations such as taking powers or logarithms. As a substitute, one could use a very similar rounding, which we call a *two-parts representation*, that also turns out to be more useful if we want to implement queries in constant time. Instead of rounding up given number $x \in [0, n-1]$ to the power of $2^{1/b}$, we can store the first $\log b + 2$ most significant bits of x and also the length of its binary representation on $\log \log n$ bits. This way only $\log b + 2 + \log \log n$ bits are used, and bits of x after the $(\log b + 2)$-th one are lost (we round up by just adding 1 to the stored most significant bits), which results in rounding by a factor of $1 + 1/2b$. As $1 + 1/2b \leq e^{1/2b} < 2^{1/b}$, this is not more than in the previous method. But now, two-parts representation operates only on powers and logarithms in base two. This representation is basically floating-point numbers with precision parameterized by b, used for integers only, and always rounding up.

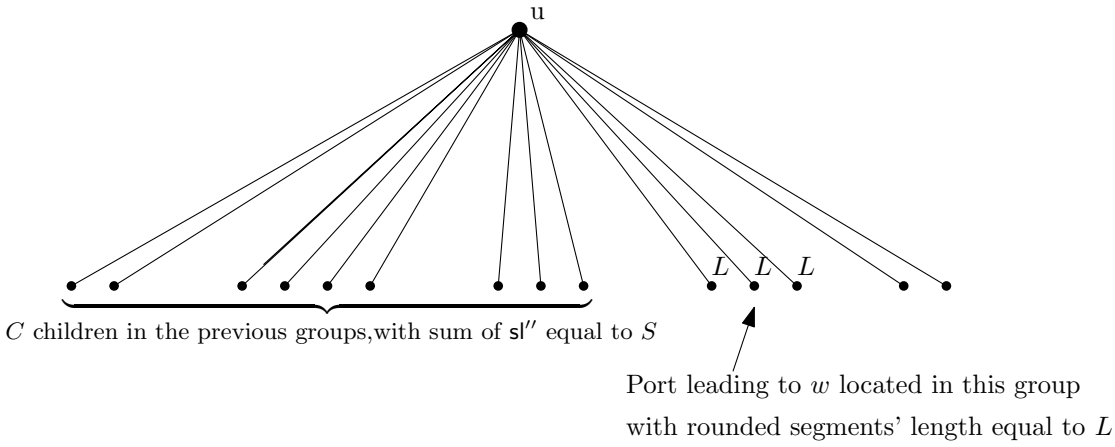


Figure 3.4: Information needed during a query: locating the correct group, the rounded size of a reserved segment there, and prefix sums of the number of ports and segment sizes in all previous groups. $\mathsf{start}(w) = \mathsf{start}(u) + S + kL + x$, and port $C + k + 2$ is chosen.

Finding port number with prefix sums. This appendix gives an overview of a routing labeling scheme with a constant-time decoder, providing description of a construction and sketched proofs. The additional ingredient is a data structure for storing some information concerning prefixes of children. Our goal will be to prove the following theorem:

Theorem 3.22. *There exists a labeling scheme for routing in trees on n nodes with labels of length $\log n + \mathcal{O}((\log\log n)^3)$ bits, the decoder answering queries in constant time, and the encoder working in polynomial time.*

In Section 3.5, we had to decode the routing table bit by bit, which prevented us from answering queries in constant time. Recall that we only need to consider the case that the decoder learned from $\ell(u)$ and $\ell(w)$ that w is in a subtree rooted at a light child u. Let v_w be the child of u which is also an ancestor of w. Children of u were divided into groups, and children in the same group have equal reserved segment length (sl''). Now four pieces of information are needed to locate the right port leading to v_w from u:

- In which group of children of u v_w is; only as leading to the following values, the number of a group in itself is not interesting.
- L, rounded segment length in this group.
- S, prefix sum of reserved segments of all children in the prior groups.
- C, the number of all children in the prior groups.

Consult Figure 3.4. Then, as port 0 leads to the parent of u and 1 to its heavy child, the decoder needs to answer with a port numbered $1 + C + \lceil(\mathsf{start}(w) - \mathsf{start}(u) - S)/L\rceil$.

If a scheme is able to store prefix sums for lengths of reserved segments and sizes of groups (measured in the number of children), then the decoder can use either binary search for locating

the correct port or better use some static data structure allowing for constant time rank queries, where for a fixed set $\mathsf{rank}(x)$ returns number of elements from this set less than x. We use a structure based on a parallel comparison method from [FW93] to be described later.

Lemma 3.23. *For any positive integers w' and k such that $w'k = \mathcal{O}(w)$, it is possible to store k numbers of binary length at most w' on $\mathcal{O}(w'k)$ bits, while supporting* rank *queries in constant time. Note that we might be using space smaller than one full machine word, so just its fragment.*

We will refer to this structure as a *dictionary*. Effectively, we use rank queries just to locate predecessors in constant time.

Creating prefix sums. The decoder from Section 3.5, in some sense, used just one bit in rt for every group. Simple prefix sums need to be longer, thus slight increase in the label length. Sums should not be too large, though — we use $\mathcal{O}(\log\log n)$ bits for each. This will introduce yet another rounding for span of nodes, which needs to be analysed.

Assume in advance that we are able to keep the size of the reserved segment for a node u_i within $\mathcal{O}(|T_{u_i}|\mathsf{level}(u_i)2^{\mathcal{O}((\log\log \mathsf{lw}(u_i))^3)})$ bound, so besides $\mathsf{start}(u_i)$ and $\mathsf{bound}(u_i)$ the size of binary representation of numbers in use is $\mathcal{O}(\log \mathsf{lw}(u_i))$, as they depend only on the total size of subtrees of light children of u_i. Then, we need to store the (rounded) prefix sums $\sum_{j=1}^{k} \mathsf{sl}''(v_{i,j})$ — and $\mathcal{O}(b\log\log \mathsf{lw}(u_i))$ of them, according to the division into groups; recall there are $\mathcal{O}(b\log\log \mathsf{lw}(u_i))$ groups. In other words, we want to ensure that the sum of the lengths of reserved segments for children from the first group is rounded, then that the sum of the lengths of reserved segments for children from the first two groups is rounded, and so on. Note that we round prefix sums, not individual segment's length, and increase of the length of a single segment might be very significant — in the worst case rounding is applied many times to almost the whole segment reserved for u_1.

As we plan to stick to rounding by a factor of $2^{\mathcal{O}(1/b)}$ per level, we can afford rounding by a factor of $2^{\mathcal{O}(1/b^2 \log\log \mathsf{lw}(u_i))}$ for every created prefix sum. Thus, these prefix sums can be stored in rounded two-parts representation, in the form of $\mathcal{O}(\log b + \log\log\log \mathsf{lw}(u_i))$ most significant bits and then $\mathcal{O}(\log\log \mathsf{lw}(u_i))$ bits encoding the number of following zeroes. With this many bits used for every prefix sum, so for every group, whole description of rt takes $\mathcal{O}(b(\log\log \mathsf{lw}(u_i))^2 + b\log b\log\log \mathsf{lw}(u_i))$ bits, when Lemma 3.23 is used. Note that we have two-parts representation of numbers, with some most significant bits and then a number of following zeroes, while a dictionary from Lemma 3.23 operates just on the usual binary representation of numbers. We will deal with this small issue later, for now, let us assume it can be overcome.

By setting $b = \log \mathsf{lw}(u_i)/c(\log\log \mathsf{lw}(u_i))^2$ for some constant c, we would fit the whole $\mathsf{rt}(u_i)$ in the $\log \mathsf{lw}(u_i)$ available bits created during encoding by storing the trailing zeroes of $\mathsf{start}(u_i)$ separately. Then, as b is divided by an additional $\log\log \mathsf{lw}(u_i)$ (recall that in Section 3.5 $b = \log \mathsf{lw}(u_i)/c\log\log \mathsf{lw}(u_i)$ was used), it can be checked with similar inequalities bounding sl as before, only taking into the account the constant number of additional $2^{1/b}$ factors for every level from creating prefix sums, that we will get $\log n + \mathcal{O}((\log\log n)^3)$ bits as the length of a label.

But, as additional information, we need prefix sums on groups' sizes to know how many ports need to be skipped. Analogically, we can afford rounding by a factor of $2^{\mathcal{O}(1/b^2 \log\log \mathsf{lw}(u_i))}$ for every group. Therefore, rounding prefix sum at every group by storing $\mathcal{O}(\log b + \log\log\log \mathsf{lw}(u_i))$ most significant bits and then the number of following zeroes is going to be sufficient — total increase in reserved segments' length will be just a factor of $2^{\mathcal{O}(1/b)}$, and such a prefix sum takes $\mathcal{O}(\log b + \log\log \mathsf{lw}(u_i))$ bits to be stored. As we are increasing the sizes of groups, we do not want to add dummy nodes to any group but the last one, so rounding of the number of children is done by increasing the reserved segments for a necessary number of children in the following groups. More precisely, after rounding the size of a group g up to z, until g consists of exactly z light children of u_i, the reserved segment of a single child in any of the further groups is artificially increased to the size of segments in g, and then this child is moved to g. This way, dummy nodes will have to be used for the last group only, which cannot borrow children from the further groups, but in this case it is not an issue, as the decoder will never answer a query with a port leading to a dummy child inside the last group.

The total increase in $\mathsf{sl}(u_i)$ is indeed by a factor of $2^{\mathcal{O}(1/b)}$, as the already existing groups were increased by at most this value. These prefix sums are built in parallel to prefix sums of segment lengths, so during processing of a single group first we round prefix sum on groups' sizes, then prefix sum on segment lengths, then proceed to the next group.

Finally, together with the prefix sum for the number of children, we also need to store the size of reserved segments in a given group. Rounded by $2^{\mathcal{O}(1/b)}$, it needs $\mathcal{O}(\log b + \log\log \mathsf{lw}(u_i))$ bits to be stored in two-parts representation.

To sum up, the entry for a group g_j consists of three elements:

- As a key for the dictionary, value S_j being the rounded sum of all segments used for groups $1, \ldots, (j-1)$. Then the reserved segment of the first child in g_j is starting at $\mathsf{start}(u_i) + S_j + 1$. S_j is stored in rounded two-parts representation, with $\mathcal{O}(\log b)$ most significant bits saved, which takes space of $\mathcal{O}(\log b + \log\log \mathsf{lw}(u_i))$ bits.
- C_j, number of children of u_i already processed in the previous groups. Through moving children between groups this number was made to have just $\mathcal{O}(\log b)$ significant bits, and then all zeroes, thus taking $\mathcal{O}(\log b + \log\log \mathsf{lw}(u_i))$ bits to store exactly.
- L_j, the size of the reserved segments in g_j, also rounded and stored on $\mathcal{O}(\log b + \log\log \mathsf{lw}(u_i))$ bits.

These three values enable locating the sought port number in a way described at the beginning. We make them accessible through rank queries on S_j values — the answer to this query is interpreted as an index in an array, and there three values are stored together. As any of these values for a single group takes $\mathcal{O}(\log b + \log\log \mathsf{lw}(u_i))$ bits, we still can use $b = \log \mathsf{lw}(u_i)/c(\log\log \mathsf{lw}(u_i))^2$ to achieve labels with length $\log n + \mathcal{O}((\log\log n)^3)$ bits.

Now the assignment of the values in the first phase of the encoder, as in Algorithm 3.7, becomes a bit more involved. In a given node, sl values of all light children are gathered, then

Algorithm 3.9 High-level description of the first phase of the encoder for a labeling scheme achieving a constant time query.

function ASSIGN-$\mathsf{sl}(P)$
  Input: heavy path $P = u_1, \ldots, u_p$. c is a fixed constant.
  Requirement: all nodes in the subtrees hanging off P are already processed

  for $i = 1 \ldots p$ **do**
    $b_i \leftarrow \log \mathsf{lw}(u_i)/c(\log\log \mathsf{lw}(u_i))^2$
    $C \leftarrow$ CONSTRUCT-CLASSES(u_i, b_i)
    for $j = 1 \ldots \mathsf{deg}(u_i) - 1$ **do**
      Set $\mathsf{sl}'(v_{i,j})$ to be the boundary value of the class of $v_{i,j}$ in C
    $G \leftarrow$ CONSTRUCT-GROUPS(u_i, b_i)
    for $j = 1 \ldots \mathsf{deg}(u_i) - 1$ **do**
      Set $\mathsf{sl}''(v_{i,j})$ to be the size of the largest child in the group of $v_{i,j}$ in G
    for $j = 1 \ldots |G|$ **do**
      Let g_j be j-th group
      Set rounding precision to $\Theta(\log b_i)$ most significant bits
      Round to L_j the length of segments in g_j
      Round to C_{j+1} the prefix sum on sizes of groups up to j
      Increase size of g_j if necessary
      Round to S_{j+1} the prefix sum on the length of segments in groups up to j
    Create the dictionary on S_j values for u_i
    Create an array with tuples (S_j, C_j, L_j)
  Set $\mathsf{sl}(u_1)$ to an appropriate upper bound value

rounding in classes and groups happens. Then the groups are considered one by one. Firstly, we round the length of the reserved segment for a group g_j (this corresponds to L_j). Secondly, the size of a group might be increased, by moving some children from the further groups (with increasing sizes of their reserved segments to L_j). This is to ensure that the prefix sum on groups' sizes (C_{j+1}) can be stored exactly on small number of bits. Then a recursive assignment of the values in the subtrees of children from this group happens. At the end the prefix sum on segments' lengths is rounded (this corresponds to S_{j+1}). Observe that gaps between reserved segments are introduced — some additional intervals of possible values for IDs are skipped when a prefix sum is rounded up. S_j prefix sums are counting also these gaps, not only 'useful' reserved segments. A very high-level pseudocode for this phase is presented in Algorithm 3.9. The accumulator in the second phase of the encoder is increased according to the sequence of S_j and light weights of nodes.

Storing prefix sums. First, we are going to prove Lemma 3.23.

Proof. (Lemma 3.23) We just use the method from the paper of Fredman and Willard about

fusion trees [FW93]. Recall that we have a collection of k numbers $a_1 \leq \ldots \leq a_k$, each of binary length w' (possibly with leading zeroes). We store them explicitly separated with ones, as a binary string s of form $s = 1 \circ a_1 \circ 1 \circ a_2 \circ \ldots \circ 1 \circ a_k$. For a query $\mathsf{rank}(x)$, first by computing the most significant bit in constant time we check whether binary length of x is longer than w'. In that case, x is larger than every a_i. Now we assume that x has length of w' bits, possibly with leading zeroes. We multiply x by $(0^{w'} \circ 1)^k$ to get $(0 \circ x)^k$, that is k blocks of x separated with additional single zeroes. Then we subtract this value from s. Observe that this let us subtract $0 \circ x$ from each $1 \circ a_i$, with no interference from carrying. First bit of block i of length $w'+1$ will be 1 if and only if $x \leq a_i$. After subtracting, we AND the result with $(1 \circ 0^{w'})^k$ to get only this first bit in each block. The numbers were sorted, so now we just need to find the first 'interesting' bit which flipped to 0 from 1. Finding the most significant bit can be implemented with a constant number of standard operations, but we note that for this monotonic case, equivalently, we need to find the number of 1-bits. To achieve this in our case, we can multiply by $(0^{w'} \circ 1)^k$ — all the bits set to 1 will collide in the first block of the result, without a carry from other blocks for values of w' and k used later, so we can find the sum of ones by looking at this first block. If we subtract the resulting sum from k, we get $\mathsf{rank}(x)$.

Note that value of $(0^{w'} \circ 1)^k$ can be produced in constant time by computing $(1 \ll (k(w'+1)))/((1 \ll (w'+1)) - 1)$, using C-like $\ll$ notation for a shift to the left. □

Finally, we should go back to the issue with using a dictionary while having numbers in rounded two-parts representation. In the following, by light abuse of notation, we use s as either a number or a binary string representing this number, possibly padded with some leading zeroes. Assume that we have two numbers x_1 and x_2 in two-parts representation, with the same number of most significant bits stored. Let $x_1 = m_1 2^{e_1}$ and $x_2 = m_2 2^{e_2}$, with $|m_1| = |m_2|$. Now, if the shorter of binary strings e_1 and e_2 is padded with leading zeroes to have the same length as the longer one, we can compare x_1 and x_2 by comparing concatenated parts of their representation:

Fact 3.24. *Assuming $|m_1| = |m_2|$ and $|e_1| = |e_2|$, $x_1 > x_2$ iff $e_1 \circ m_1 > e_2 \circ m_2$, as firstly exponents are compared, and only when they are equal stored most significant bits are compared.*

Recall that created prefix sums have the same number of stored most significant bits, and we can add leading zeroes to the exponents to make their lengths equal, which is also necessary for placing them in a dictionary. Therefore, we can use a dictionary from Lemma 3.23 to store created prefix sums and answer rank queries in constant time. The answer to a query is used as an index in the array storing tuples (S, C, L).

Let us sum up the process of obtaining the sought prefix sums:

1. The decoder computes $q = \mathsf{start}(w) - \mathsf{start}(u)$ and translates it into two-parts representation, by changing all bits after fixed number of most significant ones to zeroes, then constructs a string of fixed length by concatenating these two parts and padding with leading zeroes.

2. The resulting number is used in a rank query to the dictionary.

3. The acquired rank is used to access an index in an array, where values of L, C, S are stored. With them, the decoder can answer a query.

All steps can be executed in constant time. Overall, both the dictionary and the array storing prefix sums take $\mathcal{O}(\log \mathsf{lw}(u_i))$ bits, with $\mathcal{O}(\log \mathsf{lw}(u_i)/\log\log \mathsf{lw}(u_i))$ elements of size $\mathcal{O}(\log\log \mathsf{lw}(u_i)))$. For big enough constant c in the formula for b they can be fit into $\log \mathsf{lw}(u_i)$ bits of space available in the label of every node u_i. Recall that we use $b = \log \mathsf{lw}(u_i)/c(\log\log \mathsf{lw}(u_i))^2$, thus increasing constant c decreases number of groups (there are $\mathcal{O}(b \log\log \mathsf{lw}(u_i))$ of them), which in turn decreases size of structures used, allowing us to fit them in $\log \mathsf{lw}(u_i)$ bits. As a side effect, though, b decreases and the rounding becomes more rough, increasing the range of the numbers used for IDs. Namely, the constant in the exponent in $2^{\mathcal{O}((\log\log \mathsf{lw}(u_i))^3)}$ factor increases. But this affects only the constant in the second-order term of the bound on the length of a label, and we only use the property that the binary length of all numbers stored in our structures is $\mathcal{O}(\log \mathsf{lw}(u_i))$ anyway.

Chapter 4

Simpler Adjacency Labeling for Planar Graphs

In this chapter, we prove the following:

Theorem 4.1. *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.*

The result is simplification of [DEG+21], with a slight improvement in size and decoding time. Usually, encoding time is not that relevant for the labeling schemes. Here, the encoder works in polynomial time.

By the standard connection between adjacency labeling schemes and induced-universal graphs (see for example [KNR92]) we also have that:

Corollary 4.2. *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.*

Beyond minimizing the number of vertices in a universal graph, sometimes an additional goal is to reduce the number of edges, as done for example in [AC07, AA02] for the case of graphs of bounded degree. As stated in Corollary 4.2, the adjacency labeling from [DEG+21] leads directly to an induced-universal graph with $n^{1+o(1)}$ vertices, but that graph can have n^2 or more edges. Very recently, Esperet, Joret and Morin [EJM20] further refined previously existing labeling scheme to obtain an induced-universal graph with just $n^{1+o(1)}$ edges:

Theorem 4.3. *(from [EJM20]) For every n, there is a 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 show that this is also possible with our approach, in fact arguably simpler.

4.1 Preliminaries

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

Lemma 4.4. *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 $2^r \leq \max(1, w_i/2)$, 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 subtree is $c(w_i)$. The length of $c(w_i)$ is at most $\log n + 1 - \log(w_i/4) \leq \log(n/w_i) + 3$. All chosen subtrees are disjoint and $c(w_i)$ are lexicographically sorted.

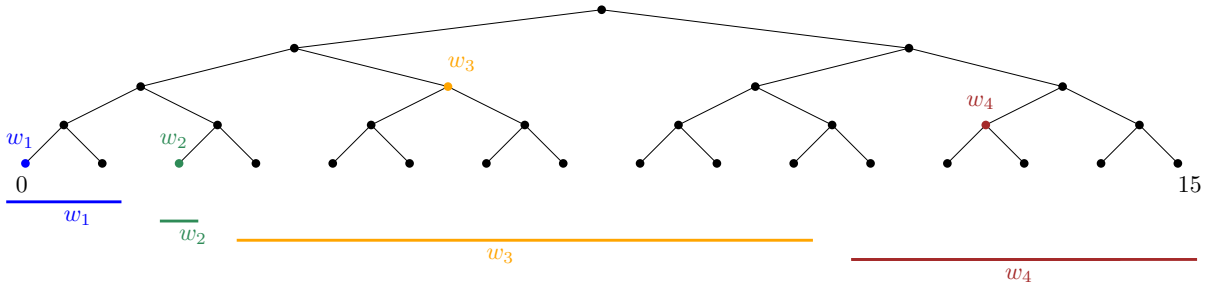


Figure 4.1: Assignment of weighted prefix-free codes for $w_1 = 2, w_2 = 1, w_3 = 8, w_4 = 5$. We have $c(w_2) = 0010$, $c(w_3) = 01$, $\mathsf{succlabel}(w_2) = \mathsf{predlabel}(w_3) = 1$.

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 4.1 for an example. □

Fractional cascading. We will need a specific variant of the fractional cascading technique [CG86]. 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 4.5. *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. $\forall_{i=1}^{m} |V_i| \leq n$,
3. $\forall_{i=1}^{m} S_i \subseteq V_i$,
4. *for any* $t \leq k$ *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* $t \leq k$ *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 [DEG+21]. 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 $1 \leq i < m$ in increasing order, say ordered elements of U_i are $U_i = \{u_0, u_1, \ldots\}$, then assign $U_{i+1} = S_{i+1} \cup \{u_j \in U_i : j \mod k \neq 0\}$. 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 $|U_{i+1}| \leq |S_{i+1}| + \lfloor (1-1/k)|U_i| \rfloor$, and so by induction we have $|U_{i+1}| \leq \sum_{j=1}^{i+1} (1-1/k)^{i+1-j} |S_j|$. Therefore:

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

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. □

We note that it is possible to obtain sets $V_1, \ldots, V_m$ 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.

4.2 Framework

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 4.4. Throughout the chapter, 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 4.2 for a small example.

Definition 4.6. *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:*

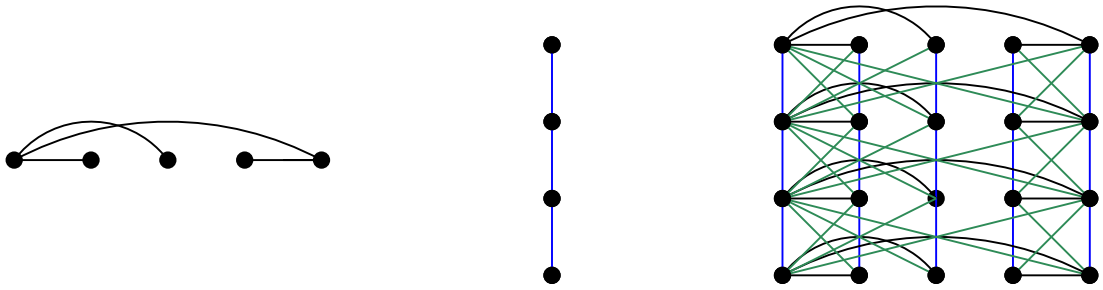


Figure 4.2: A strong product of a tree and a path.

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 [DJM+20] by Dujmović, Joret, Micek, Morin, Ueckerdt, and Wood:

Theorem 4.7. *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' [UWY21]. For the adjacency labeling scheme, this stronger formulation affects only constants in the second-order term.

Theorem 4.7 is the crucial tool used in [DEG+21] and in this chapter. Say $P = (p_1, \ldots, p_m)$, 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 observed in Figure 4.2, 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 [DEG+21] 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.

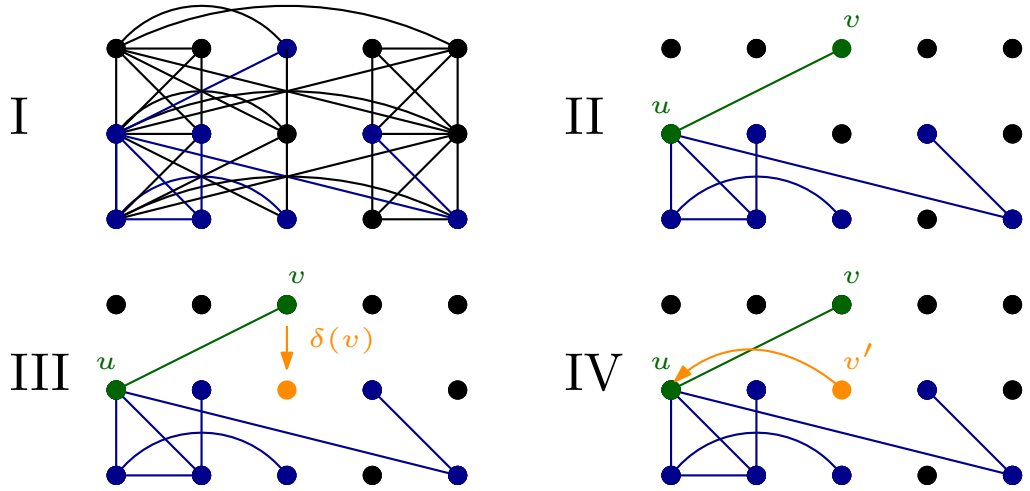


Figure 4.3: I. We consider the blue subgraph of a strong product of a path and a graph of bounded treewidth.
II. Say that we want to check for the existence of the green edge between v and u. First, we check $\mathsf{row}(v)$ and $\mathsf{row}(u)$ and see that a row of vertex u is the next one after 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 4.3 and its description. We can use weighted prefix-free codes from Lemma 4.4 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 4.5 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.

4.3 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 [DEG+21] with B-trees as described here allows us to simplify the adjacency labeling scheme.

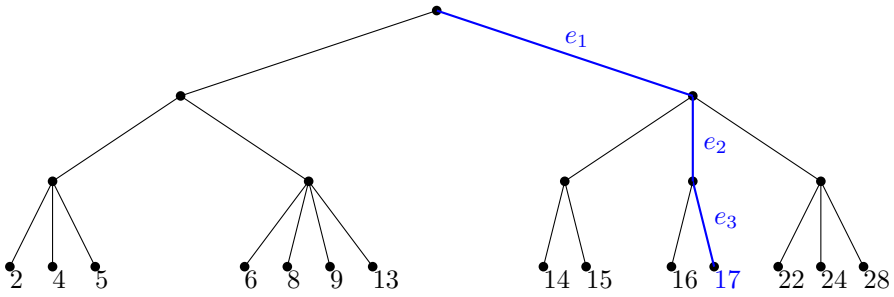


Figure 4.4: Example of B-tree of height 3. We use it to encode paths to the leaves, not as a dictionary, so internal nodes store no keys. The path to leaf with value 17 is stored simply as a concatenation of prefix-free codes of edges e_1, e_2 and e_3.

We use weight-balanced B-trees. Similarly to the use of trees for example in [AV96], 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 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 4.5, 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 4.8. *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 4.9. *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 4.10. *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 $V_{i+1} \setminus V_i$ and delete values from $V_i \setminus V_{i+1}$. 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 4.1 for a pseudocode.

Algorithm 4.1 A sequence of B-trees with a cyclic rebalancing.

function COMPUTE-LABELS($V_1, \ldots, V_m$, a)
  Input: augmented sets $V_1, \ldots, V_m$, 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 $i = 2 \ldots m$ **do**
    Delete leaves with values from $V_{i-1} \setminus V_i$
    Insert leaves with values from $V_i \setminus V_{i-1}$
    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 $v \in V_{i-1} \cap V_i$

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 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 4.5 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 $V_1, V_2, \ldots, V_m$ are obtained by applying Lemma 4.5 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:

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

Thus, by using Lemma 4.5 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 4.11. *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.

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 4.2 for a pseudocode.

Algorithm 4.2 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 ← S\{s}
16:         if S ≠ ∅ then
17:             Create a new rightmost child of v with all ordered subtrees from S as children
18:         if v is the root with the only child u then
19:             Delete v, make u the new root
20:     return T
```

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 4.2 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 4.12. *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 $\mathsf{path}(d, T_{i+1})$.*

4.4 Labels for a Strong Product

In this section, we put together the framework presented in Section 4.2 with encodings of paths and transition labels on B-trees from Section 4.3, to obtain the final adjacency labeling scheme.

Note that not much here is changed from the previous work [DEG+21], but we present most of the details for completeness.

4.4.1 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 [DEG+21]. 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'. 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. An 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 4.13. *For any t-tree H' with m vertices, there exists a function* interval *mapping $V(H')$ into intervals of integers such that:*

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

Proof Sketch. 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 $v \in H'$ 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 T by $\mathsf{lca}(v, T)$. We have the following:

Lemma 4.14. *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 4.13. 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. □

Recall that in T we store all right endpoints of intervals $\mathsf{interval}(u)$ for $u \in H'$. Proposition 4.13 impose strict restrictions on positions of nodes assigned to vertices from $C_{H'}(v)$:

Lemma 4.15. *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 4.5 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

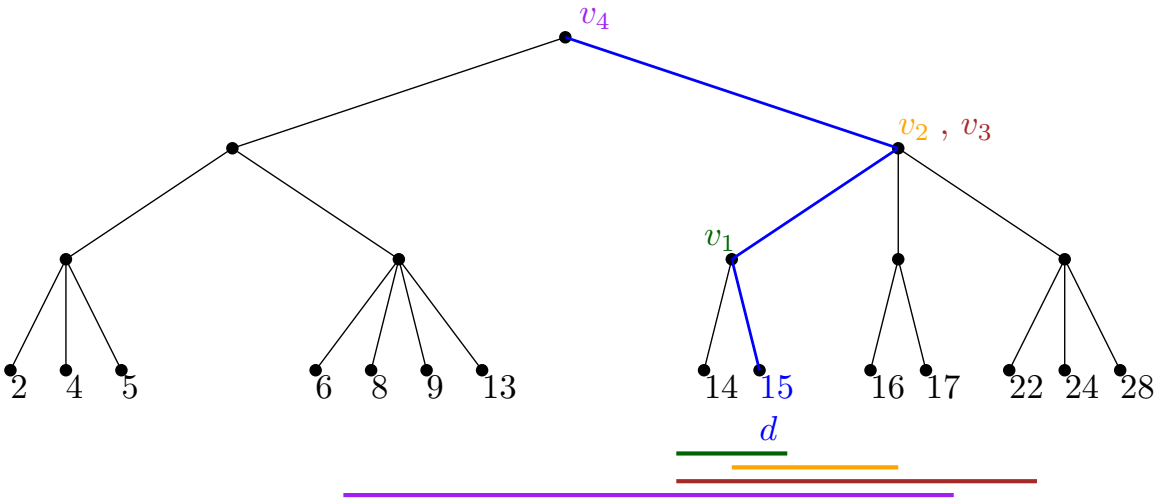


Figure 4.5: B-tree T representing H', with $C_{H'}(v_2) = \{v_1, v_2, v_3, v_4\}$ and $\mathsf{interval}(v_1) = [14, 15], \mathsf{interval}(v_2) = [15, 16], \mathsf{interval}(v_3) = [14, 22], \mathsf{interval}(v_4) = [8, 17]$. $\mathsf{lca}(v_2, T) = \mathsf{lca}(v_3, T)$. $d = \mathsf{lowest}(v_2, H') = 15$, and all nodes assigned to vertices from $C_{H'}(v_2)$ lie on a single path from the root to the leaf storing value d.

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 4.14.

4.4.2 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', 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 4.13 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 4.5 and Lemma 4.11 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 4.3.

In order to identify a row of a vertex from $P \boxtimes H'$, we use weighted codes from Lemma 4.4, 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 4.4.1. Say that we want to create a label for a node (v,p_i), for $v \in S_i$. Using Lemma 4.15 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 4.10. 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 4.14 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 $\mathsf{nodeid}(w,T_i)$ of each $w \in C_{H'}(v)$ 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 4.12, 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 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 $\log n + \mathcal{O}(\sqrt{\log n})$.

4.4.3 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 4.1 is an immediate consequence of Theorem 4.7 and the following theorem proved in this section:

Theorem 4.16. *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 4.7 can be found in polynomial time [DJM+20].

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 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.

4.5 Sparse Induced-Universal Graphs

As stated in the introduction, the authors of [EJM20] described how to modify the adjacency labeling scheme from [DEG+21] to obtain a sparse induced-universal graph for planar graphs:

Theorem 4.3. *(from [EJM20]) For every n, there is a universal graph U_n with $n^{1+o(1)}$ vertices and edges that contains every planar graph on n vertices as an induced subgraph.*

In this section, we briefly repeat their reasoning for our modified scheme which uses B-trees. Their methods can be applied in a straightforward way, while one part of their proof becomes considerably simpler as we do not need to invoke an additional nontrivial property of graphs with bounded treewidth, which maybe indicate additional benefits of simplifying the technical part of [DEG+21].

Modifying the labeling scheme — overview. In case of our labeling scheme, we can use the perfect ordering to orient the edges in the universal graph created as described. For any adjacent pair of vertices in the universal graph, the decoder answers positively only if it believes that it is given labels of some nodes $v = (v', p_i)$ and $w = (w', p_j)$ of some strong product $H' \boxtimes P$ and $v' \in C_{H'}(w')$ or $w' \in C_{H'}(v')$. Let us orient the edge from the vertex corresponding to v to the vertex corresponding to w in the first case, and in the opposite direction otherwise. Now we would like to bound in-degree of any vertex in the universal graph. It would be useful if having the label $\ell(v)$ we were able to compute the set of at most $n^{o(1)}$ labels containing all possible labels of the vertices in the in-neighborhood of v. To this end, recall the subsection detailing lengths of each part of the label. We can ignore many parts which always have length of $\mathcal{O}(\sqrt{\log n})$, as $2^{\mathcal{O}(\sqrt{\log n})} = n^{o(1)}$. There are only two parts of the label that are possibly longer: the code $\mathsf{row}(v', i)$ of the row in the strong product, and the code $\mathsf{path}(\mathsf{lowest}(v', H'), T_i)$ of a path to the leaf in the B-tree T_i. We will need to modify both to reduce the number of edges in the universal graph, while all the other elements of the adjacency scheme remain generally unchanged.

It is not hard to modify row codes in a way that $\mathsf{row}(w', j)$ can be computed given the label of a vertex v, for adjacent v, w. This will mean that all neighbors of v have only three candidates for this part of their label: they are either in the same row, or in the fixed next or previous rows.

The encodings of paths in B-trees pose a slightly bigger challenge. Recall that from $\ell(v)$ we are able to compute $\mathsf{path}(\mathsf{lca}(w', H'), T_j)$ for all $w' \in C_{H'}(v')$, so any neighbor w of v can have only one of a constant number of possible paths to a node $\mathsf{lca}(w', H')$ assigned to it in a B-tree. The problem is, in $\ell(w)$ we store $\mathsf{path}(\mathsf{lowest}(w', H'), T_j)$, the path to the leaf which contains all nodes assigned to vertices from $C_H(w')$. Such a path might be much longer, and we have no way of computing it just from $\ell(v)$. We will remedy this by changing the assignment of nodes of a B-tree to vertices of H'. See also Figure 4.6.

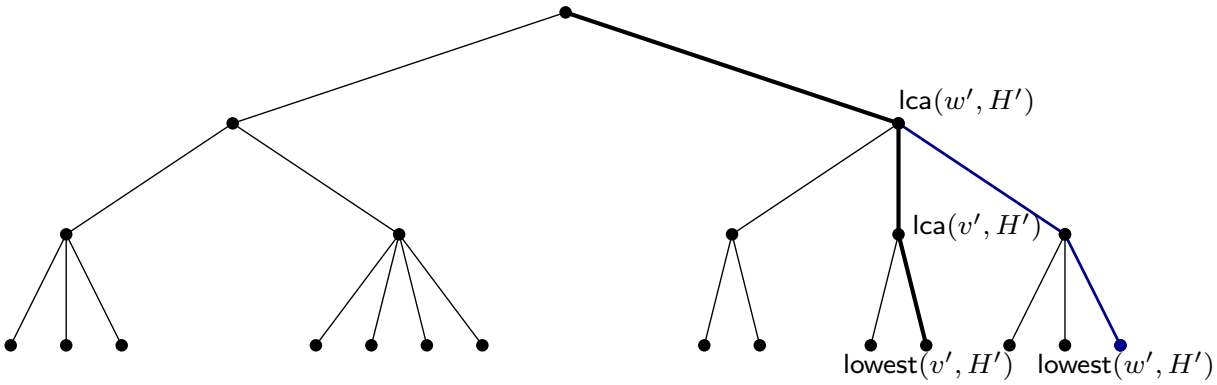


Figure 4.6: For $w' \in C_{H'}(v')$, $\mathsf{path}(\mathsf{lca}(w', T), T)$ can be computed from $\ell(v)$ as a prefix of $\mathsf{path}(\mathsf{lowest}(v', H'), T)$. But in $\ell(w)$ we store $\mathsf{path}(\mathsf{lowest}(w', H'), T)$, which might be arbitrarily longer, and the adjacency labeling scheme from the previous section offers no way of computing this path, which is a problem if we want to bound the number of neighbors of $\ell(v)$.

4.5.1 row Codes

Recall that to assign the row code $\mathsf{row}(v, i)$ for a vertex (v, p_i) we use Lemma 4.4. We also assign short $\mathsf{succlabel}$ and $\mathsf{predlabel}$ to allow for checking for adjacent rows. In fact, we can also easily assign short strings $\mathsf{next}, \mathsf{prev}$ allowing for explicitly computing code of the next row:

Lemma 4.17. *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 $\mathsf{next}(w_i), \mathsf{prev}(w_i)$ of size $\mathcal{O}(\log \log n)$ allowing to compute a code of the adjacent integers in the sequence, that is there is a function f such that $f(c(w_i), \mathsf{prev}(w_i), \mathsf{next}(w_i)) = (c(w_{i-1}), c(w_{i+1}))$.*

Proof. We proceed as in the proof of Lemma 4.4 and compress the final tree. First, we mark all the nodes used for assigning codes (the roots of proper subtrees). Then, delete all the nodes not being ancestors of any marked node. Finally, compress unnecessary edges, that is repeatedly for any node v with the only child u replace v with u. The order of the marked nodes is preserved, and their depth can only decrease, thus by adjusting encodings to the current positions of marked nodes we still get prefix-free codes with lengths bounded as in Lemma 4.4.

Now assume that v_1 and v_2 are marked nodes used to define codes $c(w_i)$ and $c(w_{i+1})$, and u is the LCA of v_1 and v_2. Then it can be seen that the path from u to v_2 consists of firstly one right edge and then only left edges. As $\mathsf{next}(w_i)$ we can store the depth of u and the depth of v_2, which is enough to compute $c(w_{i+1})$ from $c(w_i)$. Similarly, the path from u to v_1 consists of firstly one left edge and then only right edges, and we can store $\mathsf{prev}(w_i)$ accordingly. □

4.5.2 Shortening Stored Paths

The subject of the second change we make are positions of nodes in a B-tree T assigned to the vertices from the clique $C_{H'}(v)$. Recall that by Lemma 4.15 all those nodes lie on a single

root-to-leaf path in T. To decrease the number of edges in the universal graph, we demand more, namely that all those nodes are at depth at most one larger than the depth of a node assigned to v.

Algorithm 4.3 Reassigning nodes in a B-tree by lifting. This is the same as the function `FIXUP` in [EJM20].

```
1: function REASSIGN(r, T)
2:     Input: node r of a B-tree T.
3:
4:     for all w ∈ H' such that lca(w, T) = r and lnode(w, T) is undefined do
5:         lnode(w, T) ← r
6:     for all w ∈ H' such that lnode(w, T) = r do
7:         for all u ∈ C_{H'}(w) do
8:             if lca(u, T) is lower in T than children of r then
9:                 Let v be the child of r being an ancestor of lca(u, T)
10:                lnode(u, T) ← v
11:    for all p child of r do REASSIGN(p, T)
```

We start with the assignment from the previous section, vertex w assigned to the node $\mathsf{lca}(w,T)$ being LCA of leaves with values in its interval $\mathsf{interval}(w)$, but possibly change this assignment to the new one denoted by $\mathsf{lnode}(w,T)$. Initially values $\mathsf{lnode}(w,T)$ are undefined for all w, we will assign them recursively in a top-down manner. Say that we are at node r in T, initially the root. For any vertex w with $\mathsf{lca}(w,T) = r$ and $\mathsf{lnode}(w,T)$ still undefined, we set $\mathsf{lnode}(w,T) = r$. Then we iterate through all vertices w with $\mathsf{lnode}(w,T) = r$, and check positions of $\mathsf{lca}(u,T)$ for all $u \in C_{H'}(w)$. $\mathsf{lca}(u,T)$ can be an ancestor of r, r itself, child of r, or it can be at even deeper level. Only in this last case we lift u by reassigning it to another node in T: denote an ancestor of $\mathsf{lca}(u,T)$ being the child of r by p, then we set $\mathsf{lnode}(u,T) = p$. After we are done with all vertices assigned to node r, we recurse on all children of r. The pseudocode is presented in Algorithm 4.3. Observe that all lnode values will be defined in this process. Since $\mathsf{lnode}(u,T)$ is an ancestor of $\mathsf{lca}(u,T)$, all nodes assigned to vertices from $C_{H'}(w)$ still lie on a single path. Therefore we get the following:

Lemma 4.18. *Assume we are given B-tree T representing t-tree H' with assignment lnode computed with Algorithm 4.3. Then for any vertex $w \in H'$, all nodes from the set $\{\mathsf{lnode}(u,T) : u \in C_{H'}(w)\}$ lie on a single path in T, with the lowest node being $\mathsf{lnode}(w,T)$ or one of its children. We denote this lowest node by $\mathsf{lowest}(w,T)$.*

Recall that in the adjacency scheme we stored a path to the leaf, now we change it to a path to the lowest node $\mathsf{lowest}(w,T)$ assigned to some vertex from the clique $C_{H'}(w)$, but this is a minor modification, as still all nodes of interest lie on the path from the root to $\mathsf{lowest}(w,T)$. What we achieved is that $\mathsf{path}(\mathsf{lowest}(w,T),T)$ is only slightly longer than $\mathsf{path}(\mathsf{lnode}(w,T),T)$, since we need to extend it by at most one edge.

Unfortunately, after reassigning vertices we violate Lemma 4.14, possibly by a large factor, as one vertex can change the assignment of up to t vertices from its clique and this could snowball in the recursion, so in the worst case we could end up with too many vertices assigned to a single node. In [EJM20] that was an issue, as the depth of a used BST was logarithmic, and authors had to use some non-trivial property of t-trees to bound this increase to not be linear in the number of vertices m. But in our case, the depth of a B-tree is $\sqrt{\log m} + \mathcal{O}(1)$. As t is a constant, this means that the accumulated effect of reassignments is not relevant. We have:

Lemma 4.19. *Assume in T representing t-tree H' on m vertices the maximum degree of a node is Δ, and t is a constant. Then any node r in T is assigned $\mathcal{O}(\Delta \log m \cdot t^{\sqrt{\log m}})$ vertices, that is, $|\{v \in H' : \mathsf{lnode}(v, T) = r\}| = \mathcal{O}(\Delta \log m \cdot t^{\sqrt{\log m}})$.*

Proof. The proof is by induction on the depth of nodes. We claim that any node r of depth d is assigned at most $\mathcal{O}(\Delta \log m \cdot t^{d+1})$ vertices. This is true for the root, as Algorithm 4.3 does not assign any new vertices to it, so the bound from Lemma 4.14 holds. Now consider a node r of depth $d+1$ and its parent p. The set $\{v \in H' : \mathsf{lnode}(v, T) = r\}$ might contain all vertices from the set $\{v \in H' : \mathsf{lca}(v, T) = r\}$ of size $\mathcal{O}(\Delta \log m \cdot t)$. Additionally, each vertex from the set $\{v \in H' : \mathsf{lnode}(v, T) = p\}$ might lift up to t vertices to r. Thus, the claim holds by induction and the bound on the depth of T. □

This means that $\mathsf{nodeid}(w)$, a unique identifier of a vertex w among all the vertices assigned to the single node $\mathsf{lnode}(w, T)$, can still be stored on just $\mathcal{O}(\sqrt{\log n})$ bits, for a constant t.

4.5.3 Counting the Edges

After adjusting row encodings and shortening stored paths, we can bound the number of edges in the universal graph, which gives us Theorem 4.3.

Lemma 4.20. *The induced-universal graph created by using the modified adjacency labeling scheme has $n^{1+o(1)}$ edges.*

Proof Sketch. Here we provide only a general idea of the proof, for the exhaustive formal description see [EJM20]. In the following we identify a label s with a vertex of the universal graph. Let us partition elements of any label s into three parts:

- s_1, an encoding row.
- s_2, an encoding path of a path to the node lowest.
- s_3, all the remaining elements, with the total size of $\mathcal{O}(\sqrt{\log n})$ bits.

We orient the edges of the universal graph and argue that for an in-neighborhood of any vertice, the choice for s_1 and s_2 is very limited. s_3 is relatively short, so there are always at most $2^{\mathcal{O}(\sqrt{\log n})}$ possibilities for this part of the label.

Assume that the decoder responds positively to the adjacency query for some labels s and s'. This happens only if the decoder discovers that s and s' are the labels of two vertices in the same row of the strong product or in two adjacent rows. Let us consider the first case, which is the same row, then s_1 and s'_1 are identical. Recall that s stores a constant number of pointers to prefixes of its encoded path, that is s_2, and a positive answer of the decoder requires that a path to lnode stored in one label is exactly one of those interesting prefixes of the second label. We orient an edge in the universal graph from s' to s if s'_2 is one of stored interesting prefixes of s_2, and from s to s' otherwise. We would like to bound the in-degree of s by bounding the number of possible strings s'_2 for s' being in-neighbors of s. By Lemma 4.18, any such s'_2 is one of the interesting prefixes of s_2 extended by at most one edge. As any edge in B-trees is encoded on $\mathcal{O}(\sqrt{\log n})$ bits by our choice of parameters, we have at most $2^{\mathcal{O}(\sqrt{\log n})}$ possible candidates for s'_2.

The second case, when s and s' are the labels of two vertices in the adjacent rows of the strong product, is analogous, though slightly more involved. The only two candidates for s'_1 can be computed exactly using s_1 and codes next or prev from s_3, by Lemma 4.17. With the help of the transition label δ being part of s, again using parts of s we can compute a constant number of interesting prefixes which can be extended by at most one edge to form the set of all possible candidates for s'_2.

As we oriented all the edges and bounded the size of the in-neighborhood of any vertex by $2^{\mathcal{O}(\sqrt{\log n})} = n^{o(1)}$, we obtain that the universal graph is sparse. □

Chapter 5

Optimal Distance Labeling for Permutation Graphs

In this chapter, we prove the following:

Theorem 5.1. *There is a distance labeling scheme for permutation graphs with n vertices using labels of size $3\log n + \mathcal{O}(\log\log n)$ bits. The distance decoder has constant time complexity, and labels can be constructed in polynomial time.*

This result is tight up to the second-order term, due to the lower bound of $3\log n - \mathcal{O}(\log\log n)$ by Bazzaro and Gavoille [BG05].

5.1 Overview and Organisation

In Section 5.2, we present basic definitions and our approach to permutation graphs. Then, in Section 5.3, we build on the methods of Gavoille and Paul [GP08], as well as Bazzaro and Gavoille [BG05] for creating distance labelings of interval and permutation graphs. We can represent a permutation graph as a set of points in the plane, where two points (two vertices) are adjacent when one is above and to the left of the other.

The first thing we need to notice when considering distances is the presence of two *boundaries* in such representation. We say that the top boundary is formed by points with empty (containing no other points) top-left (north-west) quadrant, and bottom boundary by points with empty bottom-right (south-east) quadrant. Points on the boundaries are especially important – it can be seen that for any pair of points, there is a shortest path between them with all internal points of the path being on boundaries. As a set of boundary points forms a bipartite graph, such shortest path strictly alternates between boundaries.

We can also observe that for a point v not on a boundary, there are four boundary points of special interest for it, see Figure 5.1. These are pairs of extreme points on both boundaries from the points adjacent to v. Any shortest path from v to u with distance $d(v,u) > 2$ can have as a second point one of these special points for v, and as a penultimate point one of the special

points for u. We need to handle distances of 1 and 2 separately, but otherwise, this means it is enough to be able to compute distances between boundary points.

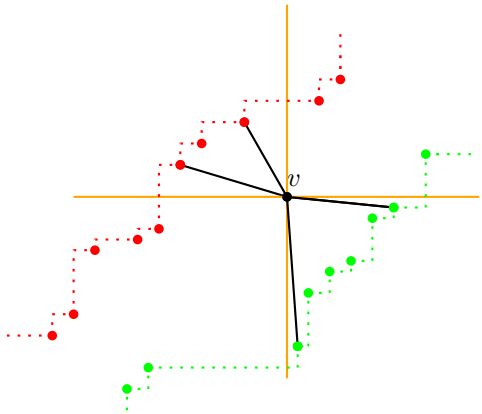


Figure 5.1: Green points form bottom boundary, red points form top boundary. v is not on the boundary, orange lines show its quadrants. For v, there are four points of special interest, extreme neighbours on both boundaries.

If we can build distance labeling for boundary points, and store labels of special points efficiently, we can obtain good distance labelings for permutation graphs. This is possible as boundaries are highly structured, in particular ordered. In [BG05] authors view two boundaries as two proper interval graphs and deal with them using methods from [GP08]. An interval graph is proper when no interval is completely contained by another one. Gavoille and Paul first partition vertices of a proper interval graph into distance *layers*, by distances to the vertex representing the leftmost interval. Let us denote layer number of vertex u by $L(u)$. It can be seen that for any two vertices u, v in interval graph we have either $d(u,v) = |L(u) - L(v)|$ or $d(u,v) = |L(u) - L(v)| + 1$. Then the following lemma is used [GP08]:

Lemma 5.2. *There exists a total ordering λ of vertices of proper interval graph such that given $\lambda(v), \lambda(u)$ and layer numbers $L(u) < L(v)$ for two vertices u, v, we have $d(u,v) = L(v) - L(u)$ if and only if $\lambda(u) > \lambda(v)$.*

In other words, we can assign to each vertex v just two numbers $L(v), \lambda(v)$, and then still be able to determine all exact distances. Going back to permutation graphs, when we view two boundaries as proper interval graphs, it is possible to obtain straightforward distance labeling for permutation graphs using $20 \log n$ bits, where the big constant is due to storing many distance labels for interval graphs completely independently. Then authors are able to reduce the size of labels to $9 \log n$ bits, after eliminating many redundancies in the stored sub-labels.

In this chapter, we show that working with both boundaries at once can yield better results. To do this, we modify the methods of Bazzaro and Gavoille and then carefully remove even more redundancies. First, we partition points on both boundaries into layers, defined by distances from some initial point, see Figure 5.2. As we use distances from a single point to define layers, the distance between any two boundary points is a difference of their layer numbers, or this

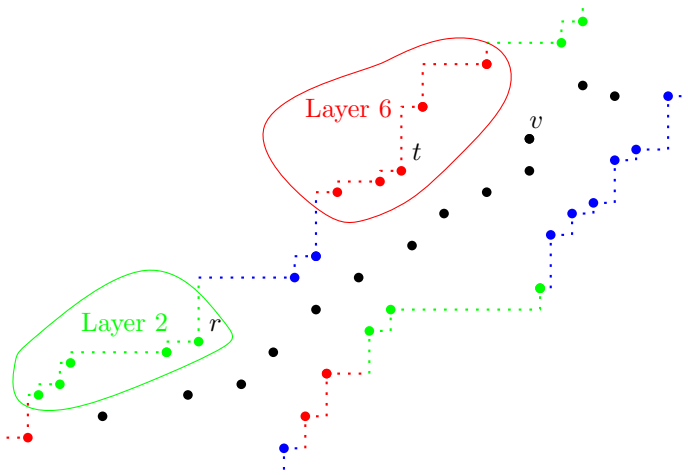


Figure 5.2: Boundary points partitioned into layers. r, t are on the top boundary, but in layers 2 and 6. For any two points in layers a and b, the distance between them is always either $|a-b|$ or $|a-b|+2$; here, $d(r,t)=4$. v is not on the boundary, and any such point can be adjacent to points from at most three different layers.

difference increased by two. It can be shown that again some ordering λ can be used, and storing it takes around $\log n$ bits for each boundary point.

As a single point is adjacent to at most three layers, layer numbers of four special points are easy to store, and we could achieve labeling of length $(2+1+4)\log n + \mathcal{O}(1) = 7\log n + \mathcal{O}(1)$ by storing for each point respectively its 2D coordinates, layer numbers of neighbours packed into $\log n + \mathcal{O}(1)$ bits, and four times λ values for extreme neighbours, in order to compute distances between boundary points. This can be reduced to $5\log n$ by dealing with distances 1 and 2 more carefully, allowing us to not store point coordinates explicitly. All of the above is described in Section 5.3.

After additional analysis and reductions laid out in Section 5.4, we can decrease the size to $3\log n$. This is since, roughly speaking, one can collapse information stored for two pairs of extreme boundary neighbours into just two numbers, due to useful graph and layers properties. More precisely, we can observe that we store excessive information about the set of four extreme neighbours. For vertex v, two extreme right points on both boundaries are used to reach points to the right of v, and extreme left are used to reach points to the left. But we do not need the exact distance between points to the left of v and right extreme points, thus we have some possibility to adjust the stored λ values. Particularly, the main case is when λ value of the right extreme point on the bottom boundary is smaller than λ value of the left extreme point on the top boundary; it turns out that these two values can be equalised and stored as some single value in between the original values. The second pair of extreme points can be dealt with in a similar manner, and then we need to ensure that all of this did not interfere with the correctness of deciding about distances 1 and 2, which are different cases than all distances larger than 2.

5.2 Preliminaries

Permutation graphs. We will use a geometric (or 'grid') representation of permutation graph G_π on n vertices as a set of points with coordinates in $[1, n]$, with point $(i, \pi^{-1}(i))$ for each $i \in [1, n]$. Considering a point p, we always denote its coordinates by $p = (p_x, p_y)$. Top-left quadrant of point p, TL_p, is a subset of points $\{v : v_x < p_x \wedge v_y > p_y\}$ from the graph. Similarly, we have TR_p (top-right), BL_p (bottom-left) and BR_p (bottom-right) quadrants. Two points are adjacent in the graph iff one is in TL or BR quadrant of the other. See Figure 5.3. We have transitivity in a sense that if $w \in \mathsf{BR}_v$ and $u \in \mathsf{BR}_w$, then $u \in \mathsf{BR}_v$; similarly for other quadrants. By distance $d(u, v)$ between two points we will mean distance in the graph.

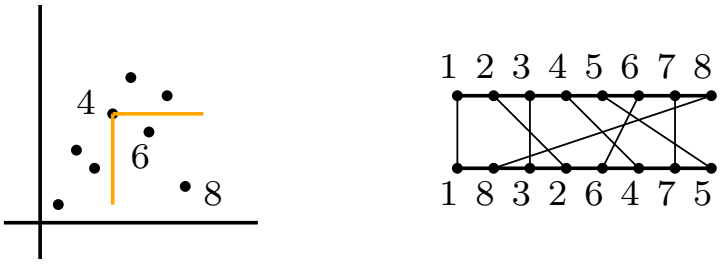


Figure 5.3: Geometric representation of the graph from Figure 1.4, so the permutation is [1,8,3,2,6,4,7,5]. Point $(4, 6)$ is adjacent to $(6, 5)$ and $(8, 2)$, as these two points are in its bottom-right quadrant, while top-left quadrant of $(4, 6)$ is empty.

We will assume the given permutation graph is connected. There are standard ways to enhance labelings to disconnected graphs by adding at most $\mathcal{O}(\log \log n)$ bits to the labels, and we will describe how it can be done after the main theorem. We note that for connected graphs of size at least two, no point could be on both boundaries, as it would be isolated otherwise.

Organization of the labels. The final labels will consist of a constant number of parts. We can store at the beginning of each label a constant number of pointers to the beginning of each of those parts. As the total length of a label will be $\mathcal{O}(\log n)$, pointers add only $\mathcal{O}(\log \log n)$ bits to the labels.

5.3 Scheme of Size $5 \log n$

In this section, we describe how to use boundaries to design distance labeling of size $7 \log n + \mathcal{O}(1)$, and then how to refine it to reach $5 \log n + \mathcal{O}(1)$.

5.3.1 Properties of Boundaries

For a set of points S, we have its top boundary defined as a subset of points from S whose top-left quadrants are empty, and bottom boundary as a subset of points whose bottom-right quadrants are empty. See Figure 5.4. Observe that points on boundaries are ordered, that is, for

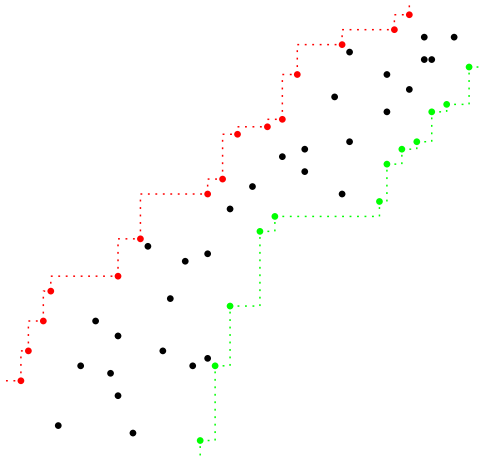


Figure 5.4: Green points are on the bottom boundary, red points are on the top boundary.

u and v on the same boundary, either $u_x > v_x$ and $u_y > v_y$, or $u_x < v_x$ and $u_y < v_y$. We use $<$ to denote this relation on boundary points.

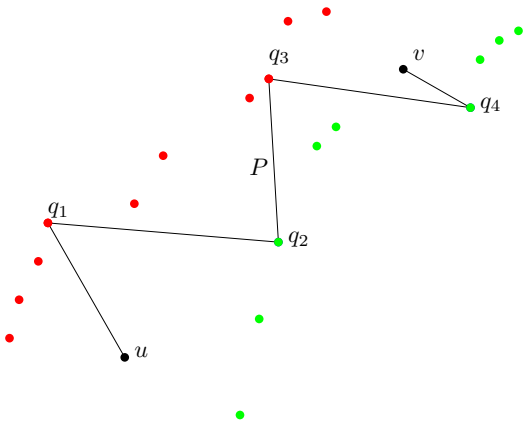


Figure 5.5: Path between two points alternating between top and bottom boundaries. It is not beneficial to move through points not on boundaries.

Boundaries are particularly useful when considering distances between points:

Property 5.3. *For any two points u, v at distance d, there is a path $P = (u = q_0, q_1, q_2, \ldots, q_d = v)$ of length d such that all points except possibly u, v are on alternating boundaries.*

Proof. Take any shortest path P' and any adjacent q_i and q_{i+1} on P', assume without loss of generality that $q_{i+1} \in \mathsf{TL}_{q_i}$. Suppose that q_{i+1} is not on the top boundary. We either have $q_{i+2} \in \mathsf{TL}_{q_i+1}$ or $q_{i+2} \in \mathsf{BR}_{q_i+1}$. Note that by transitivity if $q_{i+2} \in \mathsf{TL}_{q_{i+1}}$, then $q_{i+2} \in \mathsf{TL}_{q_i}$ and we could have a shorter path by removing q_{i+1}. Thus, assume $q_{i+2} \in \mathsf{BR}_{q_{i+1}}$. If q_{i+1} is not on the top boundary, then by definition there exists a boundary point $q' \in \mathsf{TL}_{q_{i+1}}$, and it must be that $q_{i+2} \in \mathsf{BR}_{q'}$. This means that we could replace q_{i+1} by q', increasing the number of points from the path lying on the boundary, and then repeat the argument. Therefore, we have that all points except the first and last ones can always lie on boundaries. See Figure 5.5 for an

illustration. These must be alternating boundaries, as by definition no two points on the same boundary are adjacent. □

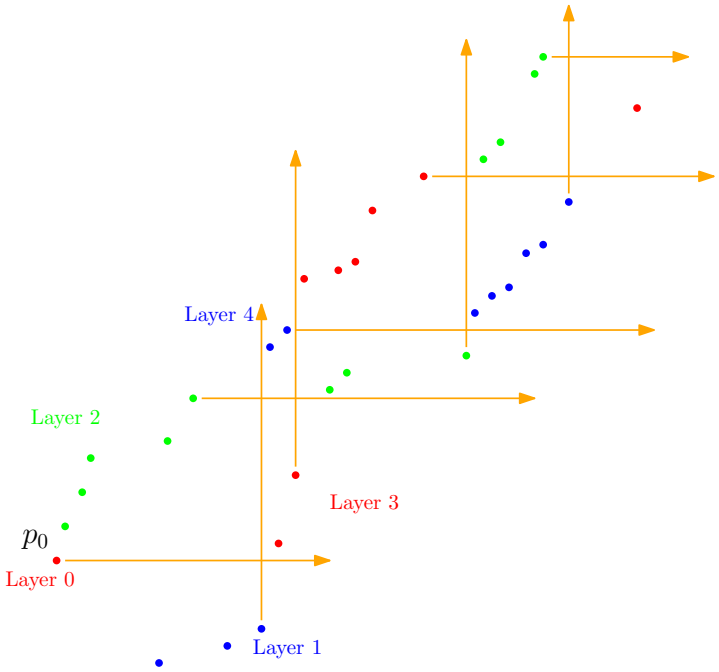


Figure 5.6: Layers on boundary points defined as distances from the leftmost point p_0. Orange lines represent which points are adjacent to (being in BR or TL) the last point in the layer.

We partition all points on the boundaries into layers, in the following way. The layer number 0 consists of a single left-most point p_0 in the whole set S. Note that p_0 is on the top boundary. Then, a boundary point is in a layer number i if its distance to p_0 is i. By $L(u)$ we denote the layer number of u. See Figure 5.6, we will soon see that indeed layers are always nicely structured, as pictured. Observe that in even layers, we have only points from the top boundary, and in odd layers only from the bottom boundary, as points on a single boundary are non-adjacent. Thus, points in a single layer are ordered, by both coordinates.

To determine the distance between boundary points, we use a method similar to the one from the paper of Gavoille and Paul [GP08], precisely Theorem 3.8. This is also connected to what Bazzaro and Gavoille [BG05] do in their work, but not identical, as they use bottom and top boundaries separately, as two almost independent interval graphs.

Lemma 5.4. *There exists a total ordering λ of boundary points such that given $\lambda(v), \lambda(u)$ and layer numbers $L(u) < L(v)$ for two boundary points u, v, we have $d(u, v) = L(v) - L(u)$ if and only if $\lambda(u) > \lambda(v)$.*

Proof. As noted, points on both boundaries are ordered, and layers switch between boundaries, starting with layer number 0 containing just a single left-most point from the top layer. Say ordered points on the top layer are $t_0, t_1, t_2, \ldots$. We prove that there exist strictly increasing numbers $i_0 = 0, i_2, i_4, \ldots$ such that layer number 0 consists of t_0, and then any layer $2k$ consists

of consecutive points $t_{i_{2k-2}+1}, \ldots, t_{i_{2k}}$. Similarly, for points $b_0, b_1, \ldots$ on bottom layer, there exists numbers $i_1, i_3, \ldots$ defining analogous ranges. Denote by $\mathsf{last}(q)$ the last point (with largest coordinates) in layer q. We prove by induction, in a given order, some intuitive properties (considering without loss of generality odd layer):

1. All points from layer $2k+1$ are to the right of all points from layer $2k$ (or all points from layer $2k$ are above layer $2k-1$).
2. All points from layer $2k+1$ are adjacent to $\mathsf{last}(2k)$.
3. Layer $2k+1$ is formed by consecutive points $b_{i_{2k-1}+1}, \ldots, b_{i_{2k+1}}$.
4. Ordered points from layer $2k$ are adjacent to increasing prefixes of the sequence of points $b_{i_{2k-1}+1}, \ldots, b_{i_{2k+1}}$. This means that, firstly, any point t_j with $L(t_j) = 2k$ is adjacent exactly to points $b_{i_{2k-1}+1}, \ldots, b_q$ from layer $2k+1$, for some $q \leq i_{2k+1}$. Secondly, for t_{j+1} with $L(t_{j+1}) = 2k$, t_{j+1} is adjacent to points $b_{i_{2k-1}+1}, \ldots, b_r$ with $q \leq r$.

The base of layer 0 is apparent, except for the third property, which can be done as in the induction step. Now consider layer $2k+1$. As all points from layer $2k$ are adjacent to $\mathsf{last}(2k-1)$, meaning they are in $\mathsf{TL}_{\mathsf{last}(2k-1)}$, all these points are to the left of points from layer $2k+1$. Moreover, the last point in layer $2k$ has the largest y coordinate, thus if any point from layer $2k+1$ is adjacent to some point from layer $2k$, then it is also adjacent to $\mathsf{last}(2k)$. This gives us the first two properties.

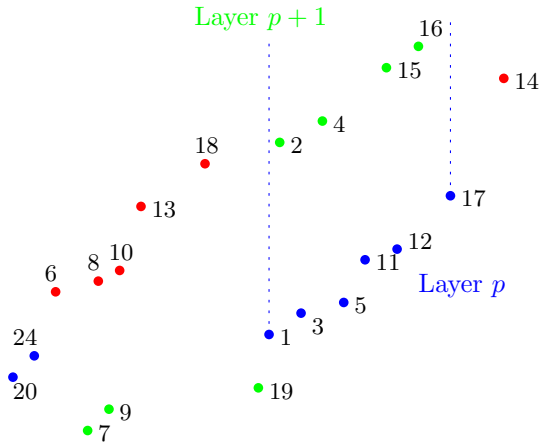


Figure 5.7: Considering layers p and $p+1$, a point with λ value of 1 is not adjacent to any point from layer $p+1$, an empty prefix. The point with value 3 is adjacent to just point with value 2, so range $[2,2]$. 5 and 11 are adjacent to prefix $[2,4]$. 12 is adjacent to $[2,15]$, and finally, 17 is adjacent to all points from layer $p+1$.

Now, by definition, if points b_q, b_r with $r > q$ are adjacent to some point v on the top boundary, then all points $b_q, b_{q+1}, \ldots, b_r$ are adjacent to v. Thus, if $b_{i_{2k-1}+1}$ is adjacent to $\mathsf{last}(2k)$, we get the third property. But it must be adjacent, as otherwise it would be above $\mathsf{last}(2k)$, and then layer $2k+1$ would be empty. We can apply the above principles to any point

from layer $2k$ - it either neighbours the first point in layer $2k+1$ or no point from this layer. This means any point from layer $2k$ neighbours prefix of points from layer $2k+1$ (possibly empty, possibly full layer). Moreover t_{j+1} is adjacent to the same points from layer $2k+1$ that t_j is, and possibly more, as t_{j+1} is above t_j. See Figure 5.7.

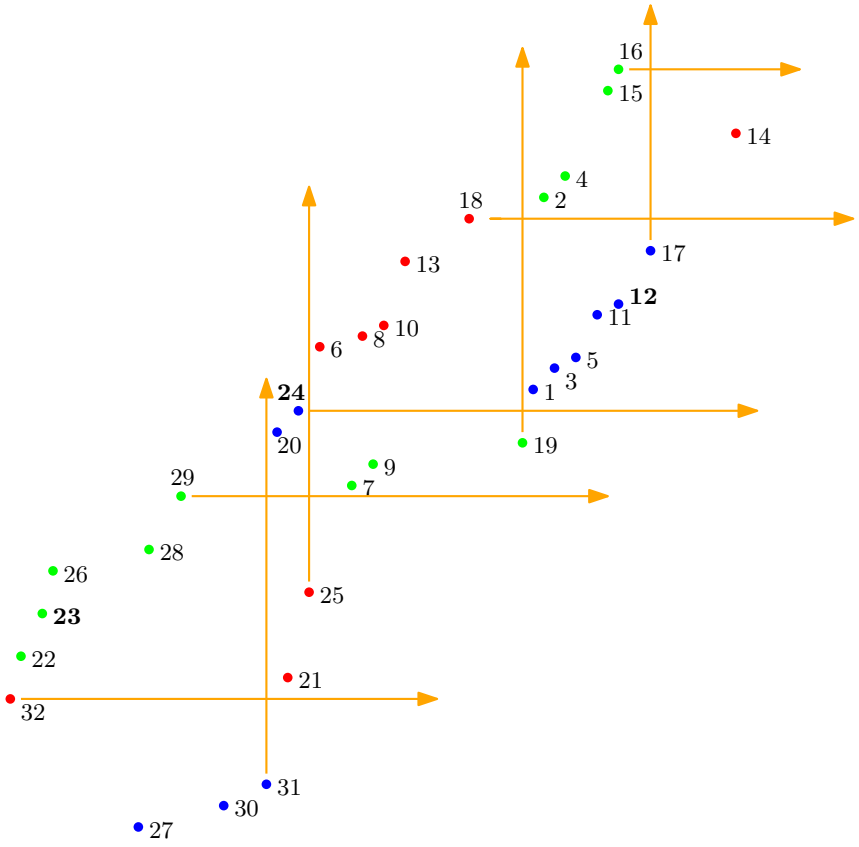


Figure 5.8: Boundary points with their λ values. The point with value 23 can reach point 12 using five edges, by path 23-21-20-19-18-12. But 23 cannot reach 24 using two edges, it needs four.

By the definition of layers, for points u, v, $d(u,v) \geq |L(u) - L(v)|$. If $d(u,v) = |L(u) - L(v)|$, we say that there is a *quick path* between them. Going back to the statement of the lemma, it says there is such ordering $\lambda(v)$ applied to boundary points, that there is a quick path between two points if relations between their λ values and layer numbers are opposite. We can create ordering λ in the following greedy way: starting from λ value of 1, always assign current value to the lowest point in the lowest layer such that it is adjacent to no points in the next layer without λ values already assigned, then increment current value. In other words, repeatedly choose the lowest (by layer) possible point having all neighbours from the next layer already assigned λ values. This is always possible, at the extreme case we will choose a point from the highest layer containing points without assigned λ value. See Figure 5.8 for an example.

For correctness, observe that when we choose v from layer k, for all layers larger than k the last point with assigned value is adjacent to all points with assigned values in the next layer. This is by greedy procedure, as assume there is a layer $j > k$ such that u, the last point with

assigned value in layer j, is not adjacent to w, the last point with assigned value in layer $j+1$. It is only possible if $\lambda(w) > \lambda(u)$. But by definition of greedy procedure, there is no reason to choose w at any point after choosing u and before choosing v, as no other point from layer j was chosen between these events and procedure always choose the lowest layer. Now, using the above, we can observe that at the moment we assign value for point v in layer k:

- There are quick paths from v to all points with assigned values and in layers larger than k. This is true by the choice of v and transitivity. It is given that v is adjacent to all points in layer $k+1$ with assigned values, then the last of these points is adjacent to all points in layer $k+2$ with assigned values, and so on.
- There are no quick paths from v to any point without assigned value in a layer larger than k. This is clear from the greedy procedure, previous point and the fourth inductive property - we do have quick paths to points in larger layers up to the last point with an assigned value, and these points cannot be adjacent to any more points, since they were chosen only when all their neighbours from the next layer got assigned values. □

By Lemma 5.4, we are able to detect when quick paths exist, and to complete knowledge about distances between boundary points we observe the following:

Property 5.5. *For any boundary points u, v with $L(v) \geq L(u)$, $d(u,v)$ is equal to either $L(v) - L(u)$ or $L(v) - L(u) + 2$.*

Proof. First let us argue that $d(u,v) \leq L(v) - L(u) + 2$. We observed $\mathsf{last}(i)$ is adjacent to all points in layer $i+1$. Thus, $(u, \mathsf{last}(L(u)-1), \mathsf{last}(L(u)), \ldots, \mathsf{last}(L(v)-1), v)$ is always a correct path with length $L(v)-L(u)+2$. By definition of layers, $d(u,v)$ cannot be less than $L(v)-L(u)$. Finally, by the Property 5.3 there is a shortest path that alternates between boundaries, so it cannot be of length $L(v) - L(u) + 1$, as we cannot change parity. □

To simplify our proofs, we will add some points to the original set. For each point v on the bottom boundary and from the original input, we add point $(v_x + \epsilon, v_y - \epsilon)$. It is easy to see that such a point lies on the bottom boundary, adding it does not change distances between any existing points, and it removes v from the bottom boundary. Then we change numeration to integer numbers again, increasing the range of numbers by some constant factor. Similarly, for any original point v on top boundary, we add $(v_x - \epsilon, v_y + \epsilon)$. What we achieve is that after this change no point from the original input lies on the boundary, which reduces the number of cases one needs to consider when assigning labels (only to the original points).

Now let us focus on any point v not on the boundary. Assume v is adjacent to some points from layers i and j, $j > i$. It cannot be that $j > i+2$, by definition of layers as distances from p_0. Thus, v is adjacent to points from at most three (consecutive) layers. We note that v is adjacent to a consecutive segment of ordered points from any layer i. Let us denote by $\mathsf{Bfirst}(v)$ and $\mathsf{Blast}(v)$ the first and last points on the bottom boundary adjacent to v and by $\mathsf{Tfirst}(v)$ and $\mathsf{Tlast}(v)$ the first and last points on the top boundary adjacent to v. Consult Figure 5.9.

We can make easy observation on points at distance two:

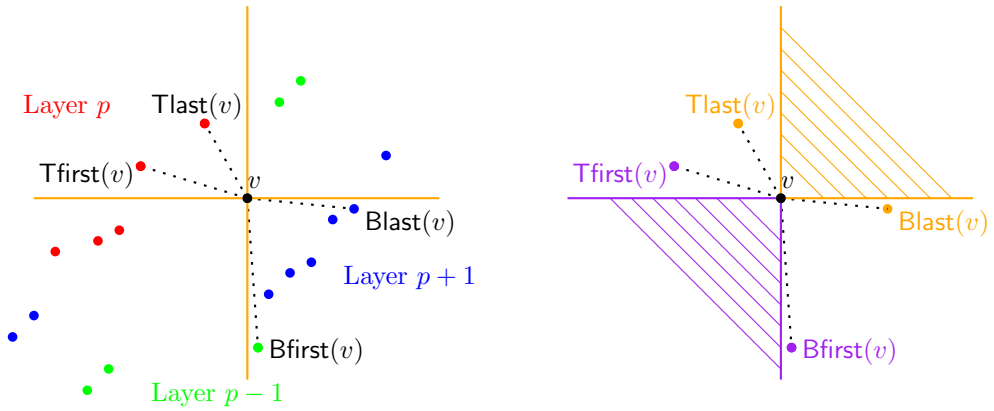


Figure 5.9: On the left: boundary points adjacent to v. Orange lines represent TL_v and BR_v. v is adjacent to the two last points in red layer p, one last point in green layer $p-1$, and the first five points in layer $p+1$. On the right: Two orange extreme neighbours are needed for paths to points in marked TR_v, and two purple ones for BL_v.

Property 5.6. *For any two points u, v, $d(u,v) \leq 2$ is equivalent to* $[\mathsf{Bfirst}(u), \mathsf{Blast}(u)] \cap [\mathsf{Bfirst}(v), \mathsf{Blast}(v)] \neq \emptyset$ *or* $[\mathsf{Tfirst}(u), \mathsf{Tlast}(u)] \cap [\mathsf{Tfirst}(v), \mathsf{Tlast}(v)] \neq \emptyset$.

This is since by Property 5.3, we must have a path between u, v at distance two going through a single point on the boundary. In the case of $d(u,v) = 1$, assume without loss of generality that $u \in \mathsf{TL}_v$, then $[\mathsf{Tfirst}(u), \mathsf{Tlast}(u)] \subseteq [\mathsf{Tfirst}(v), \mathsf{Tlast}(v)]$, and ranges are never empty.

Considering points at a distance of at least three, we have the following:

Lemma 5.7. *For any two non-boundary points u, v with $d(u,v) > 2$ and $u_x < v_x$, there is a shortest path from u to v with the second point being either* $\mathsf{Blast}(u)$ *or* $\mathsf{Tlast}(u)$ *and the penultimate point being either* $\mathsf{Bfirst}(v)$ *or* $\mathsf{Tfirst}(v)$.

Proof. We will prove statement for the second point, as the penultimate point is symmetric. By Property 5.3 there always exists a shortest path P with all but extreme points lying on alternating boundaries, with $P = (u = q_0, q_1, q_2, \ldots, w, q_{d(u,v)} = v)$, so we denote penultimate point by w.

Consider layer number of w. As $d(u,v) > 2$ and $u_x < v_x$, it must be that $v, w \in \mathsf{TR}_u$ and so $L(w) \geq \min(L(\mathsf{Tlast}(u)), L(\mathsf{Blast}(u)))$. If $L(w) \geq \max(L(\mathsf{Tlast}(u)), L(\mathsf{Blast}(u)))$, then by Property 5.5 and Lemma 5.4 we can replace q_1 with $\mathsf{Blast}(u)$ or $\mathsf{Tlast}(u)$ (which have the largest λ values in their layers among neighbours of w), while keeping the length of P and w as penultimate point. We are left with $L(w) = \min(L(\mathsf{Tlast}(u)), L(\mathsf{Blast}(u)))$. Assume $L(\mathsf{Blast}(u)) > L(\mathsf{Tlast}(u))$, so $L(w) = L(\mathsf{Tlast}(u)$ and also $w > \mathsf{Tlast}(u)$. Then w is adjacent to $\mathsf{last}(L(w)-1) \in \mathsf{BR}_u$ and thus also to $\mathsf{Blast}(u)$, so we can have $q_1 = \mathsf{Blast}(u)$. In other case, we could similarly set $q_1 = \mathsf{Tlast}(u)$. Thus, we can always change q_1 to be $\mathsf{Blast}(u)$ or $\mathsf{Tlast}(u)$, without changing w. □

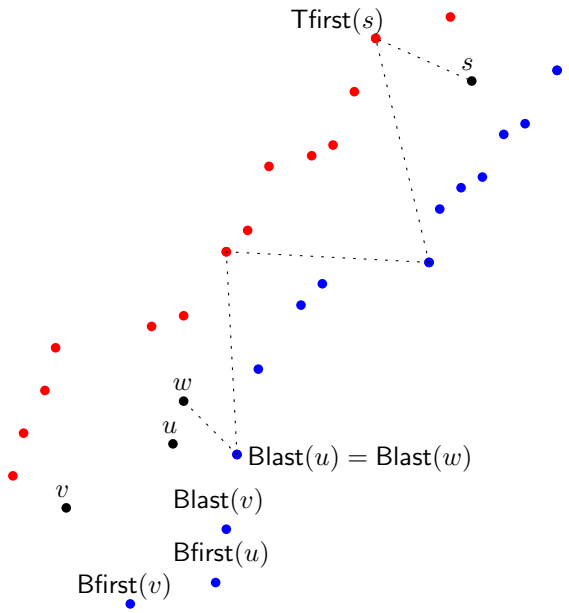


Figure 5.10: We have $d(u, v) = 2$, even though all $\mathsf{Bfirst}(v), \mathsf{Blast}(v), \mathsf{Bfirst}(u), \mathsf{Blast}(u)$ are different, but their ranges do intersect. Moreover, there is a shortest path from w to s with the second point being $\mathsf{Blast}(w)$ and the penultimate point being $\mathsf{Tfirst}(s)$.

We established ways to determine the distance between any points using distances between specific boundary points. Additionally, observe that all conditions from Lemma 5.4 and Property 5.6 can be checked using just λ values and layer numbers. That is, for u, v on the same boundary we have $u \leq v$ iff $(L(u), \lambda(u)) \leq_{lex} (L(v), \lambda(v))$.

At this stage, we could create labels of length $7 \log n + \mathcal{O}(1)$, by storing for each point v coordinates v_x, v_y, and for all points of interest $\mathsf{Bfirst}(v), \mathsf{Blast}(v), \mathsf{Tfirst}(v), \mathsf{Tlast}(v)$, their $\lambda(\cdot)$ and $L(\cdot)$ values. As a point is adjacent to at most three layers, all four layer numbers can be stored on just $\log n + \mathcal{O}(1)$ bits. Coordinates allow us to check for distance 1, distance two is checked by using Property 5.6, and larger distances by Property 5.5, Lemma 5.4 and 5.7.

5.3.2 Auxiliary Points

To better manage detecting points at distance 1 without explicitly storing coordinates, we will add, for each point in the set S, four additional artificial points, two on each boundary. Consider v from the initial set and its bottom-right quadrant BR_v. We add two points $v_b = (v_x - \epsilon, \mathsf{Bfirst}(v)_y - \epsilon)$ and $v_{b'} = (\mathsf{Blast}(v)_x + \epsilon, v_y + \epsilon)$ to the set S of points. See Figure 5.11. Then, we change the numeration of coordinates so that we still use the permutation of natural numbers up to $|S|$. This is repeated for all the initial points. First, we check that this addition did not disturb the properties of the points too much:

Lemma 5.8. *All added points are on the bottom boundary. Moreover, for any two points $u, w \in S$, $d(u, w)$ remains the same.*

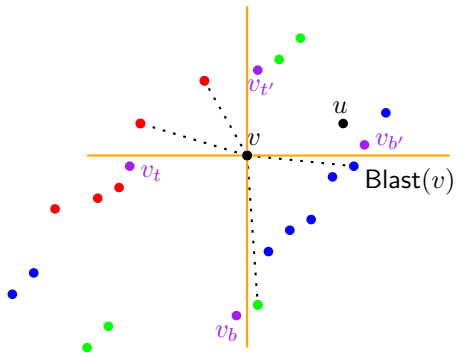


Figure 5.11: Four points added for a point v: v_b and $v_{b'}$ on bottom boundary, v_t and $v_{t'}$ on top. We have a property that whenever point u is adjacent to $v_{b'}$, it was already adjacent to $\mathsf{Blast}(v)$.

Proof. Considering the first property, we need to observe that when adding a point, its bottom-right quadrant is empty. For $v_{b'}$ it holds as the point is between $\mathsf{Blast}(v)$ and the next point on the bottom boundary on both axes. Thus it changes the status of no point on the bottom boundary and itself is on this boundary. We have a similar situation with v_b.

For the second property, we notice that any point adjacent to $v_{b'}$ is also adjacent to $\mathsf{Blast}(v)$. Since $v_{b'}$ and $\mathsf{Blast}(v)$ lie on the bottom boundary, any adjacent point must be in their top-left quadrant. As $v_{b'} = (\mathsf{Blast}(v)_x + \epsilon, v_y + \epsilon)$ and there are no points with x-coordinate between $\mathsf{Blast}(v)_x$ and $\mathsf{Blast}(v)_x + \epsilon$, if some point is to the left of $v_{b'}$, it is also to the left of $\mathsf{Blast}(v)$, and by definition we have $v_{b'} > v_y > \mathsf{Blast}(v)_y$. Similarly, any point adjacent to v_b is also adjacent to $\mathsf{Bfirst}(v)$. Therefore, these points cannot offer any shortcuts in existing shortest paths. □

Similarly, for each point in the initial set, we add two points on the upper boundary. That is, consider v and TL_v. We add two points $v_t = (\mathsf{Tfirst}(v)_x - \epsilon, v_y - \epsilon)$ and $v_{t'} = (v_x + \epsilon, \mathsf{Tlast}(v)_y + \epsilon)$ to the set S of points. Then, we again change the numeration of coordinates. This is symmetric and has the same properties.

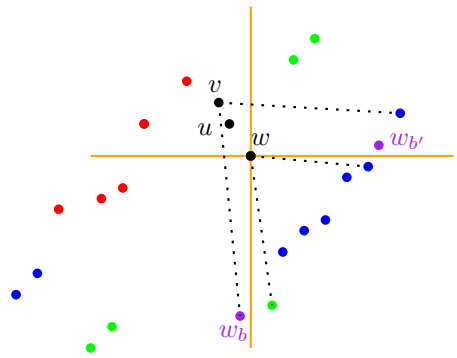


Figure 5.12: We have $w \in \mathsf{BR}_v$, and $\mathsf{Bfirst}(v) < \mathsf{Bfirst}(w) < \mathsf{Blast}(w) < \mathsf{Blast}(v)$. Meanwhile, $u \notin \mathsf{BR}_w$, as $\mathsf{Blast}(u) = w_{b'} > \mathsf{Blast}(w)$. Auxiliary points which would be added for u, v are not shown to avoid clutter.

After adding four auxiliary points for all initial points, we have the desired property:

Lemma 5.9. *For any two points v, w from the initial set, $w \in \mathsf{BR}_v$ is equivalent to $\mathsf{Bfirst}(v) < \mathsf{Bfirst}(w) \leq \mathsf{Blast}(w) < \mathsf{Blast}(v)$. Moreover, $w \in \mathsf{TL}_v$ is equivalent to $\mathsf{Tfirst}(v) < \mathsf{Tfirst}(w) \leq \mathsf{Tlast}(w) < \mathsf{Tlast}(v)$.*

Proof. The cases for both boundaries are symmetrical, so we focus on the bottom one.

If $w \in \mathsf{BR}_v$, then by transitivity v is adjacent to $\mathsf{Bfirst}(w)$, and then by definition of w_b, v is also adjacent to w_b. As $w_b < \mathsf{Bfirst}(w)$, we get $\mathsf{Bfirst}(v) < \mathsf{Bfirst}(w)$. Analogous facts hold for $w_{b'}$, therefore implication in the right direction holds.

We consider the left direction and use contraposition. Firstly, we want to show that if $w \notin \mathsf{BR}_v$, then either $\mathsf{Bfirst}(v) > \mathsf{Bfirst}(w)$ or $\mathsf{Blast}(w) > \mathsf{Blast}(v)$. $w \notin \mathsf{BR}_v$ means $w_x < v_x$ or $w_y > v_y$. Assume $w_x < v_x$ and $\mathsf{Bfirst}(v) \leq \mathsf{Bfirst}(w)$. This can be only if $\mathsf{Bfirst}(v) = \mathsf{Bfirst}(w)$, since w is to the left of v and points on boundaries are ordered. As v_b is below $\mathsf{Bfirst}(v)$ and between w_x and v_x, it must be that $v_b \in \mathsf{BR}_w$. So we have $v_b \in \mathsf{BR}_w$ and $v_b < \mathsf{Bfirst}(v) = \mathsf{Bfirst}(w)$, a contradiction, as then v_b should be $\mathsf{Bfirst}(w)$. Similarly using $v_{b'}$ we can show that assuming $w_y > v_y$ and $\mathsf{Blast}(w) \leq \mathsf{Blast}(v)$ leads to a contradiction. See Figure 5.12. □

The above lemma is useful when testing for adjacency of points – we do not need explicit coordinates to check it. Now, we are able to create labels of length $5 \log n + \mathcal{O}(1)$, as there is no longer a need to store coordinates, thus just four λ values, and additionally layer numbers using only $\log n + \mathcal{O}(1)$ bits. We can still improve on this, getting our final result in the next section.

5.4 Final Scheme of Size $3\log n$

In this section, the final improvement to label sizes is achieved, by collapsing two *pairs* of λ values into just two values, with an additional constant number of bits.

Lemma 5.10. *We can store for each input point v two integers $v_{x'}, v_{y'}$ with values in $\mathcal{O}(n)$, bit values v_{binf}, v_{tinf}, and layer numbers $L(\mathsf{Bfirst}(v)), L(\mathsf{Blast}(v)), L(\mathsf{Tfirst}(v)), L(\mathsf{Tlast}(v))$ not larger than n, such that distance queries for pairs of points can be answered using these values only.*

Proof. Let us consider any input point v, recall we made sure it does not lie on the boundary. As previously, we can store $L(\mathsf{Bfirst}(v))$, $L(\mathsf{Blast}(v))$, $L(\mathsf{Tfirst}(v))$ and $L(\mathsf{Tlast}(v))$ on $\log n + \mathcal{O}(1)$ bits, as there are no more than n layers, and the differences between these four values are at most 2. We would like to store one number instead of $\lambda(\mathsf{Blast}(v))$ and $\lambda(\mathsf{Tfirst}(v))$, then also one number instead of $\lambda(\mathsf{Bfirst}(v))$ and $\lambda(\mathsf{Tlast}(v))$. We need to consider four possible cases for the layout of layers, see Figure 5.13:

1. v is adjacent to two layers on the bottom boundary (and then necessarily one on the top).
2. v is adjacent to two layers on the top boundary.

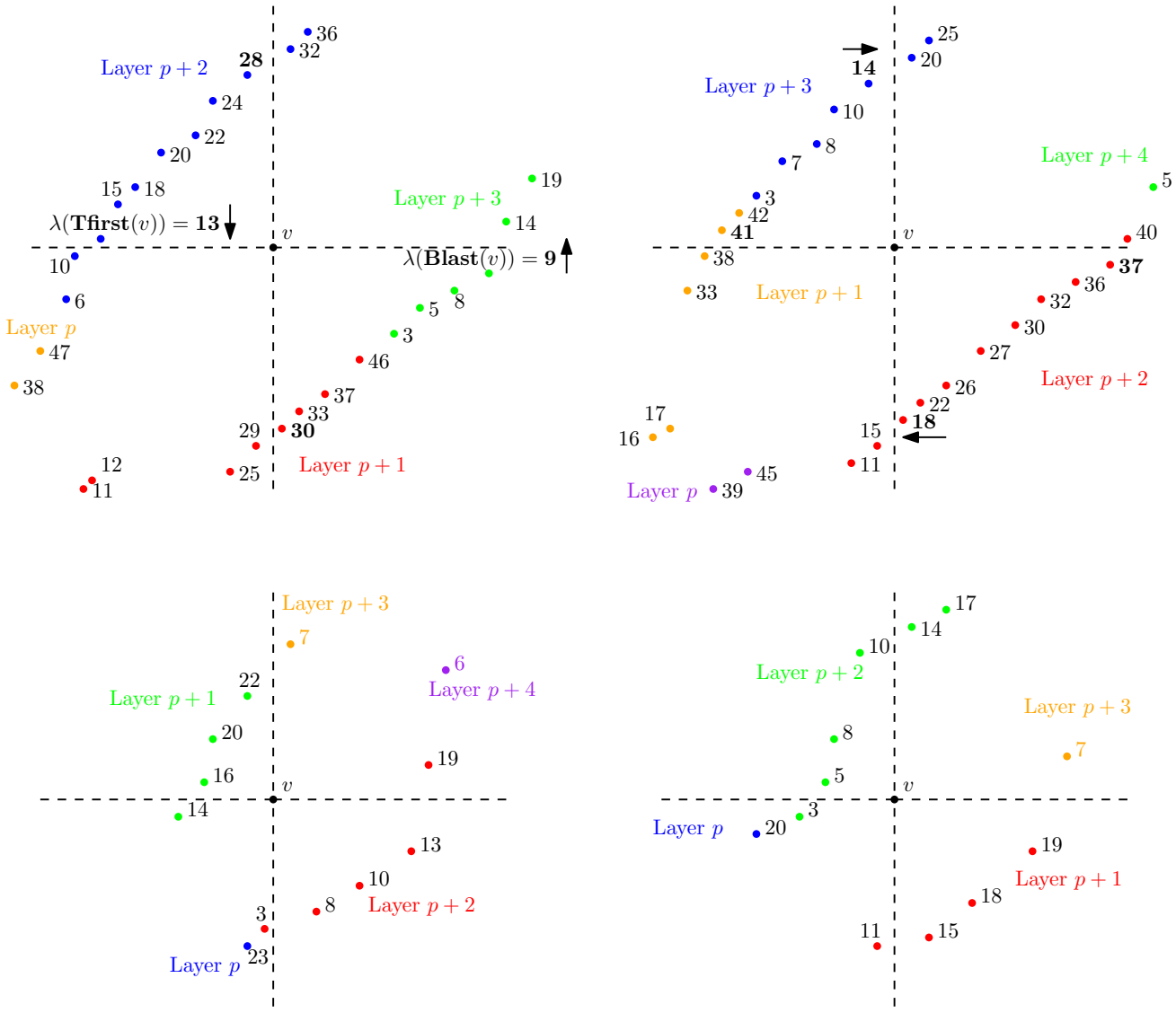


Figure 5.13: In the first case, label of v could store $\lambda(\mathsf{Bfirst}(v)), \lambda(\mathsf{Blast}(v)), \lambda(\mathsf{Tfirst}(v)), \lambda(\mathsf{Tlast}(v))$ values, which would be 30, 9, 13, and 28. But instead, we can use in its label just two values, $v_{y'} = 13 - \epsilon$ and $v_{x'} = 30 - \epsilon$. In the second case, point v could store values 18, 37, 41, 14. Instead, we can use $v_{y'} = 40 - \epsilon$ and $v_{x'} = 18 - \epsilon$. In the last case, we have point v adjacent to only two layers, and the lower one is on the bottom boundary. v could store values 15, 19, 5, and 10. Instead, we can use in its label just two numbers, $v_{y'} = 5$, $v_{x'} = 14 - \epsilon$, with two more bits indicating this case. We use the fact that a point with λ value of 19 is necessarily the last one in its layer and we won't need its exact value.

3. v is adjacent to one layer on both boundaries, and the layer on the bottom is higher.

4. v is adjacent to one layer on both boundaries, and the layer on the bottom is lower.

These will be referred to as possible layouts.

First, consider the last case. We argue that it must be that $\mathsf{Blast}(v)$ is the last point in its layer. Assume otherwise, so there is a point w on the bottom boundary, with $L(w) =$

$L(\mathsf{Blast}(v))$ and $w > \mathsf{Blast}(v)$, so necessarily $w \in \mathsf{TR}_v$. This means that there is a point u with $L(u) = L(w) - 1$ and $u_y > w_y > v_y$, by definition of layers. It cannot be $u \in \mathsf{TL}_v$, as we assumed the last case, where there are only points from layer $L(u) + 2$ in TL_v. But if $u \in \mathsf{TR}_v$, and thus $u_x > v_x$, then no point from layer $L(w) + 1$ could be in TL_v, as all of them are to the right of u, and we reach a final contradiction. Therefore, $\mathsf{Blast}(v)$ is in this case the last point in its layer. Notice that this information is easy to store, as it was proven that the last point in a layer has λ value larger than all of the points in larger layers, that is, there is always a quick path to such points. We can store a bit $v_{binf} = 1$ indicating this case, effectively using value of infinity instead of exact $\lambda(\mathsf{Blast}(v))$, and then just store unchanged $\lambda(\mathsf{Tfirst}(v))$. We still need to argue that no point from lower layers must use $\mathsf{Blast}(v)$ to reach v, and that checks for distances 1 and 2 works, which will be done later.

Now, consider the three first cases. In all of them, we have $L(\mathsf{Blast}(v)) > L(\mathsf{Tfirst}(v))$, and as these two points are adjacent $\lambda(\mathsf{Blast}(v)) < \lambda(\mathsf{Tfirst}(v))$ also holds. Denote by w a boundary point with the smallest $\lambda(w)$ value among points with $w_y > v_y$ and $\lambda(w) > \lambda(\mathsf{Blast}(v))$. Note that as $\mathsf{Tfirst}(v)$ can be chosen, such w always exists and it must be $\lambda(w) \leq \lambda(\mathsf{Tfirst}(v))$. We will store in our label value of $v_{y'} = \lambda(w) - \epsilon$ (to be normalised later), and as we will see this one number can replace both values of $\lambda(\mathsf{Blast}(v)), \lambda(\mathsf{Tfirst}(v))$. We say that value $\lambda(\mathsf{Blast}(v))$ is increased to $v_{y'}$, and $\lambda(\mathsf{Tfirst}(v))$ is decreased.

We can deal in a similar manner with $\mathsf{Bfirst}(v), \mathsf{Tlast}(v)$ values. If $\mathsf{Tlast}(v)$ is the last point in its layer (which is always true in the third case of the possible layouts), we store $v_{tinf} = 1$ and an exact value of $\lambda(\mathsf{Bfirst}(v))$. Otherwise, denote by w' a boundary point with the largest $\lambda(w')$ among points with $w'_x > v_x$ and $\lambda(w') > \lambda(\mathsf{Tlast}(v))$. We will store value of $v_{x'} = \lambda(w') - \epsilon$, and this one value can replace both $\lambda(\mathsf{Tlast}(v)), \lambda(\mathsf{Bfirst}(v))$. See Algorithm 5.1 for the summary of the encoder work.

We denote by $d((l, i), u)$, for numbers l, i and boundary point u, a distance between u and point in layer l with value of λ equal to i, as would be returned by using Lemma 5.4. Decoding navigates all possible cases and makes some distance queries, using values of $v_{x'}, v_{y'}$. See Algorithm 5.2 for the description of the decoder, some details will become clear later.

To prove correctness, first consider distances of at least three.

Property 5.11. *For u, v with $d(u, v) \geq 3$, our scheme will return $d(u, v)$, or a value less than 3.*

Proof. The formulation is for technical reasons, we will exclude the possibility of returning value less than 3 later. For some vertex v, due to distance, we are interested only in paths from v to points in BL_v and TR_v. By Lemma 5.7, the first ones start in either $\mathsf{Bfirst}(v)$ or $\mathsf{Tfirst}(v)$, other ones in either $\mathsf{Blast}(v)$ or $\mathsf{Tlast}(v)$. Let us informally go through what we need to check for any vertex v:

- For any boundary point $p \in \mathsf{TR}_v$ with $d(v, p) \geq 3$, distances between p and $\mathsf{Blast}(v), \mathsf{Tlast}(v)$ are preserved by the encoding.

Algorithm 5.1 Encoder computing labels for a set of points representing permutation graph.

function ENCODE(S)
  Input: set of points S, representing permutation graph
  Output: labels for all points in S

  $S' \leftarrow S$
  for $v \in S$ **do** ▷ Loop for adding auxiliary points
    Add to S' $v_b, v_{b'}, v_t, v_{t'}$
    if v is on the boundary of S' **then**
      Add to S' point removing v from boundary
  Compute layer numbers for boundary points of S'
  Compute λ numbers for boundary points of S'
  for $v \in S$ **do**
    if $\mathsf{Blast}(v)$ is last in its layer **then** ▷ Check for the special case
      $v_{binf} \leftarrow 1$
      $v_{y'} \leftarrow \lambda(\mathsf{Tfirst}(v))$
    else
      $v_{binf} \leftarrow 0$
      Find boundary w with $w_y > v_y$, $\lambda(w) > \lambda(\mathsf{Blast}(v))$, and minimum $\lambda(w)$
      $v_{y'} \leftarrow \lambda(w) - \epsilon$ ▷ Collapsed value for $\lambda(\mathsf{Blast}(v)), \lambda(\mathsf{Tfirst}(v))$
    if $\mathsf{Tlast}(v)$ is last in its layer **then**
      $v_{tinf} \leftarrow 1$
      $v_{x'} \leftarrow \lambda(\mathsf{Bfirst}(v))$
    else
      $v_{tinf} \leftarrow 0$
      Find boundary w' with $w'_x > v_x$, $\lambda(w') > \lambda(\mathsf{Tlast}(v))$, and minimum $\lambda(w')$
      $v_{x'} \leftarrow \lambda(w') - \epsilon$ ▷ Collapsed value for $\lambda(\mathsf{Tlast}(v)), \lambda(\mathsf{Bfirst}(v))$
    Store in $\ell(v)$ $L(\mathsf{Bfirst}(v))$
    Store in $\ell(v)$ constant number of bits encoding $L(\mathsf{Blast}(v)), L(\mathsf{Tfirst}(v)), L(\mathsf{Tlast}(v))$
    Store in $\ell(v)$ $v_{binf}, v_{y'}, v_{tinf}, v_{x'}$
    Output $\ell(v)$

- For any boundary point $p \in \mathsf{BL}_v$ with $d(v,p) \geq 3$, distances between p and $\mathsf{Blast}(v), \mathsf{Tlast}(v)$ as checked by the decoder are not smaller than in the graph (but are allowed to be larger).
- For any boundary point $p \in \mathsf{BL}_v$ with $d(v,p) \geq 3$, distances between p and $\mathsf{Bfirst}(v), \mathsf{Tfirst}(v)$ are preserved.
- For any boundary point $p \in \mathsf{TR}_v$ with $d(v,p) \geq 3$, distances between p and $\mathsf{Bfirst}(v), \mathsf{Tfirst}(v)$ as checked by the decoder are not smaller than in the graph.

First, consider $\mathsf{Blast}(v)$. We need that for any boundary point u in TR_v, $d(\mathsf{Blast}(v), u) =$

Algorithm 5.2 Decoder checking the distance between two points given only their labels.

function D($\ell(v), \ell(u)$)
  Input: labels of two points from the same permutation graph.
  Output: distance between points in the graph.

  function EXTRACT($\ell(t)$) ▷ Obtaining necessary values from label of t
    Extract $L(\mathsf{Bfirst}(t)), L(\mathsf{Blast}(t)), L(\mathsf{Tfirst}(t)), L(\mathsf{Tlast}(t))$ from $\ell(t)$
    $\lambda'(\mathsf{Tfirst}(t)) \leftarrow t_{y'}$ ▷ λ' are modified values, but to be treated as real λ
    if $t_{binf} = 1$ **then**
      $\lambda'(\mathsf{Blast}(t)) \leftarrow \infty$
    else
      $\lambda'(\mathsf{Blast}(t)) \leftarrow t_{y'}$
    $\lambda'(\mathsf{Bfirst}(t)) \leftarrow t_{x'}$
    if $t_{tinf} = 1$ **then**
      $\lambda'(\mathsf{Tlast}(t)) \leftarrow \infty$
    else
      $\lambda'(\mathsf{Tlast}(t)) \leftarrow t_{x'}$

  EXTRACT($\ell(u)$), EXTRACT($\ell(v)$)
  if range of u or v is a subset of the range of the other, on any boundary, **then**
    return 1
  if ranges of u and v intersect on any boundary **then**
    return 2
  $tmp \leftarrow \infty$ ▷ We are left with the case of distance > 2
  for $i \in \{\mathsf{Blast}(v), \mathsf{Tlast}(v)\}$ **do**
    for $j \in \{\mathsf{Bfirst}(u), \mathsf{Tfirst}(u)\}$ **do**
      $tmp \leftarrow \min(tmp, d([L(i), \lambda'(i)], [L(j), \lambda'(j)]))$
  for $i \in \{\mathsf{Blast}(u), \mathsf{Tlast}(u)\}$ **do**
    for $j \in \{\mathsf{Bfirst}(v), \mathsf{Tfirst}(v)\}$ **do**
      $tmp \leftarrow \min(tmp, d[(L(i), \lambda'(i)], [L(j), \lambda'(j)]))$
  return tmp

$d((L(\mathsf{Blast}(v)), v_{y'}), u)$. This holds by choice of $v_{y'}$, no relation between λ values has changed. In the case of the last layout, using infinite value also preserves these relations.

We also need to consider whether we could have reduced distance to some points in BL_v by increasing stored $\lambda(\mathsf{Blast}(v))$ value to $v_{y'}$. As for a quick path to exists the relations between layer numbers and λ values need to be opposite, increasing $\lambda(\mathsf{Blast}(v))$ could introduce a false quick path only for points in layers larger than $L(\mathsf{Blast}(v))$. For BL_v this is possible only in the last layout, when $\mathsf{Blast}(v)$ is the last point in its layer and is adjacent to all points in layer $L(\mathsf{Blast}(v)) + 1$ anyway, by Lemma 5.4.

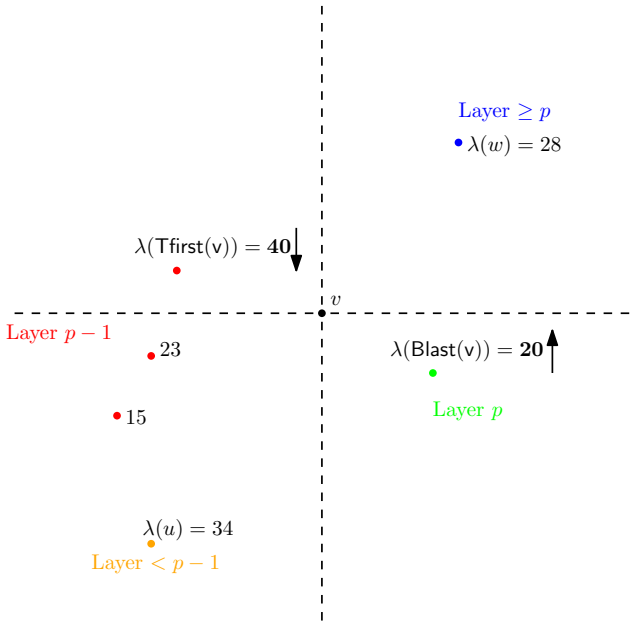


Figure 5.14: w is defined as the point above v with the smallest λ larger than $\lambda(\mathsf{Blast}(v))$. We show it is impossible for a point $u \in \mathsf{BL}_v$ to have $\lambda(u) > \lambda(w)$, as it cannot have a single point in layer $L(\mathsf{Tfirst}(v))$ on a quick path to w.

We had a $\mathsf{Blast}(v)$, now consider $\mathsf{Tfirst}(v)$, where the situation is more complicated due to the non-symmetric definition of $v_{y'}$. We assume $v_{binf} = 0$, as otherwise the exact value of $\lambda(\mathsf{Tfirst}(v))$ is stored and the decoder does not err. We need that for any boundary point u in BL_v, $d(\mathsf{Tfirst}(v), u) = d((L(\mathsf{Tfirst}(v)), v_{y'}), u)$. By examining possible layouts we see that any point $u \in \mathsf{BL}_v$ with $L(u) > L(\mathsf{Tfirst}(v))$ must have $L(u) = L(\mathsf{Blast}(v))$, thus $\lambda(u) < \lambda(\mathsf{Blast}(v))$, so $\lambda(u) < v_{y'} < \lambda(\mathsf{Tfirst}(v))$ and the distance query is still correct. If $L(u) = L(\mathsf{Tfirst}(v))$, we know the distance is always 2 regardless λ values.

Now assume there is boundary point $u \in \mathsf{BL}_v$ with $L(u) < L(\mathsf{Tfirst}(v))$ and $v_{y'} < \lambda(u) < \lambda(\mathsf{Tfirst}(v))$, which is the only remaining way to produce false result of distance query. Note that $L(\mathsf{Tfirst}(v)) < L(\mathsf{Blast}(v))$, by $v_{binf} = 0$. Recall w is a boundary point with the smallest $\lambda(w)$ value larger than $\lambda(\mathsf{Blast}(v))$ among points with $w_y > v_y$ thus we have $\lambda(\mathsf{Tfirst}(v)) \geq \lambda(w) > \lambda(\mathsf{Blast}(v))$. If $L(w) = L(\mathsf{Tfirst}(v))$, then $w = \mathsf{Tfirst}(v)$ and we have nothing to prove, so we assume $L(w) > L(\mathsf{Tfirst}(v))$. See Figure 5.14. It holds that $\lambda(u) > \lambda(w)$ and $L(u) < L(\mathsf{Tfirst}(v)) < L(w)$, thus $d(u, w) = L(w) - L(u)$. This means that there is a quick path P from u to w containing exactly one point from each layer in range $[L(u), L(w)]$.

Let $r \in P$ and $L(r) = L(\mathsf{Tfirst}(v))$. There is no quick path from u to $\mathsf{Tfirst}(v)$, so it must be $r < \mathsf{Tfirst}(v)$, which means $r_y < v_y$. This in turn means that r is not adjacent to any point in layer $L(\mathsf{Blast}(v))$ with λ value larger than $\lambda(\mathsf{Blast}(v)$. But then it cannot be that P contains a single point from layer $L(\mathsf{Blast}(v))$, as such a point must be adjacent to r and also have λ value of at least $\lambda(w)$, meaning larger than $\lambda(\mathsf{Blast}(v)$. Therefore, by contradiction, there cannot be a point $u \in \mathsf{BL}_v$ with $L(u) < L(\mathsf{Tfirst}(v))$ and $v_{y'} < \lambda(u) < \lambda(\mathsf{Tfirst}(v))$, which means that for all $u \in \mathsf{BL}_v$ we have $d(\mathsf{Tfirst}(v), u) = d((L(\mathsf{Tfirst}(v)), v_{y'}), u)$.

We also need to consider whether we could have reduced distance to some points in TR_v by using $v_{y'}$ value instead of $\lambda(\mathsf{Tfirst}(v))$. But in TR_v there are only points in layers at least as large as $L(\mathsf{Tfirst}(v))$, for which decreasing value of $\lambda(\mathsf{Tfirst}(v))$ cannot decrease output of distance query.

Proof for $v_{x'}$, $\mathsf{Tlast}(v)$, and $\mathsf{Bfirst}(v)$ is symmetric. □

As a side note, let us observe two things that may help in understanding our methods. It might be $d((L(\mathsf{Blast}(v)), v_{y'}), u) = d(\mathsf{Blast}(v), u) + 2$ for point $u \in \mathsf{BL}_v$, but by Lemma 5.7 we do not need to store this distance correctly, as there is a shortest path from u to v with the penultimate point not being $\mathsf{Blast}(v)$. In other words, we do lose some unnecessary information. Secondly, we can store infinity values for boundary points that are last in their layers because of layers and λ values definition – all points in the following layers have smaller λ values already. We could not store 'zero' or 'minus infinity' values for points that are first in their layers, but there is no need for this.

Now we turn to distances 1 and 2, with the former being easier.

Property 5.12. *For u, v with $d(u, v) = 1$, our scheme will return 1. For u, v with $d(u, v) \geq 2$, our scheme will never return 1.*

Proof. Once again, we have several cases of λ values being increased, decreased, or set to infinity, but can check that methods from Lemma 5.9 and Property 5.6 still holds for our new values. The encoder never changes layers of points, so any possible issue is connected only to λ values. Consider input points u, v with $v_y > u_y$ and $L(\mathsf{Blast}(v)) = L(\mathsf{Blast}(u))$. By $v_y > u_y$, we have $\mathsf{Blast}(v) \geq \mathsf{Blast}(u)$. Then if $v_{binf} = u_{binf} = 0$, we must have $v_{y'} \geq u_{y'}$. Indeed, when $\mathsf{Blast}(v) \neq \mathsf{Blast}(u)$, then $u_{y'} < \lambda(\mathsf{Blast}(v)) < v_{y'}$. If $\mathsf{Blast}(v) = \mathsf{Blast}(u)$, again $v_{y'} \geq u_{y'}$ as the smallest value of λ larger than $\lambda(\mathsf{Blast}(v))$ for points above u cannot be larger than analogous value for v by $v_y > u_y$. Finally, $u_{binf} = 1$ implies $v_{binf} = 1$. This means that for any u, v, we can derive whether $\lambda(\mathsf{Blast}(v)) \geq \lambda(\mathsf{Blast}(u))$ given $\ell(v), \ell(u)$ simply by comparing values from the labels. As $\mathsf{Tfirst}(v)$ too is replaced by $v_{y'}$, relation between $\mathsf{Tfirst}(v), \mathsf{Tfirst}(u)$ is retained. Similarly, we can check that respective inequalities hold for $v_{x'}, u_{x'}$ values.

Overall, this means that relations between points of the same kind (say $\mathsf{Bfirst}(v), \mathsf{Bfirst}(u)$) are retained, therefore by Lemma 5.9 our labeling correctly outputs 1 exactly when $d(u, v) = 1$. □

Lastly, we are left with the case of distance two.

Property 5.13. *For u, v with $d(u, v) = 2$, our scheme will return 2. For u, v with $d(u, v) \geq 3$, our scheme will never return 2.*

Proof. Using Property 5.6, whenever $d(u, v) = 2$ the decoder reports this correctly, as values of $\lambda(\mathsf{Blast}), \lambda(\mathsf{Tlast})$ can only be increased, and $\lambda(\mathsf{Bfirst}), \lambda(\mathsf{Tfirst})$ only decreased, and thus ranges only widen. Therefore, we need to just exclude the possibility of false intersections, that is, reporting 2 for $d(u, v) > 2$. First let us note that the case when $\lambda(\mathsf{Blast}(u)) < \lambda(\mathsf{Bfirst}(v))$

but $u_{y'} = v_{x'}$ is impossible to achieve. This is because whenever λ value is increased or decreased, it is set to some new unique value, different from all values existing at the moment, and $\lambda(\mathsf{Blast}(u)), \lambda(\mathsf{Bfirst}(v))$ are not changed simultaneously.

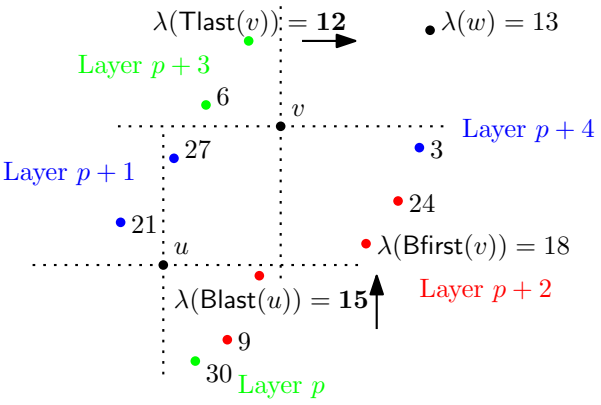


Figure 5.15: Here we consider, for $v > u$ with non-intersecting neighbourhoods on the bottom boundary and $L(\mathsf{Blast}(u)) = L(\mathsf{Bfirst}(v))$, whether we could get $u_{y'} > v_{x'}$, which would mean a false intersection in the decoding process. We show that it is impossible for some boundary point w to the right of v to have $\lambda(\mathsf{Tlast}(v)) < \lambda(w) < \lambda(\mathsf{Blast}(u))$.

Consider input points v, u with $v_y > u_y$, $L(\mathsf{Blast}(v)) = L(\mathsf{Bfirst}(u))$, $v_{tinf} = u_{binf} = 0$, and $d(u, v) > 2$. From distance constraint, we get that also $v_x > u_x$. Assume $\mathsf{Blast}(u) < \mathsf{Bfirst}(v)$, meaning ranges of neighbours of v, u on the bottom layer do not intersect. Recall that the encoder replaces $\lambda(\mathsf{Blast}(u))$ with $u_{y'}$, and $\lambda(\mathsf{Bfirst}(v))$ with $v_{x'}$, where $v_{x'}$ is defined using $\lambda(\mathsf{Tlast}(v))$. We will show that it is impossible that $u_{y'} > v_{x'}$, and thus ranges of v, u still do not intersect.

First, assume $\lambda(\mathsf{Tlast}(v)) > \lambda(\mathsf{Blast}(u))$. Then, by definition $u_{y'} < \lambda(\mathsf{Tlast}(v))$ as $\mathsf{Tlast}(v)$ is above u, and $v_{x'} > \lambda(\mathsf{Tlast}(v))$, which means $u_{y'} < v_{x'}$ as needed. So, we might assume $\lambda(\mathsf{Tlast}(v)) < \lambda(\mathsf{Blast}(u)) < \lambda(\mathsf{Bfirst}(v))$. Now, for $u_{y'} > v_{x'}$ to hold, we would need a boundary point w somewhere to the right of v, and with $\lambda(\mathsf{Tlast}(v)) < \lambda(w) < \lambda(\mathsf{Blast}(u))$. By definitions there are no points simultaneously below u and to the right of $\mathsf{Blast}(u)$, and by $\mathsf{Blast}(u) < \mathsf{Bfirst}(v)$ and $v_y > u_y$, v is to the right of $\mathsf{Blast}(u)$. Thus, w must be above u. We assumed $v_{tinf} = 0$, excluding the third case of possible layouts (Figure 5.13) v, so $L(\mathsf{Tlast}(v)) - 1 = L(\mathsf{Bfirst}(v)) = L(\mathsf{Blast}(u))$. By $\lambda(w) < \lambda(\mathsf{Blast}(u))$, it also must be that $L(w) \geq L(\mathsf{Tlast}(v))$. Summing up, we get that $L(\mathsf{Blast}(u)) = L(\mathsf{Bfirst}(v)) < L(\mathsf{Tlast}(v)) \leq L(w)$. See Figure 5.15, which depicts the only relevant remaining arrangement of points.

We examine $\mathsf{Blast}(u)$ to show that this case is also impossible. Since $L(w) > L(\mathsf{Blast}(u))$ and $\lambda(w) < \lambda(\mathsf{Blast}(u))$, it holds that $d(\mathsf{Blast}(u), w) = L(w) - L(\mathsf{Blast}(u))$ and there is a quick path between these two points, having a single point in each of layers $[L(\mathsf{Blast}(u)), L(w)]$. But as $\lambda(\mathsf{Tlast}(v)) < \lambda(w)$ and $L(\mathsf{Tlast}(v)) \leq L(w)$, we get $d(\mathsf{Tlast}(v), w) = L(w) - L(\mathsf{Tlast}(v)) + 2$. As $\mathsf{Blast}(u)$ is to the left of v, the largest adjacent point in layer $L(\mathsf{Tlast}(v))$ has λ value at most $\lambda(\mathsf{Tlast}(v))$, so smaller than $\lambda(w)$. But this is a contradiction, as then there cannot be a quick path from $\mathsf{Blast}(u)$ to w having a single point in layer $L(\mathsf{Tlast}(v))$.

This means that we can still use Property 5.6 for new stored values. □

By Properties 5.11, 5.12 and 5.13, we have proved the lemma. □

To conclude, we have that $\ell(v)$ consists of the following parts:

1. $L(\mathsf{Bfirst}(v))$, $L(\mathsf{Blast}(v))$, $L(\mathsf{Tfirst}(v))$ and $L(\mathsf{Tlast}(v))$, all stored on total $\log n + \mathcal{O}(1)$ bits due to differences between these values being at most 2.
2. Bit v_{binf} and value $v_{y'}$, on $\log n + \mathcal{O}(1)$ bits.
3. Bit v_{tinf} and value $v_{x'}$, like above.

We increased the number of vertices in the graph by a constant factor, so the final length is $3\log n + \mathcal{O}(1)$ bits. Decoding can be done in constant time.

5.4.1 Disconnected Graphs

Here we describe how to modify distance labeling for connected graphs into distance labeling for possibly disconnected graphs, by adding at most $\mathcal{O}(\log\log n)$ bits to the labels. This is a standard simple approach, present in some related works.

Assume there is some labeling scheme for family of connected graphs only, with labels of size $g(n) \geq \log n$. Given general graph, we sort connected components of the graph by decreasing size, say we have $C_1, C_2, \ldots$, then proceed with creating distance labeling for each individual connected component. The final label is the number of connected component of a vertex and then its label for the distance created in the component. If $v \in C_i$, we encode i on $\log i + \mathcal{O}(\log\log n)$ bits, using binary representation of i and indicating its length. We have $|C_i| \leq n/i$, so the modified label size is at most $g(n/i) + \mathcal{O}(\log\log n) + \log i \leq g(n) + \mathcal{O}(\log\log n)$, as claimed.

5.5 Conclusion

Improving upon the previous results, we have described a distance labeling scheme for permutation graphs matching existing lower bound up to an additive second-order $\mathcal{O}(\log\log n)$ term. This also improves constants in distance labeling for circular permutation graphs, as described in [BG05]. Namely, one can construct distance labeling of size $6\log n + \mathcal{O}(\log\log n)$ for such graphs. We leave as an open question determining the complexity of distance labeling for circular permutation graphs, and finding more interesting generalisations.

Chapter 6

Dynamic Approximation of LIS in Polylogarithmic Time

We consider maintaining an approximation of longest increasing subsequence (LIS) under insertions and deletions of elements anywhere in a sequence. The main result is as follows:

Theorem 6.1. *For any $\epsilon > 0$, there is a fully dynamic algorithm maintaining an $(1+\epsilon)$-approximation of LIS with insertions and deletions working in $\mathcal{O}(\epsilon^{-5}\log^{11} n)$ worst-case time.*

In fact, our algorithm allows for more general queries, namely approximating LIS of any continuous subsequence $(a_i, \ldots, a_j)$, in $\mathcal{O}(\log^2 n)$ time. Furthermore, if the returned approximation is k, then in time $\mathcal{O}(k)$ the algorithm can also provide an increasing subsequence of length k. Finally, the algorithm can be initialised with a sequence of length n in time $\mathcal{O}(n\epsilon^{-2}\log^6 n)$.

In the second part of this work, we apply the construction of Abboud and Dahlgaard [AD16], originally designed for distances in planar graphs, to provide conditional polynomial lower bounds for dynamic LIS with 1D queries (so also for dynamic weighted LIS). These are Theorem 6.32 and 6.33.

Parallel and independent work. Shortly after a preliminary version of our paper appeared on arXiv, two other relevant papers were made public. First, Seddighin and Mitzenmacher [MS20b] independently observed that their approximation algorithm can be applied to obtain Erdős-Szekeres partition (in particular, they provide a detailed description of how to modify their solution to extract the elements of LIS). Second, Kociumaka and Seddighin [KS21] modified the grid packing technique to obtain an algorithm with update time $\mathcal{O}(n^{o(1)})$ and approximation factor $1+o(1)$. While their modification allows for more general queries, it is not able to provide constant approximation in polylogarithmic time.

Overview of our approach. We start with a description of the main ingredient of our improved solution in a static setting, in which we are given an array $(a_1, a_2, \ldots, a_n)$ and want to preprocess it for computing LIS in any subarray $(a_i, \ldots, a_j)$, or (i, j) for short. While it is known

how to build a structure of size $\mathcal{O}(n \log n)$ capable of providing exact answers to such queries in $\mathcal{O}(\log n)$ time using the so-called unit-Monge matrices [Tis07, Chapter 8], this solution seems inherently static, and we follow a different approach that provides approximate answers.

We require that the structure is able to return $(1+\epsilon)^2$-approximate solution for any subarray (i, j). To this end, it consists of $\log_{1+\epsilon} n$ levels. The purpose of level k is to return, given a subarray (i, j) with LIS of length at least $(1+\epsilon)^{k+1}$, a subarray (i', j') with $i \leq i' \leq j' \leq j$ and LIS of length at least $(1+\epsilon)^k$. This indeed allows us to approximate the length of LIS in a subarray (i, j) by binary searching over the levels to find the largest level k for which the structure does not fail. The information stored on each level could of course be just a sorted list of all minimal subarrays (i', j') with LIS of length $\lceil (1+\epsilon)^k \rceil$. This is, however, not very useful when we try to make the structure dynamic for the following reason. Whenever we, say, delete an element at position x, this might possibly affect every level. Now, for a level k of the structure, there could be even $\Omega(n)$ minimal subarrays containing x, or (what is even worse) using the element a_x for their LIS. Such a situation might repeat again and again, which makes obtaining even an amortised efficient solution problematic. This suggests that we should maintain a sorted list of subarrays with small *depth*, defined as the largest number of stored subarrays possibly containing the same position x. Somewhat surprisingly, it turns out that there always exists such a sorted list of depth ϵ^{-1}. Our proof is constructive and based on a simple greedy procedure that actually constructs the list efficiently. The sorted list of subarrays stored at each level is called a cover, while the whole structure is referred to as a covering family. We call this method of creating an approximation covering family of small depth the sparsification procedure.

To explain how this insight can be applied for dynamic LIS, first we focus on the decremental version of the problem, in which we only need to support deletions. This is already quite hard if we aim for polylogarithmic update time, and has interesting consequences.

We start with normalising the entries in the input sequence $(a_1, a_2, \ldots, a_n)$ to form a permutation of $[n]$ (if there are ties, earlier elements are larger as to preserve the length of LIS). Then, it is helpful to visualise the input array $(a_1, a_2, \ldots, a_n)$ as a set of points $S = \{(i, a_i) : i \in [n]\}$. A deletion simply removes a point from S, and do not re-normalise the coordinates of the remaining points. It is straightforward to translate a deletion of element at position i of the current sequence into a deletion of a specified point of S in $\mathcal{O}(\log n)$ time. Now, finding LIS in the current sequence translates into finding the longest *chain* of points in the current S, defined as an ordered subset of points with the next point strictly dominating the previous point. In fact, our structure will implement more general queries corresponding to finding LIS in any subarray $(a_i, \ldots, a_j)$ of the current array, or using a geometric interpretation the longest chain in the subset of S consisting of all points with the x-coordinate in a given interval (x_1, x_2).

We maintain a recursive decomposition of $[n] \times [n]$ into smaller rectangles, roughly speaking by applying a primary divide-and-conquer guided by the y-coordinates, and then a secondary divide-and-conquer guided by the x-coordinates. Formally, we define dyadic intervals of the form $(i2^k, (i+1)2^k - 1)$, and consider all rectangles of the form $R = (x_1, y_1, x_2, y_2)$ such that (x_1, x_2) and (y_1, y_2) are dyadic. For each such dyadic rectangle R that contains at least one point from

S, we maintain a list of all points inside it. Our goal will be to allow approximating the longest chain in every R. To this end, we maintain a covering family $\mathsf{CF}(R)$ for the array obtained by writing down the y-coordinates of the points in R in the order of increasing x-coordinates. The precise definition and the choice of parameters for the family is slightly more complex than in the description above, in particular we need the approximation guarantee to depend on the height of R and be sufficiently good so that composing $\mathcal{O}(\log n)$ approximations still results in the desired bound. Also, now every cover consists of a sorted list of intervals, and each interval explicitly stores a chain of appropriate length.

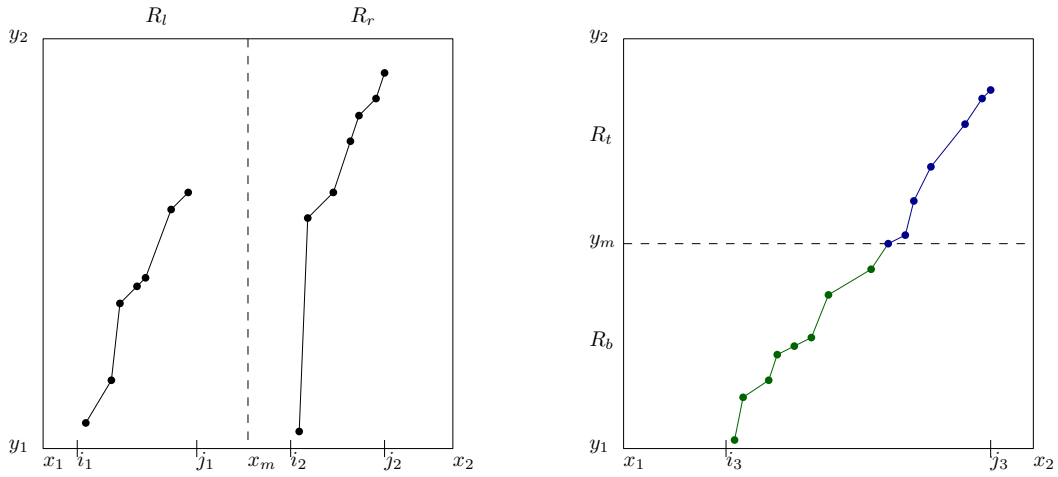


Figure 6.1: Intervals lying entirely in the left or right rectangle are already covered by their families. Intervals crossing x_m will be covered by concatenating pairs of segments from covering families of the bottom and top rectangle.

Every point of S belongs to $\mathcal{O}(\log^2 n)$ rectangles, and upon a deletion we need to update their covers at possibly all the levels. Due to the recursive nature of dyadic rectangles, a cover at level k of $\mathsf{CF}(R)$ can be obtained as follows. First, we split $R = (x_1, y_1, x_2, y_2)$ vertically into $R_l = (x_1, y_1, x_m - 1, y_2)$ and $R_r = (x_m, y_1, x_2, y_2)$, where $x_m = \lceil (x_1 + x_2)/2 \rceil$. We take the unions of covers at levels k in $\mathsf{CF}(R_l)$ and $\mathsf{CF}(R_r)$ and observe that we only need to additionally take care of the queries concerning intervals (x'_1, x'_2) with $x'_1 < x_m \leq x'_2$. It turns out that this can be done by splitting R horizontally into $R_b = (x_1, y_1, x_2, y_m - 1)$ and $R_t = (x_1, y_m, x_2, y_2)$, where $y_m = \lceil (y_1 + y_2)/2 \rceil$, and operating on a number of chains stored in $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$. By inspecting a few cases, this number can be bounded by the depth of every covering family, which will be kept $\mathcal{O}(\epsilon^{-1} \log n)$. Roughly speaking, we need to concatenate some chains from appropriately chosen levels of $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$. Consult Figure 6.1. Now it is tempting to form level k of $\mathsf{CF}(R)$ by taking the union of levels k from $\mathsf{CF}(R_l)$ and $\mathsf{CF}(R_r)$, and adding a small number of new intervals together with their chains. This is however not so simple, as we would increase the depth of the maintained covering family in every step of this process, and this could (multiplicatively) accumulate $\log n$ times. Fortunately, we can run our sparsification procedure on all points belonging to the new chains to guarantee that the depth remains small.

Taking the union of levels k from $\mathsf{CF}(R_l)$ and $\mathsf{CF}(R_r)$ can be done in only $\mathcal{O}(\log n)$ by main-

taining each cover in a persistent BST. However, running time of the sparsification step is actually quite large on later levels, and this seems problematic. We overcome this hurdle by running the sparsification only when sufficiently many deletions have been made in R to significantly affect the approximation guarantee provided by the cover. By appropriately adjusting the parameters, this turns out to be enough to guarantee polylogarithmic update time.

Having obtained a decremental version of our structure, we move to the fully dynamic version. Now the main issue is that, as insertions can happen anywhere, we cannot work with a fixed collection of dyadic rectangles. Therefore, we instead apply a two-dimensional recursion resembling 2D range trees. Then, we need to carefully revisit all the steps of the previous reasoning. The last step of our construction is removing amortisation. This follows by quite standard method of maintaining two copies of every structure, the first is used to answer queries while the second is being constructed in the background. However, this needs to be done in three places: sparsification procedure, secondary recursion, and primary recursion.

Organisation of the chapter. We start with preliminaries in Section 6.1. Then, in Sections 6.2, 6.3 and 6.4 we describe and analyze a decremental structure with deletions working in amortised polylogarithmic time. First, in Section 6.2 we define the subproblems considered in our structure, introduce the notion of covers, and explain how to compute covers with good properties at the expense of increasing the approximation guarantee. We will refer to this technique as sparsification. Second, in Section 6.3 we describe how to maintain the information associated with every subproblem under deletions of points. Third, in Section 6.4 we analyse approximation guarantee and running time of the decremental structure. In Section 6.5 we show to use it to obtain an improved algorithm for Erdős-Szekeres partitioning, and also provide the details of the $\Omega(n \log n)$ lower bound for this problem in the comparison-based model. Finally, in Section 6.6 we provide the necessary modifications to make the structure fully dynamic, and the update time worst-case. The last Section 6.7 considers two variants of the problem, for which we give conditional polynomial lower bounds.

6.1 Preliminaries

Let $[k]$ denote $\{0, 1, \ldots, k-1\}$. A pair of numbers $p_1 = (x_1, y_1)$ is smaller than $p_2 = (x_2, y_2)$, denoted $p_1 \prec p_2$, if $x_1 < x_2$ and $y_1 < y_2$. We usually treat an array of distinct numbers $A = (a_0, a_1, \ldots, a_{n-1})$ as a set S of 2D points with pairwise distinct x- and y-coordinates by defining $S = \{(i, a_i) : i \in [n]\}$. For an array A, by its subarray (i, j) we mean the subarray $(a_i, \ldots, a_j)$. For a set P of points, by its interval (i, j) we mean the set $P_{i,j} = \{p = (x, y) : p \in P, i \leq x \leq j\}$. In this work, intervals are always closed on both sides.

For any set of points P, its ordered subset of points $X = (x_1, x_2, \ldots, x_k)$ is a chain of length k if $x_1 \prec x_2 \prec \ldots \prec x_k$. We write $X_{i,j}$ to denote $(x_i, \ldots, x_j)$, and X_i refers to x_i. $X \circ Y$ denotes the concatenation of chains $X = (x_1, \ldots, x_i)$ and $Y = (y_1, \ldots, y_j)$, defined when $x_i \prec y_1$.

We want to maintain a structure allowing querying for approximate LIS under deletions and insertions of elements in an array (called the main array). The structure returns the approximated length and on demand it can also provide a chain of such length. An algorithm is p-approximate (or provides p-approximation), for $p \geq 1$, if it returns a chain of length at least r when the longest chain has length $p \cdot r$.

A *segment* is a triple (b, e, L), where b and e denote the beginning and the end of a subarray or an interval, and L is a list of consecutive elements of a chain inside that interval. The *score* of a segment is the length of L. We use the natural notation for positions of points, subarrays, intervals, or segments. Point (k, y) is inside interval (i, j) if $i \leq k \leq j$. Interval $A = (i_1, j_1)$ is to the left of interval $B = (i_2, j_2)$, denoted $A \prec B$, if $i_1 < i_2 \wedge j_1 < j_2$ (note that A and B can still overlap). A is inside B if $i_2 \leq i_1$ and $j_1 \leq j_2$.

As a subroutine, we use Fredman's $\mathcal{O}(n \log n)$ algorithm for computing LIS in an array. Recall that it can be used to maintain LIS under appending (or prepending, but not both) elements. That is, we can use it to incrementally compute LIS of the subarrays $(i, i), (i, i+1), \ldots$ so that after having computed LIS of the subarray (i, j) we can compute LIS of the subarray $(i, j+1)$ in $\mathcal{O}(\log n)$ time. Similarly, we can incrementally compute LIS of the subarrays $(j, j), (j-1, j), (j-2, j), \ldots$. When we wish to compute the longest chain in a set of points, we simply arrange its points in an array, sorted by the x-coordinates, and then compute LIS with respect to the y-coordinates.

BST. We use balanced binary search trees to store sets of elements. Each tree stores items ordered by their keys and provides operations $\mathsf{Insert}(x)$, $\mathsf{Delete}(k)$ and $\mathsf{Successor}(k)$. Operation $\mathsf{Find}(k)$ returns an item with the biggest key not greater than k, and $\mathsf{FindRank}(r)$ returns an item with the key of rank r in the set of keys in a BST. $\mathsf{Join}(T_1, T_2)$ merges two trees, provided that the keys of all items in T_1 are smaller than the keys of items in T_2. $\mathsf{Split}(k)$ divides a tree into two trees, the first containing items with keys less than k and the second containing the remaining items. $\mathsf{DeleteInterval}(k_1, k_2)$ deletes all elements with keys between k_1 and k_2; it can be implemented with two splits and one join.

We need a persistent BST which preserves the previous version of itself when it is modified. This property is used mostly when we join two trees T_1 and T_2 into one, but still want to access the original T_1 and T_2. Additionally, we need to augment the tree by storing the size of each subtree to allow for efficient $\mathsf{FindRank}$ implementation. All operations work in $\mathcal{O}(\log n)$ time, for a tree storing n items, by using e.g. persistent AVL trees [AVL62, DSST86].

6.2 Covers and Sparsification

In this and several next sections we assume that only deletions and queries are allowed. Later, we will explain how to modify an algorithm to allow also insertions. For the case without insertions, we replace each element of the input array by its rank in the set of all elements. Thus, the set S representing the input array consists of points with both coordinates from $[n]$. These coordinates

are not renumbered during the execution of the algorithm, we only delete points from S. For any pair of elements, their current positions in the main array can be compared in constant time by checking the x-coordinates. Their values can be also compared in constant time by inspecting the y-coordinates.

We say that interval (a, b) is dyadic if $a = i2^k$ and $b = (i+1)2^k - 1$ for some natural numbers i, k. Similarly, rectangle $R = (x_1, y_1, x_2, y_2)$ is dyadic if $x_1 = i2^{k_1}$, $x_2 = (i+1)2^{k_1} - 1$, $y_1 = j2^{k_2}$ and $y_2 = (j+1)2^{k_2} - 1$, for some natural numbers i, j, k_1, k_2. In other words, a dyadic rectangle spans dyadic intervals on both axes. Rectangle R contains point $s_r = (r, a_r)$ if $x_1 \leq r \leq x_2$ and $y_1 \leq a_r \leq y_2$. We say that k_2 is the *height* of a rectangle R. Assume for simplicity that n is a power of 2. We consider all dyadic rectangles with $0 \leq x_1, y_1, x_2, y_2 \leq n - 1$ possibly containing points from S. These rectangles have height between 0 and $\log n$. Moreover, only $\mathcal{O}(n \log^2 n)$ of them are nonempty, since each of the n points from S falls into $1 + \log n$ dyadic intervals on each of the axes. Every nonempty rectangle stores the set of points from S inside it in a BST, and we will think of the rectangles storing just one point as the base case.

Let $1 < \lambda < 2$ be the approximation parameter, where $\lambda = 1 + \epsilon'$ for some ϵ'. Our solution will be able to provide, for any dyadic rectangle R of height h, λ^{4h}-approximation of the longest chain in R. Thus, the approximation factor in the rectangle encompassing the whole set S will be $\lambda^{4 \log n}$. By setting $\epsilon' = \epsilon/(8 \log n)$, the solution provides an $(1 + \epsilon)$-approximation of LIS, for any $\epsilon \in (0, 1]$, as $(1 + \epsilon/(8 \log n))^{4 \log n} < e^{\epsilon/2} < 1 + \epsilon$. Furthermore, we have $\log_\lambda n = \log n / \log(1 + \epsilon/(8 \log n)) = \mathcal{O}(\epsilon^{-1} \log^2 n)$, for any $\epsilon \in (0, 1]$.

Covering family. In our solution, the approximation will be ensured by storing a sequence of sets of segments, each consecutive set containing segments with scores larger by a factor of λ^2. Given an interval of points containing a chain of length k, we should be able to find among stored segments one inside the given interval and with a score close to k. To achieve this, we need the notion of covers and covering families.

Definition 6.2. *Consider a set P of points.* (k_1, k_2)-cover *of P is a set S of segments, each with a score of at least k_1 and at most k_2, such that for any i, j, if $P_{i,j}$ contains a chain of length k_2, then S contains at least one segment X inside interval (i, j), and we say that (i, j) is* covered *by X. Moreover, we demand that in S no segment is inside another, so they can be sorted increasingly by their beginnings and ends at the same time and stored in a BST.*

We will refer to a (k, k)-cover simply as a k-cover, and say that the score of a (k_1, k_2)-cover is k_1. If we were operating on real numbers, γ^2-approximation could be achieved by storing a sequence of $\Theta(\log_\gamma n)$ covers, namely a (γ^r, γ^{r+1})-cover for every $0 \leq r < \log_\gamma n$. Then, after receiving a query about the longest chain in any interval, we could perform a binary search over the covers, in order to find the one with the largest score containing a segment inside the given interval. As unfortunately lengths of chains are natural numbers only, we need a slightly more complex choice of which covers to store. Additionally, we need the approximation factor to depend on the height of a rectangle.

Definition 6.3. *Let $1 < \gamma < 2$, r be a natural number greater than 1, and P a set of n points.* (γ, r)-covering family *of P consists of $\mathcal{O}(\log_\gamma n)$ covers, ordered by increasing scores. We call each of these covers a level. For the first $k = \min(n, 3(\gamma-1)^{-1})$ levels, the i-th level is just a i-cover of P. Then, level $k+j$ is a $(\lceil k\gamma^j \rceil, k\gamma^{j+r-1})$-cover of P, for all $j > 0$ such that $k\gamma^{j+r-1} \leq n$.*

Lemma 6.4. *(γ, r)-covering family of P provides a γ^r-approximation of the longest chain for any interval, in time $\mathcal{O}(\log(\log_\gamma n) \cdot \log n)$.*

Proof. We execute a binary search over the levels (sorted by scores). On each level, we query the BST storing the segments, trying to find a chain that lies inside the given interval I. We stress that the property of admitting such a chain is not monotone over the levels, but the binary search is still correct because of the following argument. Choose the largest level i corresponding to a (k_1, k_2)-cover such that I contains a chain of length k_2. Then, by definition we are guaranteed that on all levels from 1 to i, there is a segment inside I. Possibly, there are larger levels containing segments inside I, but this can only help in our binary search over the levels, and it will always return level i or possibly larger.

There are clearly $\mathcal{O}(\log_\gamma n)$ levels, each containing at most n segments, so the procedure takes $\mathcal{O}(\log(\log_\gamma n) \cdot \log n)$ time. Suppose the longest chain inside I has length l. If $l \leq k$, the covering family returns the exact answer, as level l is an l-cover. Otherwise, let e be such that $k\gamma^e \leq l < k\gamma^{e+1}$. If $e < r$, then a segment with a score of k from the k-th level of the family is good enough. If $e \geq r$, then the covering family returns a segment with a score of at least $\lceil k\gamma^{e-(r-1)} \rceil \geq k\gamma^{(e+1)-r}$, which is a γ^r-approximation. □

Additionally, by a $(\cdot, 0)$-covering family of m points we mean a set of k-covers, for each $1 \leq k \leq m$, which can always provide an exact answer.

Now we describe a greedy algorithm for creating covers with some additional properties. Given a cover C consisting of segments (b_i, e_i, L_i), we define *depth* of C as the largest subset of pairwise intersecting intervals $[b_i, e_i]$. In other words, we calculate the largest number of such intervals containing the same x. We design two algorithms for creating covers with small depth. The first one is actually exact, as for the short chains we cannot afford to decrease their length even by one during our recursive approximation. The second variant computes sets of segments for further levels of a covering family, approximating the long chains.

6.2.1 Exact Cover

We start with the non-approximate solution. The greedy algorithm for the exact k-cover works as follows. Assume $k > 1$, as 1-cover is basically a set of all elements. Starting with the whole input array and an empty cover C, the algorithm finds the shortest prefix P of an array in which there is a chain of length k. Now, some suffixes of P need to be covered, so the algorithm finds the shortest suffix S of P in which there still is a chain of length k, as S covers all of the mentioned suffixes. S and an arbitrary chain of length k inside it is added as a segment to a

cover C. Note that both the first and the last element of S must be in any chain of length k in S. Then, elements from the first one in the array up to the first one of S (including them) are cut off from the array. This is one step of the algorithm, which then repeats itself. Pseudocode is presented in Algorithm 6.1.

Algorithm 6.1 The greedy algorithm computing an exact cover.

```
1:  function COVER-EXACT(A, k)
2:      Input: array A = a_0, ..., a_{n-1}, integer parameter k.
3:      Output: a collection of segments forming k-cover of A.
4:      C ← ∅
5:      i ← 0
6:      while i < n do
7:          j ← i
8:          while |LIS(i, j)| < k and j < n do          ▷ LIS is computed incrementally
9:              j ← j + 1
10:         if j = n then
11:             return C
12:         ▷ (i, j) is the shortest prefix with a chain of length k
13:         q ← j
14:         while |LIS(q, j)| < k do                    ▷ LIS is computed incrementally
15:             q ← q − 1
16:         ▷ (q, j) is the shortest suffix with a chain of length k
17:         C ← C ∪ {(q, j, LIS(q, j)},   ▷ LIS(q, j) is a list of the elements of LIS in (a_q, ..., a_j)
18:         i ← q + 1
19:     return C
```

In the remaining part of this subsection we will prove the following theorem.

Theorem 6.5. *Algorithm 6.1 returns a k-cover of depth at most k in $\mathcal{O}(nk \log n)$ time.*

Note that such depth is optimal for a k-cover, for example in the case of a sorted array. We will prove Theorem 6.5 by induction, but first, some additional notation is needed. Let $P = (a, c)$ be some shortest prefix computed during the execution of the algorithm. Then, $S = (b, c)$ is its shortest suffix still containing a chain of length k, and let $X = (x_1, \ldots, x_k)$ be the lexicographically minimal (according to the positions of the elements) such chain. Till the end of this section, we will denote by x_i just a position of an element, while $|x_i|$ denotes its value. Recall that $X_{i,j}$ refers to the chain $(x_i, \ldots, x_j)$. Let $P' = (b + 1, d)$ be the next shortest prefix computed by the algorithm after P, and $X' = (x'_1, \ldots, x'_k)$ be the lexicographically minimal chain in the computed shortest suffix of P'. Our goal is to show the following.

Lemma 6.6. *For any two consecutive steps of Algorithm 6.1 and all $1 \leq i < k$, we have $x'_i \geq x_{i+1}$.*

Proof. The overall strategy is to apply induction on increasing i, however inside the inductive step we will need another induction.

Suppose $x'_i < x_{i+1}$. From the induction hypothesis, or in the base case of $i = 1$, we also have $x'_i > x_i$. It cannot be that $|x'_i| < |x_{i+1}|$, because then $X'_{1,i} \circ X_{i+1,k}$ is a chain of length k with $x'_1 > x_1$, so suffix S would not be the shortest one. Now, if $i = k - 1$, then $X_{1,k-1} \circ x'_{k-1}$ is a chain of length k contradicting the minimality of P, so assume $i < k - 1$. It is also not possible that $|x_{i+1}| \leq |x'_i| < |x_{i+2}|$, because then chain $X_{1,i} \circ x'_i \circ X_{i+2,k}$ is lexicographically smaller than X. Thus, we have $|x'_i| \geq |x_{i+2}|$.

Now, we claim that by secondary induction those properties extends further, namely for every $i \leq j < k$ we have $x'_j < x_{j+1}$, and for every $i \leq j < k - 1$ we have $|x'_j| \geq |x_{j+2}|$. The base case of $j = i$ was just proved. Now, assume the claim holds for $j - 1$, so $|x'_j| > |x'_{j-1}| \geq |x_{j+1}|$. If $x'_j > x_{j+1}$, then $X_{2,j+1} \circ X'_{j,k}$ is a chain of length $k + 1$ which contradicts P' being the shortest prefix. Additionally, $x'_j = x_{j+1}$ cannot hold since $|x'_j| > |x_{j+1}|$. Thus, we have $x'_j < x_{j+1}$. But then, for $j < k - 1$, it cannot be that $|x'_j| < |x_{j+2}|$, because in such case $X_{1,i} \circ X'_{i,j} \circ X_{j+2,k}$ is a chain of length k lexicographically smaller than X and inside S, since we assumed $x'_i < x_{i+1}$. Therefore, indeed for every $i \leq j < k$ we have $x'_j < x_{j+1}$.

But then in particular $x'_{k-1} < x_k$, which is the final contradiction needed for the main induction, since then $X_{1,i} \circ X'_{i,k-1}$ is a chain of length k contradicting minimality of P. □

Lemma 6.6 is enough to prove Theorem 6.5 by transitiveness as follows. During the execution of the greedy algorithm, the first element of any newly computed chain (which is the first element of the respective shortest suffix) is at position equal or larger than the position of the last element of a chain computed $k - 1$ steps before (which is the last element of the respective suffix). Therefore, at most k of the computed segments admit nonempty pairwise intersections. The bound on the time complexity of the whole algorithm follows, as the overall length of all arrays on which we run Fredman's algorithm is $\mathcal{O}(nk)$, and this algorithm takes logarithmic time per element. It is easy to see that no computed segment is inside another, thus all segments can be stored as a BST to meet the definition of a cover.

6.2.2 Approximate cover

The second algorithm uses two integer parameters, k_1 and k_2 with $k_2 > k_1$, and its goal is to create a (k_1, k_2)-cover. Let $\Delta = k_2 - k_1$. As it turns out, a greedy solution provides a roughly optimal depth of k_1/Δ. Similarly as with the exact cover, a greedy algorithm finds the shortest prefix P in which there is a chain of length k_2, then it finds the shortest suffix S of P in which there is a chain of length k_1, adds a segment corresponding to S to a cover C and then repeats, starting from the first element of S. The pseudocode of this procedure is very similar to the one of Algorithm 6.1. We just modify step 8 to search for the shortest prefix of length k_2 and step 14 to search for the shortest suffix of length k_1. Additionally, we change step 18 to $i \leftarrow q$, without incrementing by 1, as this is not necessary here and simplifies the proof. The modified

pseudocode is presented as Algorithm 6.2. Note that line 18 could be $i \leftarrow q+1$, but this is not necessary. See Figure 6.2 for a simple example.

Algorithm 6.2 The greedy algorithm computing one level of an approximate cover.

```
1:  function COVER-APPROX(A, k1, k2)
2:      Input: array A = a_0, ..., a_{n-1}, and two integer parameters, with k2 > k1.
3:      Output: a collection of segments forming (k1, k2)-cover of A.
4:      C ← ∅
5:      i ← 0
6:      while i < n do
7:          j ← i
8:          while |LIS(i, j)| < k2 and j < n do
9:              j ← j + 1
10:         if j = n then
11:             return C
12:         ▷ (i, j) is the shortest prefix with a chain of length k2
13:         q ← j
14:         while |LIS(q, j)| < k1 do
15:             q ← q − 1
16:         ▷ (q, j) is the shortest suffix with a chain of length k1
17:         C ← C ∪ {(q, j, LIS(q, j)},
18:         i ← q
19:     return C
```

In the remaining part of this subsection we will prove the following theorem.

Theorem 6.7. *Algorithm 6.2 returns a (k_1, k_2)-cover of depth at most k_1/Δ, where $\Delta = k_2 - k_1$, in $\mathcal{O}((n \log n)k_1/\Delta)$ time.*

In order to prove the above theorem, we will again examine the relationship between the elements of chains in the consecutive computed subarrays. We will do this in two steps. Let $P = (a, c)$ be some shortest prefix computed during the execution of the algorithm, and $X = (x_1, \ldots, x_{k_2})$ be the lexicographically minimal (according to the positions of the elements) chain of length k_2 in P. Then, $S = (b, c)$ is the shortest suffix of P still containing a chain of length k_1, and let $Y = (y_1, \ldots, y_{k_1})$ be lexicographically minimal such chain. Firstly, we need to inductively show the following.

Lemma 6.8. *For any two consecutive steps of Algorithm 6.2 and all $1 \leq i \leq k_1$, we have $y_i \geq x_{i+\Delta}$.*

Proof. The base case of $i = 1$ holds, since $X_{1+\Delta,k_2}$ is a chain of length k_1. Now, assume the claim holds for $i-1$. From the induction hypothesis, we have $y_i > x_{i-1+\Delta}$. Suppose $y_i < x_{i+\Delta}$. It

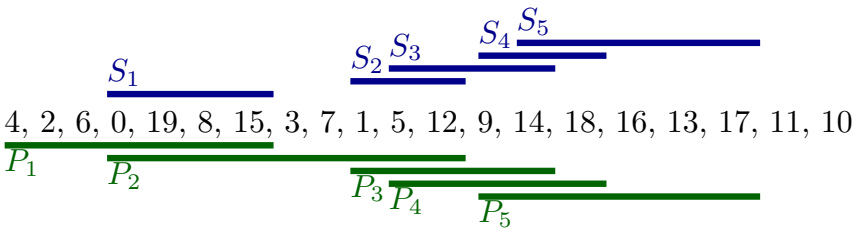


Figure 6.2: An example of the greedy algorithm for (3,4)-cover. The lexicographically minimal LIS in prefix P_2 is $(0, 3, 7, 12)$, and $(14, 16, 17)$ in suffix S_5. The set $S = \{S_1, \ldots, S_5\}$ is enough to cover any subarray containing a chain of length 4, for example subarray from value 15 to 14 is covered by S_2 and S_3. The depth of S is 3, as S_3, S_4 and S_5 have pairwise nonempty intersections.

cannot be that $|y_i| < |x_{i+\Delta}|$, since then $Y_{1,i} \circ X_{i+\Delta,k_2}$ is a chain of length k_1+1, which contradicts the minimality of suffix S. Thus, $|y_i| \geq |x_{i+\Delta}|$ must hold. But this is a contradiction, because then $X_{1,i-1+\Delta} \circ Y_{i,k_1}$ is a chain of length k_2 lexicographically smaller than X. □

With the above, we established a connection between (lexicographically minimal) chains in P and S, and now we need to connect the chain in S with the chain in the next prefix P'. So, let $P' = (b, d)$ be the next prefix computed by the algorithm after P, and $X' = (x'_1, \ldots, x'_{k_2})$ is the lexicographically minimal chain of length k_2 in P'. Additionally, let z be the maximum number such that $x'_z \leq c$, so exactly z elements of X' are inside S. Note that $z \leq k_1$.

Lemma 6.9. *For any two consecutive steps of Algorithm 6.2 and all $1 \leq i \leq k_1$, we have $x'_i \geq y_i$.*

Proof. For $i > z$ this is obvious. We proceed by induction on decreasing i with the base case $i = z$. The base and the step are proved similarly, so now assume that $i \leq z$ and the claim holds for $i + 1$. Suppose $x'_i < y_i$. It cannot be that $|x'_i| < |y_i|$, because then $X'_{1,i} \circ Y_{i,k_1}$ is a chain of length $k_1 + 1$ inside S. Thus, we have $|x'_i| \geq |y_i|$. Now, there are two cases. If $Y_{1,i}$ is lexicographically smaller than $X'_{1,i}$, we have a contradiction, since then $Y_{1,i} \circ X'_{i+1,k_2}$ is a chain of length k_2 lexicographically smaller than X' and in P'. To deal with the case of $Y_{1,i}$ being lexicographically larger than $X'_{1,i}$, we first need to argue that if $i < k_1$, then $|x'_i| < |y_{i+1}|$. If $x'_{i+1} = y_{i+1}$ this is trivial, and if not the inequality must hold because otherwise $Y_{1,i+1} \circ X'_{i+1,k_2}$ is a chain of length $k_2 + 1$ inside P'. Having established that $|x'_i| < |y_{i+1}|$ if $i < k_1$, we see that if $Y_{1,i}$ is lexicographically larger than $X'_{1,i}$, then we also arrive at a contradiction, because $X'_{1,i} \circ Y_{i+1,k_1}$ would be a chain of length k_1 lexicographically smaller than Y. Thus, in either case it cannot be that $x'_i < y_i$, so the claim holds. □

Similarly to Theorem 6.5, Theorem 6.7 holds by transitiveness and Lemmas 6.8 and 6.9, as the positions of the first elements in two consecutively computed segments shift by at least Δ. The bound on the time complexity follows by the same argument.

6.3 Decremental Structure

Having described covers and how to compute them, we can go back to our recursive structure of dyadic rectangles, and explain how to maintain their associated information. Let us consider a rectangle $R = (x_1, y_1, x_2, y_2)$ of height h and the set $P(R)$ of points from the main array inside R. We will maintain a $(\lambda^2, 2h)$-covering family of $P(R)$, denoted $\mathsf{CF}(R)$. At the very beginning, each covering family is constructed by running the greedy algorithms for every nonempty rectangle. To maintain the covers while the elements are being deleted, level k of $\mathsf{CF}(R)$ is obtained by taking the union of levels k of two smaller dyadic rectangles, and adding some extra segments. The extra segments are recomputed from time to time. More precisely, for each level of $\mathsf{CF}(R)$ we keep a counter. The counter is initially set to 0 and we increase it by 1 whenever a point inside R is deleted. As soon as the counter is sufficiently large, we recompute the extra segments. Clearly, we cannot afford to simply run the greedy algorithm on the whole set of points still existing in R. Instead, we use covering families from some other two smaller dyadic rectangles. We stress that level k of $\mathsf{CF}(R)$ is updated whenever a point is deleted $P(R)$, but the extra segments are recomputed only when sufficiently many deletions have occurred.

In this section, we describe how to maintain a single level of $\mathsf{CF}(R)$, say it is the level providing a (k_1, k_2)-cover. Let $x_m = \lceil (x_1 + x_2)/2 \rceil$ and $y_m = \lceil (y_1 + y_2)/2 \rceil$. In the recursive structure of rectangles, R is divided into the left, right, bottom and top rectangles. Denote them as $R_l = (x_1, y_1, x_m - 1, y_2)$, $R_r = (x_m, y_1, x_2, y_2)$, $R_b = (x_1, y_1, x_2, y_m - 1)$, and $R_t = (x_1, y_m, x_2, y_2)$, respectively; here we ignore the trivial base case of $x_1 = x_2$ or $y_1 = y_2$ to avoid clutter. Observe that as long as the approximation factor is tied to the height of a rectangle, covering families of R_l or R_r contain segments providing good approximation for any interval (i, j) with $j < x_m$ or $i \geq x_m$. In fact, $\mathsf{CF}(R_l)$ and $\mathsf{CF}(R_r)$ both contain a level providing a (k_1, k_2)-cover. The left and right rectangle cannot help in the case of intervals crossing the middle point x_m, though, so we need to somehow add segments covering any interval spanning both the left and right rectangle and containing a chain of length k_2. To this end, we will use R_b and R_t, concatenating pairs of segments from their covers. Consult Figure 6.1.

Merge procedure. We say that interval (i, j) crosses the middle point x_m if $i < x_m$ and $j \geq x_m$. Let us focus on some interval (i, j) crossing x_m, which contains chain X of length k_2. X can be split into two parts (one of them possibly empty), the first consisting of elements with y-coordinates smaller than y_m and the second consisting of the remaining elements of X. Say $X_p < y_m$ and $X_{p+1} \geq y_m$, then these parts are $X_{1,p}$ and X_{p+1,k_2}. Observe that we have access to some approximation of $X_{1,p}$ in R_b and of X_{p+1,k_2} in R_t in the form of segments from the covering families of the bottom and top rectangle. Let these segments be X'_b and X'_t, respectively, and assume they belong to levels l and l' of $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$, respectively. If there are many possible candidates for X'_b or X'_t, for the analysis we pick arbitrary ones. We will describe an operation Merge that selects and concatenates the relevant pairs of segments from $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$ for all intervals (i, j) crossing x_m, effectively finding segments that can be used as such X'_b and X'_t.

Merge consists of two steps. The first one is creating an initial set of segments. There are four cases of how X'_b and X'_t can be located. Consult Figure 6.3.

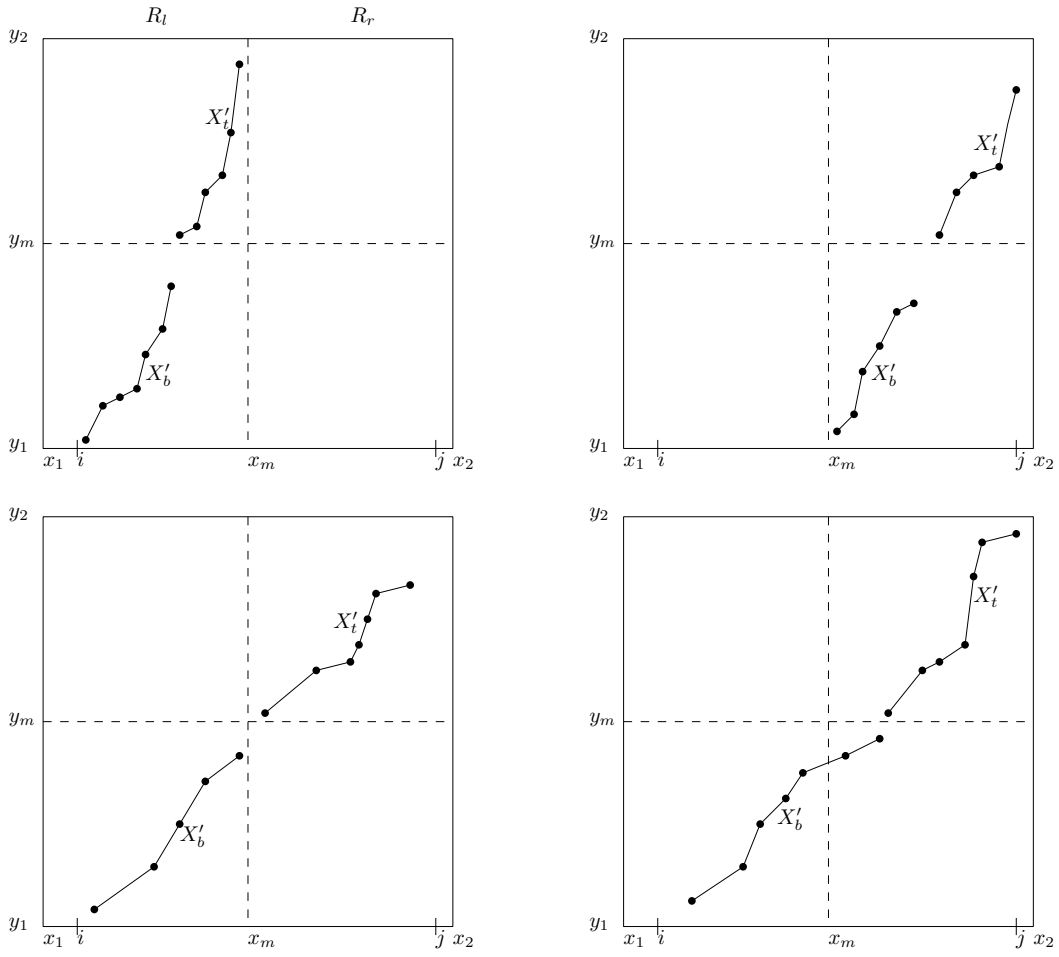


Figure 6.3: Possible cases of positions of X'_b and X'_t.

- Both X'_b and X'_t are inside the left rectangle R_l. Recall that segments in each level of a covering family are stored in a BST. Thus, we can quickly find the rightmost segment on level l' of $\mathsf{CF}(R_t)$ which is inside R_l, say it is segment $S' = (a, b, L)$. Observe that $S' \succeq X'_t$ must hold. We append S' to the rightmost segment S on l-th level of $\mathsf{CF}(R_b)$ whose interval ends before a. The resulting segment is inside (i, j) (as $S \succeq X'_b$ must hold) and provides a good approximation for a chain in (i, j), as S and S' comes from levels l and l'.

  Observe that even though both X'_b and X'_t reside in the left rectangle, we are not guaranteed to obtain the sought approximation of X just from $\mathsf{CF}(R_l)$. $X'_b \circ X'_t$ is already an approximation, so taking another approximation of it would square the approximation factor, which we cannot afford there.

- Both X'_b and X'_t are inside right rectangle R_r. This is symmetric to the first case, we could find the leftmost segment S on level l of $\mathsf{CF}(R_b)$ which is inside R_r, and append to it the leftmost segment on level l' of $\mathsf{CF}(R_t)$ starting after S.

- X'_b is inside R_l and X'_t is inside R_r. Then, we can find the rightmost segment S on level l of $\mathsf{CF}(R_b)$ which is inside R_l, and the leftmost segment S' on level l' of $\mathsf{CF}(R_t)$ which is inside R_r. It must be that $S \succeq X'_b$ and $S' \preceq X'_t$, thus $S \circ S'$ provides good approximation.

- One of X'_b, X'_t is not inside either left or right rectangle. Say that X'_b crosses x_m and X'_t is inside R_r, as the other case is analogous. Here, we can simply iterate over all segments on level l of $\mathsf{CF}(R_b)$ that cross x_m, and append to each of them the leftmost segment on level l' of $\mathsf{CF}(R_t)$ starting further. This allows us to find X'_b, and if it was concatenated with S' then $S' \preceq X'_t$.

There are obviously two issues. The first issue is that we are guessing levels l and l'. However, we can afford to iterate through all possibilities, as there are not so many levels, namely $\mathcal{O}(\log_\lambda n) = \mathcal{O}(\epsilon^{-1} \log^2 n)$. The second issue is that we are iterating through every segment crossing x_m and we do not want this quantity to be big. However, this is guaranteed by maintaining covers of small depth.

To sum up, performing the first step of Merge as described above gives us the desired approximation, assuming $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$ provide a good approximation. The pseudocode of both steps of Merge is shown as Algorithm 6.3, ignoring some corner cases. Namely, notice that 'the rightmost segment that is inside R_l' could not exist (and similarly in the symmetric cases), that is, there might be no segments inside R_l in the cover on that level. We add artificial guards to each cover, namely segments $(-\infty, -\infty, \emptyset)$ and $(\infty, \infty, \emptyset)$. The algorithm is aware that those segments are empty and does not concatenate them, and we will ignore this detail from now on. Additionally, as one of X'_b, X'_t can be empty, we pretend that each covering family contains an artificial level with covers of zero score. Keeping this in mind, the set of segments U computed in line 23 is enough to cover all intervals crossing x_m. However, it is slightly too big.

The goal of the second step of Merge (in lines 24-29) is to decrease the number of segments to $\mathcal{O}(\log n)$ at the expense of worsening the approximation by a factor of λ. Suppose U provided segments with scores of at least $k_1\lambda^2$. We create a set P of points appearing in a chain of any segment from U (we stress that only the points belonging to these chains are considered, not all points in the corresponding intervals), and then compute a $(\lceil k_1\lambda \rceil, k_1\lambda^2)$-cover U' of P with the greedy algorithm (for $k_1 \leq 3/\epsilon'$, we calculate an exact cover). As U provided a segment with a score of at least $k_1\lambda^2$ inside any interval crossing x_m and containing a chain of length at least k_2, now U' provides a segment with a score of at least $k_1\lambda$.

Let us state some properties of Merge more precisely. Depending on the last Boolean argument, we have approximated and exact variants.

Lemma 6.10. *Assume covering families of the bottom and top rectangle, denoted* $\mathsf{CF}(R_b)$ *and* $\mathsf{CF}(R_t)$*, provide* p*-approximation. Let* S *be the set of segments returned after invoking an approximation variant of* Merge *with parameters* $\mathsf{CF}(R_b), \mathsf{CF}(R_t), x_m, k,$ `true`*. Then, for any interval* (i, j) *that crosses* x_m *and contains a chain* X *of length* pk*,* S *contains a segment with a score of at least* $\lceil k/\lambda \rceil$ *inside* $P(R)_{i,j}$*.*

Algorithm 6.3 Implementation of Merge.

function MERGE(F_b, F_t, m, k, apx)
  Input: bottom and top covering families, middle element m, integer k, Boolean apx.
  Output: set of segments.

  $U \leftarrow \emptyset$
  for C being (r_1, r_2)-cover in F_b **do** ▷ Iterate over levels
    if $apx =$ `false` **then**
      Let C' be (r'_1, r'_2)-cover in F_t for r'_1 such that $r_1 + r'_1 = k$
    else
      Let C' be (r'_1, r'_2)-cover in F_t with the smallest r'_1 such that $r_1 + r'_1 \geq k$

    Let $s \in C$ be segment (s_i, s_j, s_L) with the largest s_j such that $s_j < m$
    Let $s' \in C$ be segment (s'_i, s'_j, s'_L) with the lowest s'_i such that $s'_i \geq m$
    ▷ Those are two non-crossing segments on the left and right of m
    for segment $u = (u_i, u_j, u_L)$ in C between s and s' **do** ▷ All segments crossing m
      Find segment $u' = (u'_i, u'_j, u'_L)$ in C' with the lowest u'_i such that $u'_i > u_j$
      $V \leftarrow (u_L \circ u'_L)_{1,k}$ ▷ Merging two chains and trimming
      $U \leftarrow U \cup \{(V_1, V_k, V)\}$

    Let $s \in C'$ be segment (s_i, s_j, s_L) with the largest s_j such that $s_j < m$
    Let $s' \in C'$ be segment (s'_i, s'_j, s'_L) with the lowest s'_i such that $s'_i \geq m$
    for segment $u' = (u'_i, u'_j, u'_L)$ in C' between s and s' **do**
      Find segment $u = (u_i, u_j, u_L)$ in C with the largest u_j such that $u_j < u'_i$
      $V \leftarrow (u_L \circ u'_L)_{1,k}$
      $U \leftarrow U \cup \{(V_1, V_k, V)\}$

  $P \leftarrow \{p : p \in L \text{ for any } (i, j, L) \in U\}$
  Store P as an array
  if $apx =$ `false` **then**
    $U' \leftarrow$ COVER-EXACT(P, k)
  else
    $U' \leftarrow$ COVER-APPROX$(P, \lceil k/\lambda \rceil, k)$
  In U' keep only segments crossing m and two non-crossing closest to m from both sides
  return U'

Proof. This mostly follows from the discussion above. X can be partitioned into two parts X_b and X_t lying entirely inside R_b and R_t, respectively. $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$ provide approximations of these parts, denoted X'_b and X'_t, with the sum of lengths at least k; in fact, there might be many possible segments X'_b and X'_t covering X_b and X_t. As described before in the four cases, the first part of Merge finds some $X'_b \circ X'_t$ and adds it to set U. Then in the second part we run the greedy algorithm to create a cover U' for the set P of points appearing in a chain of any segment of U. For any segment $(i', j', L) \in U$, $P_{i',j'}$ clearly contains a chain of length k. Additionally, for any interval (i, j) in P, if $P_{i,j}$ contained a chain of length k, then U' contains a segment with a score of $\lceil k/\lambda \rceil$ inside (i, j). It is easy to see that this remains true for all intervals crossing x_m if we keep in U' only the segments crossing x_m, the rightmost segment lying entirely in R_l, and the leftmost segment lying in R_r. □

A similar lemma holds for levels providing the exact covers. The proof is analogous but simpler, since there is no approximation at all, and we omit it here.

Lemma 6.11. *Assume in the covering families of the bottom and top rectangle, denoted* $\mathsf{CF}(R_b)$ *and* $\mathsf{CF}(R_t)$*, levels up to* $3/\epsilon'$ *are the exact covers. Let* S *be the set of segments returned after invoking an exact variant of* Merge *with parameters* $\mathsf{CF}(R_b), \mathsf{CF}(R_t), x_m, k,$ `false`*, with* $k \leq 3/\epsilon'$*. Then, for any interval* (i, j) *that crosses* x_m *and contains a chain of length* k*,* S *contains a segment with a score of* k *inside* $P(R)_{i,j}$*.*

Due to the application of the greedy algorithm in the second part, the running time of Merge might be unacceptably high, especially for large values of k_1. Therefore, we will run it only after a number of points has been deleted from R, making its running time amortised at the expense of increasing the approximation by a factor of λ. Recall that we maintain a counter associated with every level of the covering family, and increase it whenever a point is deleted from R. For a level providing a (k_1, k_2)-cover, the counter goes from 0 to $\max(1, \lfloor \epsilon' k_1 \rfloor - 1)$ (recall $\lambda = 1 + \epsilon'$). Then, we run Merge to recompute the cover. Because we recompute after having deleted $\epsilon' k_1$ points from R, the score of a segment might decrease from $k_1\lambda$ to $k_1\lambda - \epsilon' k_1 \geq k_1$, which is acceptable, and the cost of running Merge is distributed among $\epsilon' k_1$ deletions of elements from R.

The case of the first $\mathcal{O}(1/\epsilon')$ levels of a covering family is special, as for them we cannot afford to increase the approximation, even by adding one. Therefore, these covers need to be exact, and so we run Merge after every deletion of an element from R. The cost of doing so is not too high, as a single Merge for these levels considers only a polylogarithmic number of segments and points.

Bound on depth. Let us now prove the effects of sparsification as the second step of Merge on the depth of elements. Note that only the new segments computed by Merge can cross x_m.

Lemma 6.12. *The depth of a set of segments returned by* Merge *is* $\mathcal{O}(\epsilon^{-1} \log n)$*.*

Proof. The second step of Merge is sparsification by the greedy algorithm. Assume that we consider level l of a covering family. If $l \leq 3/\epsilon'$, then it was the greedy algorithm for the exact cover. As $1/\epsilon' = \mathcal{O}(\epsilon^{-1} \log n)$ and the maximum depth of the greedy algorithm for k-cover is k by Theorem 6.5, the claim holds. If l is larger, then the greedy algorithm for $(\lceil k/\lambda \rceil, k)$-cover was used. In that case, by Theorem 6.7 the maximum depth is

$$\frac{\lceil k/\lambda \rceil}{\Delta} = \frac{\lceil k/\lambda \rceil}{k - \lceil k/\lambda \rceil} < \frac{k/\lambda + 1}{k - k/\lambda - 1} = \frac{k + \lambda}{k\lambda - k - \lambda} < \frac{k+2}{k\lambda - k - 2} = \\ = \frac{k+2}{\epsilon' k - 2} = \frac{1 + 2/k}{\epsilon' - 2/k} < \frac{2}{\epsilon' - 2\epsilon'/3} = \frac{6}{\epsilon'} = \mathcal{O}(\epsilon^{-1} \log n),$$

as $1 < \lambda < 2$ and $k > 3/\epsilon'$. □

Notice that single sparsification does not have a huge impact. In Algorithm 6.3, the main loop in line 5 iterates over levels of a covering family, and there are $\mathcal{O}(\log_\lambda n) = \mathcal{O}(\epsilon^{-1} \log^2 n)$ of them. Secondary loops in lines 13 and 19 iterate over segments crossing the middle x-coordinate (and two additional extreme segments), and according to Lemma 6.12 there are $\mathcal{O}(\epsilon^{-1} \log n)$ such segments. Together this makes at most $\mathcal{O}(\epsilon^{-2} \log^3 n)$ segments produced in the first step of Merge, before sparsification reduces their number again to $\mathcal{O}(\epsilon^{-1} \log n)$ segments crossing x_m. In one step it may not seem significant, but it should be evident that this reduction in the number of segments is crucial, as otherwise we basically increase the depth multiplicatively by the number of levels for each increase in height of a rectangle.

Maintaining a cover as BST. After all steps of Merge, we have a set of segments constructed with help of $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$, but we still need to integrate it with covers of R_l and R_r into a single cover. To this end, we need to ensure that no segment is inside some other segment, and then put all of them into one BST. The latter is easy, as there are few new segments. We can just join trees of the left and right rectangle, then insert each new segment into the resulting tree. All operations here are done on persistent BSTs. But first, we need the non-inclusion property. Obviously, segments in trees from R_l and R_r are disjoint, so only the new ones computed by Merge can pose a problem.

Observe that from the new segments (i, j, L) with $j < x_m$, so being entirely in the left rectangle, we can keep only the rightmost one (or they might be no such segments at all). Similarly, from new segments lying entirely inside the right rectangle, we can keep only the leftmost one. This is done in the last lines of Algorithm 6.3. As long as one of those two segments is inside some segment from the left or right tree, we delete that unnecessary segment with a larger interval from its BST. If more than one segment needs to be deleted from one tree, they are continuous in that BST, so we can detect them all and use DeleteInterval. Now, we have these two segments and segments crossing the middle point, so $\mathcal{O}(\epsilon^{-1} \log n)$ new segments, and none of them can lie inside another. We only need to check whether any of the segments from the left or right tree lies inside any of those new segments and if so, we delete any such unnecessary new segment with a larger interval. At the end, we join the left and right tree and insert all the remaining new segments into the resulting tree, one by one. The final tree is stored as a cover, with its counter reset to zero.

In fact, joining trees from the left and right rectangle and then inserting segments returned by Merge is done not only on recomputing a cover on some level, but after *every* deletion of a point from $P(R)$. $\mathsf{CF}(R)$ relies on covers from smaller rectangles, so whenever they change, that should be taken into account. This is not too costly, though, as a single point is inside $\mathcal{O}(\log^2 n)$ rectangles, and the complexity of joining is polylogarithmic regardless of the size of the rectangle. Thus, each level of $\mathsf{CF}(R)$ stores $\mathcal{O}(\epsilon^{-1} \log n)$ segments returned by the last Merge for that level. An exception is when no Merge occurred yet, that is when a cover on that level still comes from the preprocessing. Then no segments are stored and no trees are joined.

6.4 Analysis of the Decremental Structure

In this section, we summarize how operations on our recursive structure of rectangles work, and analyse the approximation factor and the running time. First, let us recall what data is stored during the execution of the algorithm.

Information stored by the algorithm. On the global level, we store a pointer to the biggest rectangle. Each nonempty rectangle R stores:

- At most four pointers to R_l, R_r, R_b, R_t, whenever they are nonempty.
- Set $P(R)$ of points inside R, sorted by x-coordinates and stored in a BST.
- Covering family $\mathsf{CF}(R)$ of $P(R)$. $\mathsf{CF}(R)$ is an array of covers. Each cover is a BST of segments, ordered by coordinates (both simultaneously). Additionally, there is a counter tied to every cover.
- For each level of $\mathsf{CF}(R)$, a sorted list of segments returned by the last Merge on that level. In case of levels coming from preprocessing and not recomputed yet, nothing is stored here.

When an element is deleted, it is immediately deleted only in BSTs storing points inside rectangles. We do not modify the list stored for every segment in a cover, even if some of its elements are deleted. Instead, we recompute the cover and create new segments once in a while. The algorithm also stores a global Boolean array providing in constant time information which element of the input array has been already deleted.

Querying for LIS. Answering a query for approximated LIS inside any subarray of the main array is relatively uncomplicated. We need to use only the covering family of the main rectangle, which contains all of the elements. After a binary search over levels, some segment S is retrieved from the level providing (k_1, k_2)-cover. As an answer to a query, we report the lower bound k_1, not the actual length of the list storing elements of a chain in S (it could be bigger, but also possibly contains some deleted elements). If needed, in additional time $\mathcal{O}(k_1)$ the list can be traversed and elements forming the chain reported, excluding the elements that were deleted after forming that list. Therefore, we have the following.

Lemma 6.13. *The worst-case complexity of returning the approximated length of LIS in any subarray is* $\mathcal{O}(\log n \cdot \log(\epsilon^{-1}\log^2 n)) = \mathcal{O}(\log^2 n)$. *If the returned length is* k, *then the algorithm can return the corresponding increasing subsequence in* $\mathcal{O}(k)$ *time.*

Observe that we store points with x-coordinates equal to the initial indices in the input array. If we wish to answer queries for approximated LIS in subarray (i, j) of the main array, so when i, j are the current indices of elements, we need some translation. It is done simply by using $\mathsf{FindRank}(i)$ in a BST of the biggest rectangle, which stores exactly the set of non-deleted points. This way, we translate the current index into the initial index. The same method applies to deleting elements.

Handling a single deletion. When an element is deleted, the algorithm needs to go through each rectangle containing the point with coordinates defined by the initial index and value of that element. There are $\mathcal{O}(\log^2 n)$ such rectangles, and they are visited in order from the smallest to the biggest. By that, we mean in order of increasing span on y-axis, then increasing span on x-axis. For each rectangle, the counters of every level of their covering families are incremented. Some counters might reach their maximum value, which triggers computing Merge for the cover on that level. We also need to delete an element from BSTs storing points inside those rectangles. Finally, this element is marked as deleted in the global Boolean array.

Approximation factor. As mentioned earlier, the approximation guarantee is tied to the height of the rectangle. Intuitively, sparsification and amortisation are responsible for the approximation factor being $\lambda^{d\cdot 2h}$ for some constant d. We use covering families of rectangles R_b and R_t with heights smaller by one, then sparsification shortens segments by a factor of λ. Next, we let $\epsilon' k_1$ deletions to happen before recomputing, which shortens segments at most by the same factor. This is formalised in the following lemma.

Lemma 6.14. *Rectangle* R *of height* h *maintains a* $(\lambda^2, 2h)$*-covering family.*

Proof. We use induction on the size of the rectangle. Rectangles of zero height contain at most one element and the claim is trivial. Let us now consider rectangle R of height h, with its left/right/bottom/top rectangles R_l, R_r, R_b, R_t. From the assumption, R_l and R_r provide $(\lambda^2, 2h)$-covering families, while R_b and R_t provide $(\lambda^2, 2(h-1))$-covering families. The first $k = 3/\epsilon'$ levels in $\mathsf{CF}(R)$ are exact covers, recomputed with Merge after every deletion, therefore they are correct by Lemma 6.11.

Let us focus on a level providing $(\lceil k\lambda^{2j}\rceil, k\lambda^{2(j+2h-1)})$-cover for some $j > 0$. Segments coming from the same level in R_l and R_r have the correct score. We need to consider segments coming from Merge, more precisely invoking $\mathsf{Merge}(\mathsf{CF}(R_b), \mathsf{CF}(R_t), x_m, k\lambda^{2j+2}, \texttt{true})$. $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$ provide λ^{4h-4}-approximation. Say that we have an interval crossing the middle point and containing chain X of length at least $k\lambda^{2(j+2h-1)}$. By Lemma 6.10, in a set returned by Merge we have a chain of length $\lceil k\lambda^{2j+1}\rceil$ covering X.

Finally, because of amortising Merge, there might be up to $\lfloor \epsilon' \lceil k\lambda^{2j} \rceil \rfloor - 2$ deletions before recomputing the cover. Therefore, the guarantee on the length of a chain is

$$\begin{aligned} \lceil k\lambda^{2j+1} \rceil - \lfloor \epsilon' \lceil k\lambda^{2j} \rceil \rfloor + 2 &\geq \lambda k \lambda^{2j} - (\lambda - 1)\lceil k\lambda^{2j} \rceil + 2 \\ &> \lambda \lceil k\lambda^{2j} \rceil - (\lambda - 1)\lceil k\lambda^{2j} \rceil \\ &= \lceil k\lambda^{2j} \rceil, \end{aligned}$$

where we have used the assumption $\lambda < 2$. □

Time complexity. To prove that the amortised complexity of a deletion is polylogarithmic, we first need to bound the time complexity of a single Merge.

Lemma 6.15. *For any rectangle R, recomputing a (k_1, k_2)-cover from covering family of R works in $\mathcal{O}(k_1 \epsilon^{-3} \log^5 n)$ time.*

Proof. First, observe that for our construction we have $k_2 = \mathcal{O}(k_1)$. As stated before, the first step of Merge produces $\mathcal{O}(\epsilon^{-2} \log^3 n)$ segments. This takes time $\mathcal{O}((k_1 + \log n)\epsilon^{-2} \log^3 n)$, as for each segment there is one search in a BST involved and then we create a list containing elements of a chain. The score of each segment is $\mathcal{O}(k_1)$, so set P of all points appearing in any of the segments is of size $\mathcal{O}(k_1 \epsilon^{-2} \log^3 n)$. For the second step of Merge (sparsification), we sort P and run the greedy algorithm for finding a cover. This takes $\mathcal{O}(k_1 \epsilon^{-3} \log^5 n)$ time by Theorem 6.5, Theorem 6.7 and Lemma 6.12. Now the only thing that remains is joining BSTs representing covers of the left and right rectangle, then inserting the newly computed segments, all while ensuring there is no segment inside another. As described before, it is done with $\mathcal{O}(\epsilon^{-1} \log n)$ operations Find, Join, Delete and DeleteInterval. Thus, creating the final BST takes time $\mathcal{O}(\epsilon^{-1} \log^2 n)$. □

Lemma 6.16. *For any rectangle R, the amortised cost of recomputing (k_1, k_2)-cover from $\mathsf{CF}(R)$ is $\mathcal{O}(\epsilon^{-4} \log^6 n)$.*

Proof. For the case of $k_1 = k_2$, $k_1 = \mathcal{O}(1/\epsilon') = \mathcal{O}(\epsilon^{-1} \log n)$ and thus the claim holds. Otherwise, the step of recomputing that level of $\mathsf{CF}(R)$ is amortised between $\lfloor \epsilon k_1 \rfloor - 1$ deletions. We have $\epsilon' k_1 \geq 3$, and so $\lfloor \epsilon' k_1 \rfloor - 1 \geq 2\epsilon' k_1 / 3$. Therefore, using Lemma 6.15, the amortised cost is $\mathcal{O}(\epsilon^{-3}(\log^5 n)/\epsilon') = \mathcal{O}(\epsilon^{-4} \log^6 n)$. □

Now we need to sum up the cost of deletion among all the relevant rectangles.

Lemma 6.17. *The amortised cost of deletion is $\mathcal{O}(\epsilon^{-5} \log^{10} n)$.*

Proof. While deleting an element, the algorithm needs to visit $\mathcal{O}(\log^2 n)$ rectangles containing a point corresponding to that element. For each such rectangle R, we do the following:

- Delete the point from the BST storing points inside R, in time $\mathcal{O}(\log n)$.
- Increment the counter for each level of $\mathsf{CF}(R)$, there are $\mathcal{O}(\log_\lambda n) = \mathcal{O}(\epsilon^{-1} \log^2 n)$ levels.

- Pay the amortised cost of recomputing every level of $\mathsf{CF}(R)$, by Lemma 6.16 this is $\mathcal{O}(\epsilon^{-5}\log^8 n)$.
- For every level of $\mathsf{CF}(R)$, we join trees from $\mathsf{CF}(R_l), \mathsf{CF}(R_r)$ with segments returned by the last Merge. This takes time $\mathcal{O}(\epsilon^{-1}\log^2 n)$ for each level, so $\mathcal{O}(\epsilon^{-2}\log^4 n)$ in total.

Clearly, the third item is dominating. □

The last thing that should be investigated is the time complexity of preprocessing.

Lemma 6.18. *Preprocessing of the input array takes time* $\mathcal{O}(n\epsilon^{-2}\log^6 n)$.

Proof. Recall that each point is inside $\mathcal{O}(\log^2 n)$ rectangles. Thus, the recursive structure of nonempty rectangles can be built in time $\mathcal{O}(n\log^2 n)$, and BSTs of points inside each rectangle are constructed in total time $\mathcal{O}(n\log^3 n)$.

The main component is computing every covering family with the greedy algorithm. For rectangle R, the first $\mathcal{O}(1/\epsilon')$ levels of $\mathsf{CF}(R)$ are computed with the exact greedy algorithm, and then every (k_1, k_2)-cover is computed with the approximation greedy algorithm with parameters $P(R)$,$\lceil k_1\lambda\rceil$,k_2. As one point is in $\mathcal{O}(\log^2 n)$ rectangles, a covering family of any rectangle has $\mathcal{O}(\epsilon^{-1}\log^2 n)$ levels, and by computations identical to those in Lemma 6.12 the overhead of running the greedy algorithm for each element is also $\mathcal{O}(\epsilon^{-1}\log^2 n)$, the total time of computing covering families is $\mathcal{O}(n\epsilon^{-2}\log^6 n)$. □

6.5 Erdős-Szekeres Partitioning

Having finished the description of our decremental structure, we can state a particularly interesting application to decomposing a permutation on n elements into $\mathcal{O}(\sqrt{n})$ monotone subsequences in $\tilde{\mathcal{O}}(n)$ time. By repeatedly applying the Erdős-Szekeres theorem, that guarantees the existence of a monotone subsequence of length at least $\sqrt{n}$ in any permutation on n elements, it is clear that such a decomposition exists. Now we can find it rather efficiently.

Lemma 6.19. *A permutation on n elements can be partitioned into $\mathcal{O}(\sqrt{n})$ monotone subsequences in $\tilde{\mathcal{O}}(n)$ time.*

Proof. We maintain a 2-approximation of LIS with our decremental algorithm. As long as the approximated length is at least $\sqrt{n}$, we retrieve its corresponding increasing subsequence and delete all of its elements. This takes polylogarithmic time per deleted elements, so $\tilde{\mathcal{O}}(n)$ overall, and creates at most $\sqrt{n}$ increasing subsequences. Let the length of LIS of the remaining elements be k. Because we have maintained a 2-approximation, $k < 2\sqrt{n}$. Next, we run Fredman's algorithm on the remaining part of the permutation in $\mathcal{O}(n\log n)$ time. Recall that it scans the elements of the input sequence $(a_1, a_2, \ldots, a_m)$ and computes for every i the largest ℓ such that there exists an increasing subsequence of length ℓ ending with a_i. Then, it is straightforward to verify that, for every $\ell = 1, 2, \ldots, k$, the elements for which this quantity is equal to ℓ form

a decreasing subsequence, so we obtain a partition of the remaining elements into k decreasing subsequences. Thus, we partitioned the whole sequence into $\mathcal{O}(\sqrt{n})$ monotone subsequences. □

On the lower bound side, we have the following.

Lemma 6.20. *Any comparison-based algorithm for decomposing a permutation on n elements into $\mathcal{O}(\sqrt{n})$ monotone subsequences needs $\Omega(n \log n)$ comparisons.*

Proof. Assume that the algorithm creates at most $c\sqrt{n}$ subsequences using $\alpha n \log n$ comparisons. Then, we can merge all subsequences into a single sorting sequence using $n \log(c\sqrt{n})$ comparisons by merging them into pairs, quadruples, and so on. Therefore, by the lower bound of $n \log n - \mathcal{O}(n)$ for any comparison-based sorting algorithm, we must have $\alpha n \log n + n \log(c\sqrt{n}) \geq n \log n - \mathcal{O}(n)$, so indeed $\alpha \geq 1/2 - o(1)$. □

6.6 Fully Dynamic Worst-Case Structure

In this section we extend our decremental structure to also allow insertions of new elements at any position in the main array, and then describe how to deamortise the fully dynamic structure. As the required changes turn out to be relatively minor, we only provide a description of the new ingredients and their effect on both the approximation guarantee and the running time.

Coordinates of points. Recall that our decremental structure was based on normalising the entries in the input sequence $(a_1, a_2, \ldots, a_n)$, creating a set of points $S = \{(i, a_i) : i \in [n]\}$, and then maintaining some information for each nonempty dyadic rectangle. Now that we allow both deletions and insertions, it is convenient to think that the points have real coordinates, and we simply add new points to the current set without modifying the coordinates of the already present points (provided that both the x- and y-coordinates are distinct). However, we need to actually implement this efficiently.

We maintain a (totally ordered) list under deletions and insertions in an order-maintenance structure. Such a structure can handle the following operations:

- $\mathsf{Insert}(a, b)$, which inserts element a immediately after element b in a list and returns a handle to a.
- $\mathsf{Delete}(a)$.
- $\mathsf{Order}(a, b)$, answers whether a precedes b in a list given handles of two elements.

Theorem 6.21. *[DS87, BCD+02] There is an order-maintenance data structure supporting all operations in $\mathcal{O}(1)$ worst-case time.*

To allow insertions in our algorithm, we maintain a global order-maintenance structure, and x-coordinate of a point is a handle to an element in that structure. Thus, we still are able to

compare points in constant time. Additionally, we keep in the structure two artificial guards corresponding to $-\infty$ and ∞ values on x-axis. Our algorithm will sometimes still need an x-coordinate of an already deleted point, therefore an element of the order-maintenance structure is deleted only when no longer referred to. Nevertheless, it will be easy to verify that the order-maintenance structure stores $\tilde{\mathcal{O}}(n)$ elements at any time. We also note that using a BST to implement an order-maintenance structure in $\mathcal{O}(\log n)$ time per operation would not dramatically increase our time complexities.

Two-dimensional recursion. While in the decremental structure we could use a fixed collection of rectangles, doing so in the fully dynamic case is problematic, as now the number of points in a rectangle might either decrease or increase, and we still need to be able to split a rectangle into either left/right or bottom/top while maintaining logarithmic depth of the decomposition. To this end, we define the rectangles differently.

The new definition is inspired by dynamic 2D range trees, see e.g. [Lue78]. We first describe how to obtain the initial collection of rectangles, and then explain how to maintain it. We apply a primary recursion on the y-coordinates, and then a secondary recursion on the x coordinates, starting from a rectangle containing the whole set of points. Take a rectangle R containing m points considered in the primary recursion. We initially choose and store the middle y-coordinate, denoted y_m, such that exactly $m/2$ points from $P(R)$ are below the horizontal line $y = y_m$, then partition R into R_b and R_t consisting of points under or above y_m, respectively, thus each containing $m/2$ points. Then we repeat the primary recursion on R_b and R_t. Each rectangle considered in the primary recursion is then further partitioned by the secondary recursion. Take a rectangle R containing m points considered in the secondary recursion. We initially choose and store the middle x-coordinate, denoted x_m, such that exactly $m/2$ points from $P(R)$ are to the left of the vertical line $x = x_m$, then partition R into R_l and R_r consisting of points on left or right of $x = x_m$, respectively, thus each containing $m/2$ points. Then we repeat the secondary recursion on R_l and R_r.

To maintain the collection, we proceed as follows. Consider a rectangle R created in the primary recursion with m points, and let y_m be its middle y-coordinate. We keep $y = y_m$ as a line of partition of R into R_b and R_t during the next $m/4$ insertions and deletions of points *inside* R, and then recompute from scratch the whole recursive decomposition of R (including running the secondary recursion for each of the rectangles obtained in the primary recursion, and constructing a covering family for each recomputed rectangle). This is enough to guarantee that the partition of any R into R_b and R_t in the primary recursion is 2-balanced, meaning that the number of points in R_b and R_t differs by at most a factor of 2. We proceed similarly for any rectangle considered in the secondary recursion, so for any rectangle R created in the secondary recursion for m points, with x_m being the middle x-coordinate, we keep $x = x_m$ as a line of partition of R into R_l and R_r during the next $m/4$ insertions and deletions of points inside R, and then recompute from scratch the whole recursive decomposition of R (now this only refers to running the secondary recursion, and constructing a covering family for each

recomputed rectangle). The depth of both recursions is clearly logarithmic, and the amortised time of recomputing the decompositions is hopefully not too big.

The height of a node in a tree is the length of the longest path from that node to any leaf below it. Each rectangle R maintained by the algorithm is identified by a node of some secondary recursion tree T. Additionally, we identify the root of T with its corresponding node of the primary recursion tree, so a rectangle R identified with the root of T is also identified with the corresponding node of the primary recursion tree. Define the secondary height of R to be the height of R in its secondary recursion tree T, and the primary height of R to be the height of the root of T in the primary recursion tree. Since we keep 2-balanced partition, both heights are at most $\log_{3/2} n = \mathcal{O}(\log n)$. Because of this we need to slightly adjust ϵ' (but only up to a multiplicative constant).

In total, any point is inside $\mathcal{O}(\log^2 n)$ rectangles, similarly to inside how many dyadic rectangles contained a given point in the previous sections. Additionally, for a rectangle R in the primary recursion tree with $|P(R)| = m$, the sum of numbers of points inside rectangles below R in the two-dimensional recursion is $\mathcal{O}(m \log^2 m)$. For R in the secondary recursion tree, this is only $\mathcal{O}(m \log m)$. Recomputing in a rectangle is easy to amortise, but we need to explain how to maintain $\mathsf{CF}(R)$ between recomputing of the whole rectangles, that is, how Merge works in this setting.

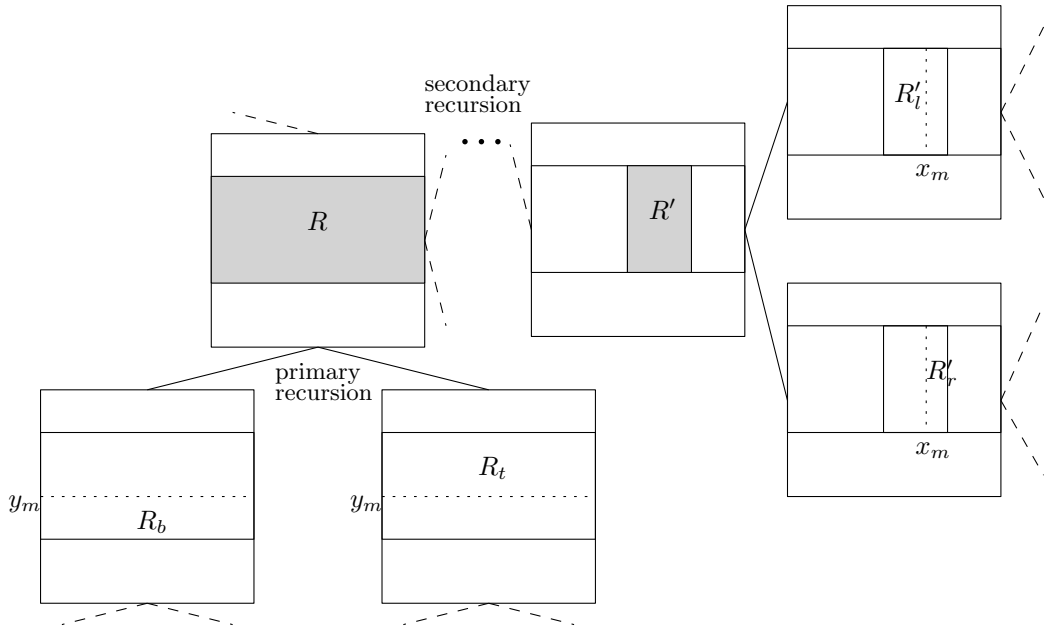


Figure 6.4: Rectangle R in the primary recursion is partitioned into the bottom and top part. Additionally, it is identified with the root of a secondary recursion tree, where rectangles are repeatedly partitioned into the left and right part.

Merge revisited. Let R be some rectangle obtained in the primary recursion, in which R is divided into 2-balanced R_b and R_t along the horizontal line $y = y_m$. Let R' be some rectangle obtained in the secondary recursion as a result of subdividing R. R' consists of all points in

$P(R)$ with x-coordinates belonging to some interval, and is divided into 2-balanced R'_l and R'_r along the vertical line $x = x_m$. Consult Figure 6.4. Say that a (k_1, k_2)-cover on level l of $\mathsf{CF}(R')$ needs to be recomputed. This can be achieved similarly as in the decremental structure. We join BSTs of covers on level l of $\mathsf{CF}(R'_l)$ and $\mathsf{CF}(R'_r)$, and then run Merge using covering families of R_b and R_t. There are two minor issues.

The first issue is that R_b and R_t include all points from $P(R')$, but possibly contains many more, as they form a partition of the whole R. But that is not a problem, we can just discard all returned segments containing points with x-coordinates outside of the interval of R', because if there is a chain of length k_2 inside R', then it must be covered by a chain of length k_1 also inside R'. The analysis of correctness of Merge still holds. The second issue is that while previously it was enough to analyse only the depth of a set of segments returned by Merge, now we need to analyse the depth of the whole cover.

Lemma 6.22. *For any rectangle R, the depth of any cover in $\mathsf{CF}(R)$ is $\mathcal{O}(\epsilon^{-1} \log^2 n)$.*

Proof. We use induction on the height in the secondary recursion tree. Let h be the secondary height of R. It was already shown in the proof of Lemma 6.12 that, due to sparsification, Merge adds only $\mathcal{O}(\epsilon^{-1} \log n)$ new segments, or at most $c'\epsilon^{-1} \log n$ for some constant c'. Now we want to show that the depth of any cover in $\mathsf{CF}(R)$ is at most $1 + c'h\epsilon^{-1} \log n$. The base case of leaf rectangles containing just one point is trivial. Otherwise, a cover in $\mathsf{CF}(R)$ is obtained by joining BSTs of the left and right rectangle, then adding segments returned by Merge (and possibly deleting some segments along the way). As the secondary height of both the left and right rectangle is at most $h - 1$, and Merge returns no more than $c'\epsilon^{-1} \log n$ new segments, the depth of the obtained cover is at most $1 + c'h\epsilon^{-1} \log n$ as claimed. Finally, as $h = \mathcal{O}(\log n)$ we obtain the lemma. □

Approximation factor. Here we analyse the approximation guarantee of the modified algorithm. Recall that a (k_1, k_2)-cover in the covering family of a rectangle R is recomputed after roughly every $\epsilon' k_1$ operations inside R. In the decremental structure, the cover was calculated with a suitable margin, namely by constructing a $(k_1\lambda, k_2)$-cover, so that it remained a valid (k_1, k_2)-cover even after $\epsilon' k_1$ deletions. In the fully dynamic structure, we need to add such a margin on both sides. Namely, we construct a $(k_1\lambda, k_2/\lambda)$-cover, which remains a valid (k_1, k_2)-cover after any sequence of up to $\epsilon' k_1$ deletions or insertions. This increases the approximation guarantee only by a constant factor, which is formalised in the lemma below. The proof very much replicates that of Lemma 6.14.

Lemma 6.23. *For any rectangle R' with primary height h, $\mathsf{CF}(R')$ is a $(\lambda^2, 3h)$-covering family.*

Proof. This is proved by induction first on the primary height, then the secondary height. Rectangles of zero height contain a single point and the claim is trivial. Let us now consider rectangle R' of primary height h, with left/right rectangles R'_l/R'_r. We also have bottom/top rectangles R_b/R_t constituting a partition of the rectangle R corresponding to the root of the secondary

recursion tree of R'. From the assumption, R'_l and R'_r provide $(\lambda^2, 3h)$-covering families, while R_b and R_t provide $(\lambda^2, 3h-3)$-covering families. The first $k = 3/\epsilon'$ levels in $\mathsf{CF}(R')$ are exact covers and they are correct by Lemma 6.11.

Let us focus on a level providing a $(\lceil k\lambda^{2j}\rceil, k\lambda^{2(j+3h-1)})$-cover for some $j > 0$. Segments coming from the same level in R'_l and R'_r have the correct score. We need to consider segments coming from $\mathsf{Merge}(\mathsf{CF}(R_b), \mathsf{CF}(R_t), x_m, k\lambda^{2j+2}, \texttt{true})$, where x_m is the middle x-coordinate of R'. $\mathsf{CF}(R_b)$ and $\mathsf{CF}(R_t)$ provide λ^{6h-6}-approximation. Say that we have an interval crossing x_m and containing a chain X of length at least $k\lambda^{2j+6h-3}$. By Lemma 6.10, in the set returned by Merge we have a chain of length $\lceil k\lambda^{2j+1}\rceil$ covering X.

Because of amortising Merge, there might be at most $\lfloor \epsilon' \lceil k\lambda^{2j}\rceil \rfloor - 2$ deletions or insertions of points in R' before the whole level of the covering family is recomputed. Thus, the guarantee on the length of a chain is at least $\lceil k\lambda^{2j}\rceil$, as in Lemma 6.14.

If after $\lfloor \epsilon' \lceil k\lambda^{2j}\rceil \rfloor - 2$ deletions or insertions of points in R' there exists an interval crossing x_m and containing chain of length $k\lambda^{2(j+3h-1)}$, then at the time of recomputing this interval contained a chain of length at least $k\lambda^{2j+6h-3}$, as $k\lambda^{2j+6h-2} - \lfloor \epsilon' \lceil k\lambda^{2j}\rceil \rfloor + 2 > \lambda^{6h-2}k\lambda^{2j} - (\lambda - 1)k\lambda^{2j} > k\lambda^{2j}(\lambda^{6h-2} - \lambda + 1) > k\lambda^{2j+6h-3}$ for $1 < \lambda < 2$. □

Because of the above and also changed depth of the recursion, which is $\log_{3/2} n$ now, we set ϵ' to be not $\epsilon/(8\log n)$ as before, but rather use $\epsilon' = \epsilon/(12\log_{3/2} n)$. This changes only the constants and we still have $1/\epsilon' = \mathcal{O}(\epsilon^{-1}\log n)$.

Time complexity. Finally, we analyse the time complexity of the modified algorithm. Observe that increasing the approximation factor and the height of the recursive structure of rectangles only affects constants in the running time. However, now Merge is slower by a factor of $\mathcal{O}(\log n)$, as more segments cross the middle x-coordinate of a rectangle. We also need to account for recomputing the covering family of a rectangle R and every rectangle below in the recursive structure whenever the number of insertions and deletions in $P(R)$ is sufficiently large. This does not dominate the running time, though, as the cost of preprocessing is smaller than the cost of maintaining covering families. The time complexities are summarised in the lemma below. Its proof very much replicates that of Lemma 6.17.

Lemma 6.24. *The amortised complexity of deleting or inserting a point is* $\mathcal{O}(\epsilon^{-5}\log^{11} n)$.

Proof. When deleting an element, the algorithm needs to visit $\mathcal{O}(\log^2 n)$ rectangles that contain it. For each such rectangle R, we do the following:

- Delete the element from the BST storing points inside R in time $\mathcal{O}(\log n)$.
- Increment the counter of every level of $\mathsf{CF}(R)$, there are $\mathcal{O}(\epsilon^{-1}\log^2 n)$ levels.
- Pay the amortised cost of recomputing every level of $\mathsf{CF}(R)$, now in total this is $\mathcal{O}(\epsilon^{-5}\log^9 n)$ instead of $\mathcal{O}(\epsilon^{-5}\log^8 n)$, because the depth of the middle x-coordinate is $\mathcal{O}(\epsilon^{-1}\log^2 n)$ instead of $\mathcal{O}(\epsilon^{-1}\log n)$.

- For every level of $\mathsf{CF}(R)$, we join BSTs from $\mathsf{CF}(R_l), \mathsf{CF}(R_r)$ with the segments returned by the most recent call to Merge. This takes time $\mathcal{O}(\epsilon^{-2} \log^4 n)$ in total.

- Check if there have been sufficiently many insertions or deletions of points in R, and if so recompute the whole recursive structure below R. If R contains m points, then as in Lemma 6.18 this is done in time $\mathcal{O}(m\epsilon^{-2} \log^6 n)$ (even less if R is not a rectangle in the primary recursion tree). The recomputation is done after $\Theta(m)$ deletions or insertions, thus the amortised time is $\mathcal{O}(\epsilon^{-2} \log^6 n)$. □

Deamortisation. The final step is deamortising the fully dynamic structure. Recall that we have used amortisation in three places: running Merge in order to maintain a cover, recomputing the whole secondary recursive structure below rectangle R', and recomputing the whole primary recursive structure below R. Additionally, we have implicitly assumed that the value of $\log n$ does not change during the execution of the algorithm, so we need to rebuild the whole structure once it changes by a constant factor (but this is standard, and we will not mention this issue again). In all three places, we run the computation only once sufficiently many updates have been performed in the corresponding structure (the structure being either a rectangle in the primary/secondary recursion or a level of some covering family). As usually, instead of performing the whole computation immediately, we would like to distribute it among the next updates.

In the following, we assume we have a fixed value of ϵ and n is the number of items in the main array; as noted above, the value of $\log n$ does not change during the execution of the algorithm. First, let us focus on some (k_1, k_2)-cover in a covering family of a rectangle R and all calls to Merge used to maintain it. In our solution, Merge works in $c_{n,\epsilon}(k_1) = \mathcal{O}(k_1 \epsilon^{-3} \log^6 n)$ time, and is performed after every $b_{n,\epsilon}(k_1) = \epsilon' k_1$ updates in R (here we ignore additive constants). After each update in R, we join trees from the left and right rectangle with the segments returned by the most recent call to Merge in time $u_{n,\epsilon}(k_1) = \mathcal{O}(\epsilon^{-2} \log^4 n)$, but note that we do not modify the set of segments returned by Merge. At any point, queries to the cover take time $q_{n,\epsilon}(k_1) = \mathcal{O}(\log n)$. This way, we achieved the amortised update cost of $u_{n,\epsilon}(k_1) + c_{n,\epsilon}(k_1)/b_{n,\epsilon}(k_1) = \mathcal{O}(\epsilon^{-4} \log^7 n)$.

Before we can deamortise this, we need to slightly modify the implementation of Merge due to the following reason. The computation needs to access some other covering families and extract segments from their covers. However, because now we want to run it in the background while updates to other structures are possibly taking place, it is not clear what are the guarantees on the retrieved information. Therefore, we always immediately execute the first part of Merge (up to line 23) and gather the relevant segments, each containing a list of elements that should be added to a set of points P. This takes only $\mathcal{O}(\epsilon^{-2} \log^5 n)$ time, which is negligible. We assume that these lists are not destroyed if pointed to in some part of the structure, so we can think that the second part of Merge is a local computation that does not need to access any other information. Similarly, recomputing the whole secondary recursive structure below R' and the whole primary recursive structure below R can be implemented as a local computation, as each rectangle stores $P(R)$ in a persistent BST (and no other information is required).

Now, to maintain a (k_1, k_2)-cover of R in worst-case update time, we use the current cover A while constructing the next cover B in the background. The queries are answered using A, and the transition between A and B consists of two phases. For the first $b_{n,\epsilon}(k_1)/3$ updates, nothing is done for B, and for A standard updates are performed in time $u_{n,\epsilon}(k_1)$. Next, as mentioned above, in time $\mathcal{O}(\epsilon^{-2}\log^5 n)$ we immediately execute the first part of Merge, which gathers the relevant segments. In the second phase, spanning the next $b_{n,\epsilon}(k_1)/3$ updates, we execute the remaining part of Merge. Namely, after every update in A, we run $3c_{n,\epsilon}(k_1)/b_{n,\epsilon}(k_1)$ steps of Merge. At the end of the second phase B is ready, so it replaces A, which is no longer needed. In total, $b_{n,\epsilon}(k_1)/3$ updates in R passed from when we started constructing B to when we started querying it. Therefore, we still can query B between the next $2b_{n,\epsilon}(k_1)/3$ updates needed to transition it into yet another cover. Now, every update takes worst-case time $u_{n,\epsilon}(k_1) + \mathcal{O}(c_{n,\epsilon}(k_1)/b_{n,\epsilon}(k_1)) = \mathcal{O}(\epsilon^{-4}\log^7 n)$, which is higher than amortised time only by a constant factor.

Next, we move to the local computation performed in the primary or secondary recursive structure, which can be deamortised using a similar approach. Let us focus on a rectangle in the primary recursive structure, as the other case is simpler, with the only difference being smaller construction time. Recall that we have a fixed value of ϵ and n is the number of items in the main array. Consider a rectangle initially storing $m \leq n$ items, for which the primary recursive structure can be constructed in time $c_{n,\epsilon}(m) = \mathcal{O}(m\epsilon^{-2}\log^6 n)$, and updated in time $u_{n,\epsilon}(m) = \mathcal{O}(\epsilon^{-5}\log^9 n)$ as long as the number of updates does not exceed $b_{n,\epsilon}(m) = m/4$. Covers in the covering family of that rectangle can be queried in $q_{n,\epsilon}(m) = \mathcal{O}(\log m) = \mathcal{O}(\log n)$ time. We obtained an amortised structure with no upper bound on the number of updates by simply rebuilding it every $b_{n,\epsilon}(m)$ updates, resulting in amortised update time $\mathcal{O}(c_{n,\epsilon}(2m)/b_{n,\epsilon}(m/2) + u_{n,\epsilon}(2m))$ and $q_{n,\epsilon}(m)$ query time, where m is the current number of items.

To obtain worst-case update time, we maintain two structures A and B, answering queries with A. Assume that A was initially constructed with m_A points. Whenever an update arrives, it is immediately executed in A. Transition between A and B consists of three phases:

- For the first $b_{n,\epsilon}(m_A)/4$ updates, B is not initialised yet. Assume that after that, A stores m_B items.
- In the second phase, we construct B storing a set of those m_B items, over at most $b_{n,\epsilon}(m_A)/8$ updates. Namely, after every update in A, we run $8c_{n,\epsilon}(m_B)/b_{n,\epsilon}(m_A)$ steps of the construction algorithm for B, if it has not finished work yet. As $m_B < 2m_A$, this is $\mathcal{O}(c_{n,\epsilon}(2m_A)/b_{n,\epsilon}(m_A))$ steps after every update. Additionally, in this and the next phase, every arriving update is added to a queue of pending updates of B. This is different from the case of Merge, as now updates needs to be performed in both A and B.
- In the third phase, after every update in A, we execute two pending updates in B, so after $b_{n,\epsilon}(m_A)/8$ updates the queue of pending updates becomes empty. Then, B replaces A.

In total, $b_{n,\epsilon}(m_A)/2$ updates are performed during the transition from A to B. Observe that in B, $b_{n,\epsilon}(m_A)/4$ updates were already performed, so we can use B for less than $b_{n,\epsilon}(m_B)$

another updates. We defined $b_{n,\epsilon}(m) = m/4$, and due to the length of the first phase we have $m_B \geq m_A - b_{n,\epsilon}(m_A)/4 = 15m_A/16$. Thus, it holds that:

$$\begin{aligned} b_{n,\epsilon}(m_B) - b_{n,\epsilon}(m_A)/4 &= b_{n,\epsilon}(m_B) - m_A/16 \\ &\geq m_B/4 - m_B/15 = 11m_B/60 > m_B/8 = b_{n,\epsilon}(m_B)/2. \end{aligned}$$

This means we still can query B during the next $b_{n,\epsilon}(m_B)/2$ updates needed to transition it into yet another structure. Updates take worst-case time $\mathcal{O}(u_{n,\epsilon}(2m) + c_{n,\epsilon}(2m)/b_{n,\epsilon}(m/2)) = \mathcal{O}(\epsilon^{-5}\log^9 n)$ and query is still in time $q_{n,\epsilon}(m) = \mathcal{O}(\log n)$.

6.7 Conditional lower bounds

We consider sets of 2D points with weights. The weight of a chain is the sum of weights of its points. $P = (p_0, p_1, \ldots, p_k)$ is an antichain if for all $0 \leq i, j \leq k$, $p_i \not\prec p_j$.

When we wish to compute the longest chain in a set of points, we simply arrange its points in an array, sorted by the x-coordinates, and then compute LIS with respect to the y-coordinates. Maintaining such an array for a dynamic set of points is easily realised in logarithmic time with any BST supporting $\mathsf{FindRank}(r)$ operation, which returns a pointer to an item of rank r in the set of items in BST. Then we can find the value of the current element at index r, delete the current element at index r, or insert a new element strictly before or after the current element at index r, all with logarithmic overhead.

Two popular conjectures. For the weighted case, we assume the following conjecture on the well-known APSP problem:

Conjecture 6.25. *There exists no algorithm solving the all-pairs shortest paths (APSP) problem in general weighted graphs in time $\mathcal{O}(n^{3-\epsilon})$, for any constant $\epsilon > 0$.*

We modify constructions from [AD16, GPPR04], originally designed for distances in planar graphs. Reduction will be from the (max,+)-matrix-product problem, denoted by $\circ$, which is known to be equivalent to APSP [FM71]. In this problem we are given $n \times n$ matrices A, B having integer weights in $\{0, \ldots, M\}$ and want to compute matrix C, with $C_{i,j} = \max_k(A_{i,k} + B_{k,j})$. As those matrices are weighted, for the unweighted case we need a weaker conjecture on Boolean variables. In the online Boolean matrix-vector multiplication (OMv) problem we are given $n \times n$ matrix A, and let $v_1, \ldots, v_n$ be Boolean vectors arriving online. We need to preprocess A and then for every i output Av_i before seeing v_{i+1}. It was conjectured in [HKNS15] (see also [LW17, CKL18]) that:

Conjecture 6.26. *There exists no algorithm solving the OMv problem in time $\mathcal{O}(n^{3-\epsilon})$, for any constant $\epsilon > 0$.*

6.7.1 The Embedding of a Matrix

Assume for now that matrices A, B are Boolean. Columns and rows are numbered from 0 to $n-1$. For each $0 \leq k < n$, we define embedding S_k of A and the k-th column of B, B_k, as the union of six sets of points, namely $S_k = S_l \cup S'_l \cup S_r \cup S_a \cup S'_a \cup S_{b,k}$. In total, S_k contains $3n^2 + 3n$ points. It consists of the left and right side, and three sets of special points. It is best to inspect Figure 6.5 before reading further.

On the left side, we imagine there is a $n \times n$ grid. There are two sets of points S_l and S'_l, both having one point in each cell of this grid. For each column or row, points of S_l form one chain, while points from S'_l form one antichain. Additionally, in each cell the point from S_l is above and to the left of the point from S'_l. As for the weights, in column j points from S_l have weights $3(n-j)$, and in cell i, j the point from S'_l has the weight $3(n-j) + A_{j,i}$, so depending on the value from transposed matrix A. Formally, for each $0 \leq i, j < n$, S_l contains the point $l_{i,j} = (j(2n+1) + i + 2, i(3n+1) + 2n + j + 1)$ with the weight $3(n-j)$, and S'_l contains the point $l'_{i,j} = ((j+1)(2n+1) - i, i(3n+1) + 2n - j)$ with the weight $3(n-j) + A_{j,i}$. The first special set of points S_a is also defined using the left grid. All points from S_a are below points from S_l and S'_l, additionally j-th point is to the left of all points from j-th column of the grid and to the right of points from $(j-1)$-th column. All these points have the weight 1 and form an antichain. Formally, for each $0 \leq j < n$, S_a contains a point $a_j = (j(2n+1) + 1, n - j)$ with the weight 1.

On the right side, we also have $n \times n$ grid and the set of points S_r, which is basically S_l rotated by 180 degrees. In each cell of the grid, there is one point from S_r, and points in columns or rows form chains, but weights in column j are $3(j+1)$. Formally, for each $0 \leq i, j < n$, S_r contains a point $((2n+j)(n+1) + i + 1, (i+1)(3n+1) + j + 1)$ with the weight $3(j+1)$. The second special set of points S'_a is defined using the right grid. All points from S'_a are above points from S_r and additionally, j-th point is to the right of all points from j-th column of the grid and to the left of points from $(j+1)$-th column. All these points have the weight of 1 and form an antichain. Formally, for each $0 \leq j < n$, S'_a contains a point $a'_{n-j-1} = ((2n + j + 1)(n+1), 3n(n+1) - j)$ with the weight 1.

The last special set of points is $S_{b,k}$. Those points form one antichain and are placed between the left and right grids. i-th point is above all the points from the i-th row from the left grid and below all the points from the i-th row of the right grid. The weight of i-th point is just $B_{i,k}$. Formally, for each $0 \leq i < n$, $S_{b,k}$ contains a point $b_i = (2n(n+1) - i, (i+1)(3n+1))$ with the weight $B_{i,k}$.

Note that for $k \neq k'$, embeddings S_k and S'_k differ only on weights of n points from $S_{b,k}$ and $S_{b,k'}$.

Properties of the longest chains. Given S_k we need to inspect the maximum weight of a chain starting with a_j and ending with a'_j, for any j. Observe that there is such a maximum chain containing a point from $S_{b,k}$, thus we can focus on chains from a_j to b_i and from b_i to a'_j, for any i.

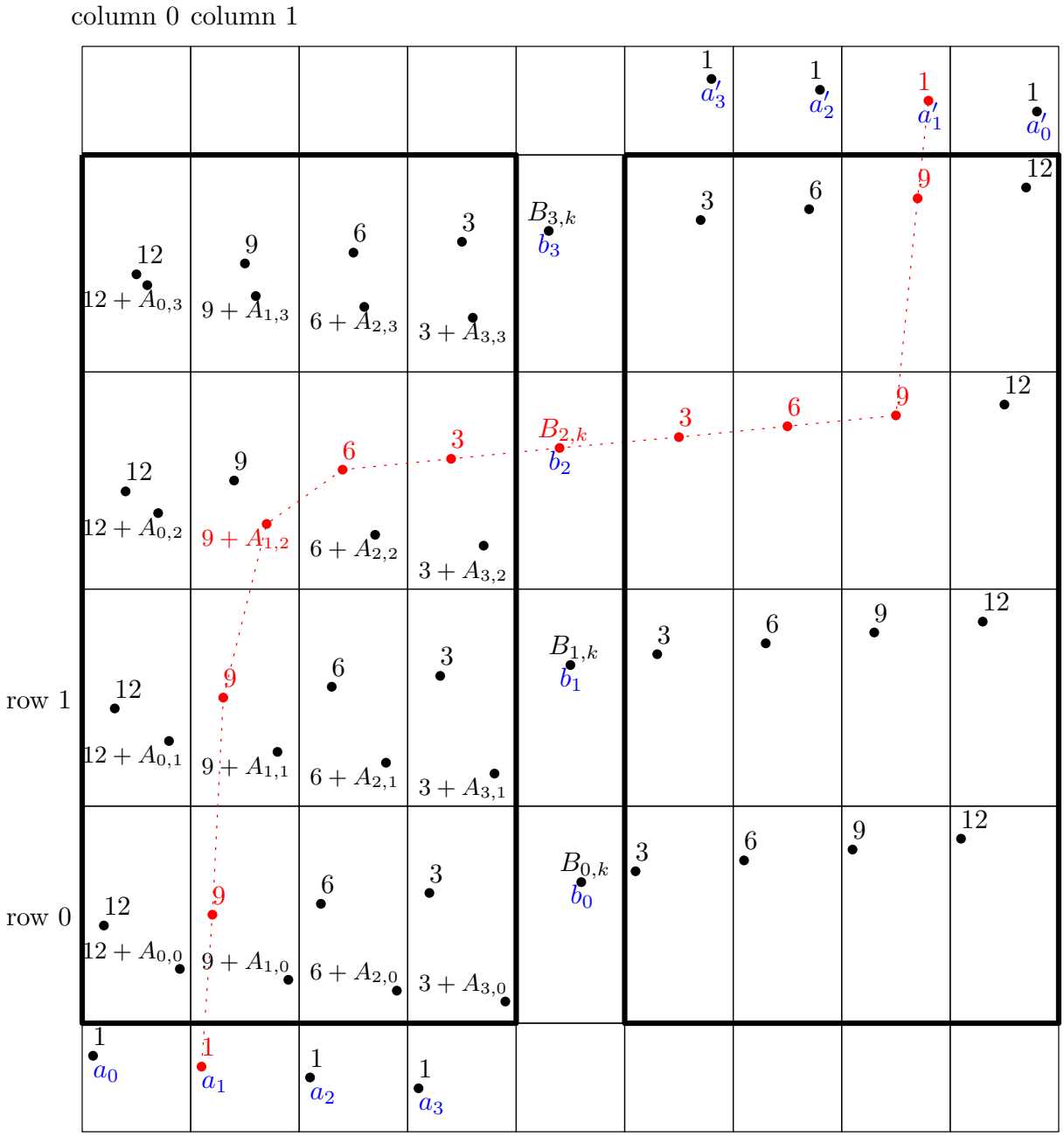


Figure 6.5: Embedding of matrix A and the k-th column of matrix B, with $n = 4$. Each point is given an integer weight. Special points $a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, a'_0, \ldots, a'_{n-1}$ are marked in blue. On the left side, weights of points use weights from transposed A. Turns allow to gain $A_{i,j}$, but taking them too early is not beneficial due to diminishing weights in columns. In red, the maximum weight chain from a_1 to a'_1 is highlighted. The right side of the embedding makes the choice of b_i dependent only on values from matrices.

The main idea here is that the maximum weight chain should use at most one point from S'_l. Those points can be seen as 'turns', meaning that if point p in column j from S'_l is taken, then the chain cannot contain any point from column j above p. It is not beneficial to take such a turn too quickly, since by changing a column early one can gain at most 2 (from two non-zero entries in a matrix), at the same time losing at least 3 from diminishing weights in columns.

Let $c_k(p_1, p_2)$ be the maximum weight of a chain starting with a point p_1 and ending with a point p_2, in set S_k. Then we have:

Lemma 6.27. *Consider any $0 \le i, i', j, k < n$, with $i \le i'$.*

1. *If $i = i'$, $c_k(l_{i,j}, b_{i'}) = 1.5(n-j)(n-j+1) + B_{i',k}$.*
2. *If $i < i'$, $c_k(l_{i,j}, b_{i'}) = (3n-3j)(i'-i) + 1.5(n-j)(n-j+1) + A_{j,i'} + B_{i',k}$.*
3. *If $i = i'$, $c_k(l'_{i,j}, b_{i'}) = 1.5(n-j)(n-j+1) + A_{j,i} + B_{i',k}$.*
4. *If $i < i'$, $c_k(l'_{i,j}, b_{i'}) = (3n-3j-3)(i'-i) + 1.5(n-j)(n-j+1) + A_{j,i} + A_{j+1,i'} + B_{i',k}$, where we assume $A_{n,\cdot} = 0$.*

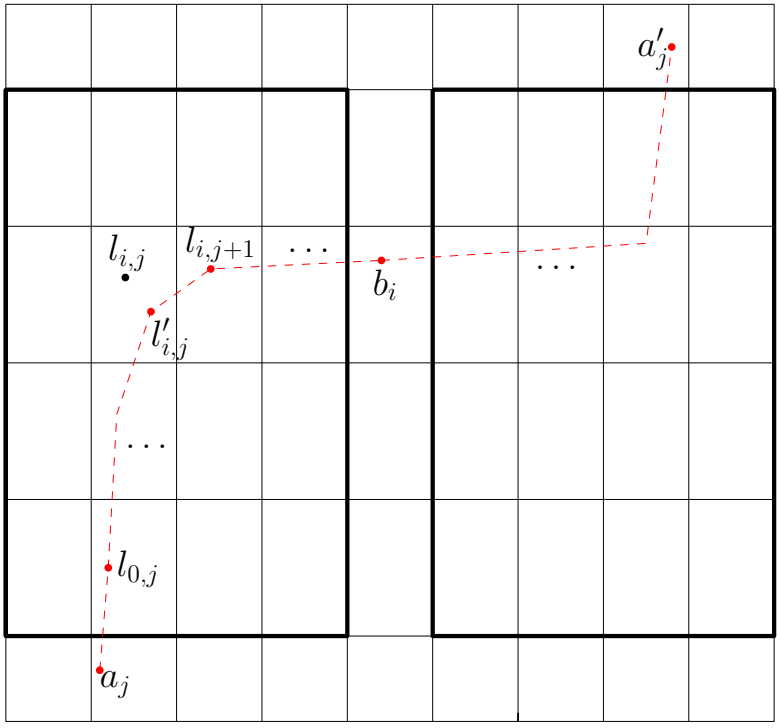


Figure 6.6: Simplified representation of the maximum chain from a_j to a'_j.

Proof. The proof, for any k and i', is by induction on decreasing j and then decreasing i. For $j = n-1$ the claim is easy to verify — the maximum chain from $l'_{i,n-1}$ to $b_{i'}$ contains only two points, and from $l_{i,n-1}$ to $b_{i'}$ it is a chain containing, except endpoints, $i'-i-1$ points from S_l in a column $n-1$, and $l'_{i',n-1}$. The case of $i = i'$ is also straightforward to verify, as then we have just a chain in a single row of the grid.

Now assume $i < i', j < n-1$. For a maximum chain starting with $l'_{i,j}$, the next point in the chain must be $l_{i,j+1}$, thus by induction the maximum weight is

$$(3(n-j)+A_{j,i})+(3n-3j-3)(i'-i)+1.5(n-j-1)(n-j)+A_{j+1,i'}+B_{i',k}=$$
$$(3n-3j-3)(i'-i)+1.5(n-j)(n-j+1)+A_{j,i}+A_{j+1,i'}+B_{i',k},$$

which gives us equality 4.

Now assume $i = i'-1, j < n-1$. For a maximum chain starting at $l_{i,j}$, the next point in the chain must be $l_{i',j}$, $l'_{i',j}$ or $l_{i,j+1}$. Thus, by induction its weight is

$$\begin{aligned}
&3(n-j)+\max\{1.5(n-j)(n-j+1)+B_{i',k},\\
&1.5(n-j)(n-j+1)+A_{j,i'}+B_{i',k},\\
&(3n-3j-3)+1.5(n-j-1)(n-j)+A_{j+1,i'}+B_{i',k}\}=\\
&3(n-j)+1.5(n-j)(n-j+1)+B_{i',k}+\max\{0,A_{j,i'},A_{j+1,i'}-3\}=\\
&(3n-3j)(i'-i)+1.5(n-j)(n-j+1)+A_{j,i'}+B_{i',k}.
\end{aligned}$$

Now assume $i < i'-1, j < n-1$. For a maximum chain starting at $l_{i,j}$, the next point in the chain must be $l_{i+1,j}$, $l'_{i+1,j}$ or $l_{i,j+1}$. Thus, by induction its weight is

$$\begin{aligned}
&3(n-j)+\max\{(3n-3j)(i'-i-1)+1.5(n-j)(n-j+1)+A_{j,i'}+B_{i',k},\\
&(3n-3j-3)(i'-i-1)+1.5(n-j)(n-j+1)+A_{j,i+1}+A_{j+1,i'}+B_{i',k},\\
&(3n-3j-3)(i'-i)+1.5(n-j-1)(n-j)+A_{j+1,i'}+B_{i',k}\}=\\
&3(n-j)+(3n-3j)(i'-i-1)+1.5(n-j)(n-j+1)+B_{i',k}\\
&+\max\{A_{j,i'},3i-3i'+3+A_{j,i+1}+A_{j+1,i'},3i-3i'+A_{j+1,i'}\}=\\
&(3n-3j)(i'-i)+1.5(n-j)(n-j+1)+A_{j,i'}+B_{i',k},
\end{aligned}$$

as $i'-i>1$ and matrices are Boolean. Thus, we get the remaining equality 2. □

From Lemma 6.27 we immediately have:

Corollary 6.28. *For any* $0 \le i,j,k < n$, $c_k(a_j,b_i) = (3n-3j)i+1.5(n-j)(n-j+1)+1+A_{j,i}+B_{i,k}$.

The next lemma for weights in the simpler right grid is straightforward to prove:

Lemma 6.29. *For any* $0 \le i,j,k < n$, $c_k(b_i,a'_j) = (3n-3j)(n-i-1)+1.5(n-j)(n-j+1)+1+B_{i,k}$.

Observe that in S_k there is a maximum chain from a_j to a'_j containing a point from $S_{b,k}$, as there is a point from $S_{b,k}$ between any two points from the left and right grid forming a chain. Thus, from Corollary 6.28 and Lemma 6.29, we have:

Lemma 6.30. *For any* $0 \le j,k < n$, $c_k(a_j,a'_j) = \max_i\{(3n-3j)(n-1)+3(n-j)(n-j+1)+2+A_{j,i}+B_{i,k}\} = (3n-3j)(n-1)+3(n-j)(n-j+1)+2+\max_i(A_{j,i}+B_{i,k})$.

Therefore, we achieved that for the maximum chain from a_j to a'_j going through b_i, the only dependence on i are values from matrices. Moreover, observe that in any S_k the maximum weight chain on points with X-coordinates between a_j and a'_j is $c_k(a_j, a'_j)$, that is, such a chain starts with a_j and ends with a'_j. This is true because any chain can contain at most one point from S_a and at most one point from S'_a, and when we consider chains on points with X-coordinates between a_j and a'_j, we can always either replace some point $a_{j'}$ with $j' > j$ by a_j or just add a_j as the first element of a chain, and similarly for a'_j. This will allow us to compute (max,+)-product of matrices using LIS supporting 1D queries.

Weighted matrices. The previous embedding for Boolean matrices extends naturally to the case of A and B having integer weights in $\{0, \ldots, M\}$ for some $M \in \text{poly}(n)$. We only need to multiply the previous weights of points in sets S_l and S_r by M, and change weights of point $l'_{i,j}$ in set S'_l to $3M(n-j) + A_{j,i}$. All logic on the maximum chains still applies, and in the weighted case the following can be proven:

Lemma 6.31. *For any* $0 \leq j, k < n$, $c_k(a_j, a'_j) = M(3n-3j)(n-1) + 3M(n-j)(n-j+1) + 2 + \max_i(A_{j,i} + B_{i,k})$.

6.7.2 Lower Bounds

Dynamic weighted LIS. Using embeddings defined in the previous section, it is straightforward to obtain a lower bound for weighted LIS:

Theorem 6.32. *There is no algorithm for dynamic weighted LIS with the amortised update and query time* $\mathcal{O}(n^{1/2-\epsilon})$ *for any constant* $\epsilon > 0$, *unless Conjecture 6.25 is false. This holds even if only weight updates are allowed.*

Proof. We reduce from (max,+)-product using defined embeddings. We are given $n \times n$ matrices A, B having integer weights in $\{0, \ldots, M\}$ and want to compute C, with $C_{i,j} = \max_k(A_{i,k} + B_{k,j})$. Performing $\mathcal{O}(n^2)$ updates, we insert all the points from embedding S_1 of A and B_1. Then we make n queries, for each $0 \leq j < n$ asking about the maximum weight of a chain on points with X-coordinates between a_j and a'_j. From Lemma 6.31, this is $M(3n-3j)(n-1) + 3M(n-j)(n-j+1) + 2 + \max_i(A_{j,i} + B_{i,1})$. As only the last part of this formula is not fixed, we can extract from the answer the value of $C_{j,1}$. Thus, after n queries we have the first column of C computed. Then we update weights of points in $S_{b,1}$ to get $S_{b,2}$, which transforms S_1 into S_2. This process is repeated until we get the whole C. The cost is $\mathcal{O}(n^2)$ initial updates, then in total another n^2 updates and n^2 queries. As the size of the LIS instance at any time is $\mathcal{O}(n^2)$ points, claimed conditional bound is proven. □

Dynamic unweighted LIS supporting 1D queries. For the unweighted case, we cannot use APSP conjecture anymore and need to switch to the weaker OMv Conjecture 6.26. Here we define S_k as an embedding of matrix A and vector v_k.

Theorem 6.33. *There is no algorithm for dynamic (unweighted) LIS supporting 1D queries with amortised update and query time $\mathcal{O}(n^{1/3-\epsilon})$ for any constant $\epsilon > 0$, unless Conjecture 6.26 is false.*

Proof. In this paragraph, we use m to denote the size of the matrix and a vector. Observe that the sum of weights in any embedding S_k of Boolean matrix A of size $m \times m$ and vector v_k of size m is $\mathcal{O}(m^3)$, as the largest weight is $\mathcal{O}(m)$. Therefore, by replacing each weighted point with weight w by a chain of w unweighted points, embedding S_k becomes an instance of unweighted LIS on $\mathcal{O}(m^3)$ points. Importantly, transforming S_k into S'_k still takes only $\mathcal{O}(m)$ updates, since only m points in $S_{b,k}$ needs to be deleted or inserted. Observe that we cannot hope to beat the trivial algorithm for OMv running in cubic time if we create an embedding of the whole input matrix, thus we will divide this matrix into smaller square pieces.

Suppose there is an algorithm for dynamic (unweighted) LIS supporting 1D queries with amortised update and query time $\mathcal{O}(n^{1/3-\epsilon})$, for some constant $\epsilon > 0$, when run on an instance with up to n points. Let $\epsilon' = 1.5\epsilon$. Divide A into m submatrices $A_{i,j}$ of size $m^{0.5} \times m^{0.5}$, and each vector v_k into $m^{0.5}$ subvectors $v_{k,j}$ of size $m^{0.5}$. Then $Av_k = u_k = (u_{k,1}, \ldots, u_{k,m^{0.5}})$, where $|u_{k,i}| = m^{0.5}$ and $u_{k,i}(j) = \max(0, \max_l((A_{i,l} \circ v_{k,l})(j)) - 1)$, as Boolean multiplication here translates to the (max,+)-product.

Define $S_{k,i,j}$ as an embedding of $A_{i,j}$ and $v_{k,j}$, for $0 \leq j, i < m^{0.5}$ and $0 \leq k < n$. Total cost of updates to create all $S_{1,.,.}$ is $\mathcal{O}(m^{3-\epsilon'})$, as there are m embeddings, each with $\mathcal{O}(m^{1.5})$ points. To recover Av_1, we compute all $A_{i,j} \circ v_{1,j}$ by making $m^{1.5}$ queries, each in an instance with $\mathcal{O}(m^{1.5})$ points, which in total takes time $\mathcal{O}(m^{2-\epsilon'})$. Then by taking $m^{1.5}$ maximums we can compute $u_1 = Av_1$. Transforming all of the $S_{1,i,j}$ into $S_{2,i,j}$, for $0 \leq i, j < m^{0.5}$, takes in total time $\mathcal{O}(m^{2-\epsilon'})$. Then the process is repeated, allowing us to compute all u_k one by one in an online fashion. Total time is $\mathcal{O}(m^{3-\epsilon'})$, which is a contradiction under Conjecture 6.26, thus assumed algorithm for dynamic LIS could not exist. □

Finally, we note that in both presented lower bounds we can achieve simple trade-offs between bounds on query and update time, as described in [AD16].

6.8 Conclusions and Open Problems

We constructed a dynamic algorithm providing arbitrarily small constant factor approximation of the longest increasing subsequence, capable of deleting and inserting elements anywhere inside the sequence in polylogarithmic time. It improves on the previous result [MS20a] providing these operations in time $\mathcal{O}(n^{\epsilon})$ and with a higher approximation factor. We believe that our notion of a covering family, and the greedy procedure for obtaining one, might be of independent interest.

Still, many aspects of this problem remain unexplored. The best upper bound on exact dynamic LIS is $\mathcal{O}(n^{2/3})$ [KS21], but there are no known lower bounds on *exact* dynamic algorithm for LIS better than $\Omega(\log n)$. We provide polynomial conditional lower bounds on variants of dynamic exact LIS [GJ21a], where the algorithm must be capable of reporting LIS of any

continuous subsequence of the array (note the algorithm presented in this chapter provides approximation of this quantity) or when elements are weighted. For approximated solutions, it should be relatively easy to decrease exponents in time complexity of our algorithm by one or two, but it seems that achieving more practical time complexity, say $\mathcal{O}(\epsilon^{-3}\log^3 n)$, would require a new approach or ideas.

Bibliography

[AA02] Noga Alon and Vera Asodi. Sparse universal graphs. *Journal of Computational and Applied Mathematics*, 142:1–11, May 2002.

[AAK+06] Serge Abiteboul, Stephen Alstrup, Haim Kaplan, Tova Milo, and Theis Rauhe. Compact labeling scheme for ancestor queries. *SIAM J. Comput.*, 35(6):1295–1309, 2006.

[AB18] Amir Abboud and Karl Bringmann. Tighter connections between formula-SAT and shaving logs. In *45th ICALP*, pages 8:1–8:18, 2018.

[ABR05] Stephen Alstrup, Philip Bille, and Theis Rauhe. Labeling schemes for small distances in trees. *SIAM Journal on Discrete Mathematics*, 19(2):448–462, 2005.

[ABW15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *56th FOCS*, pages 59–78, 2015.

[AC07] Noga Alon and Michael R. Capalbo. Sparse universal graphs for bounded-degree graphs. *Random Struct. Algorithms*, 31(2):123–133, 2007.

[ACJS21] Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct encodings for families of interval graphs. *Algorithmica*, 83(3):776–794, 2021.

[AD99] D. Aldous and P. Diaconis. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bull. Amer. Math. Soc. (N.S.)*, 36(4):413–432, 1999.

[AD16] Amir Abboud and Søren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *FOCS*, pages 477–486. IEEE Computer Society, 2016.

[ADK15] 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.

[ADKP16] Stephen Alstrup, Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Ely Porat. Sublinear distance labeling. In *24th ESA*, volume 57, pages 5:1–5:15, 2016.

[AGHP16a] 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.

[AGHP16b] Stephen Alstrup, Inge Li Gørtz, Esben Bistrup Halvorsen, and Ely Porat. Distance labeling schemes for trees. In *43rd ICALP*, pages 132:1–132:16, 2016.

[AHL14] Stephen Alstrup, Esben Bistrup Halvorsen, and Kasper Green Larsen. Near-optimal labeling schemes for nearest common ancestors. In *25th SODA*, pages 972–982, 2014.

[AKTZ15] Stephen Alstrup, Haim Kaplan, Mikkel Thorup, and Uri Zwick. Adjacency labeling schemes and induced-universal graphs. In *47th STOC*, pages 625–634, 2015.

[Alo17] Noga Alon. Asymptotically optimal induced universal graphs. *Geometric and Functional Analysis*, 27:1–32, February 2017.

[AN17] Noga Alon and Rajko Nenadov. Optimal induced universal graphs for bounded-degree graphs. In *28th SODA*, pages 1149–1157, 2017.

[AOSS18] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *STOC*, pages 815–826. ACM, 2018.

[AOSS19] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In *SODA*, pages 1919–1936. SIAM, 2019.

[AR02a] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In *13th SODA*, pages 947–953, 2002.

[AR02b] Stephen Alstrup and Theis Rauhe. Small induced-universal graphs and compact implicit graph representations. In *43rd FOCS*, pages 53–62, 2002.

[AV96] Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory (extended abstract). In *37th FOCS*, pages 560–569. IEEE Computer Society, 1996.

[AVL62] George M. Adelson-Velski and Evgenii M. Landis. An algorithm for organization of information. *Doklady Akademii Nauk*, 146(2):263–266, 1962.

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2002.

[BCG+22] Aviv Bar-Natan, Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Fault-tolerant distance labeling for planar graphs. *Theoretical Computer Science*, 918:48–59, 2022.

[BCH20] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic dynamic matching in O(1) update time. *Algorithmica*, 82(4):1057–1080, 2020.

[BDH+19] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *FOCS*, pages 382–405. IEEE Computer Society, 2019.

[BDS+24a] Edouard Bonnet, Julien Duron, John Sylvester, Viktor Zamaraev, and Maksim Zhukovskii. Small but unwieldy: A lower bound on adjacency labels for small classes. In *SODA*, pages 1147–1165. SIAM, 2024.

[BDS+24b] Édouard Bonnet, Julien Duron, John Sylvester, Viktor Zamaraev, and Maksim Zhukovskii. Tight bounds on adjacency labels for monotone graph classes. In *ICALP*, volume 297 of *LIPIcs*, pages 31:1–31:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[BDSZ24] Édouard Bonnet, Julien Duron, John Sylvester, and Viktor Zamaraev. Adjacency labeling schemes for small classes. *CoRR*, abs/2409.04821, 2024.

[BEGS21] Marthe Bonamy, Louis Esperet, Carla Groenland, and Alex D. Scott. Optimal labelling schemes for adjacency, comparability, and reachability. In *53rd STOC*, pages 1109–1117. ACM, 2021.

[BF67] Melvin A Breuer and Jon Folkman. An unexpected result in coding the vertices of a graph. *Journal of Mathematical Analysis and Applications*, 20(3):583–600, 1967.

[BF98] Reuven Bar-Yehuda and Sergio Fogel. Partitioning a sequence into few monotone subsequences. *Acta Informatica*, 35(5):421–440, 1998.

[BF08] Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theor. Comput. Sci.*, 409(3):486–496, 2008.

[BFR72] K. A. Baker, Peter C. Fishburn, and Fred S. Roberts. Partial orders of dimension 2. *Networks*, 2(1):11–28, 1972.

[BG05] Fabrice Bazzaro and Cyril Gavoille. Distance labeling for permutation graphs. *Electron. Notes Discret. Math.*, 22:461–467, 2005.

[BGP20] Marthe Bonamy, Cyril Gavoille, and Michał Pilipczuk. Shorter labeling schemes for planar graphs. In *31st SODA*, pages 446–462, 2020.

[BHR19] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. Online bipartite matching with amortized $O(\log^2 n)$ replacements. *J. ACM*, 66(5):37:1–37:23, 2019.

[BK15] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th FOCS*, pages 79–97, 2015.

[BPW19] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *STOC*, pages 365–376. ACM, 2019.

[BS16] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *SODA*, pages 692–711. SIAM, 2016.

[But09] Steve Butler. Induced-universal graphs for graphs with bounded maximum degree. *Graphs Comb.*, 25(4):461–468, 2009.

[CCP13] Alex Chen, Timothy Chu, and Nathan Pinsker. Computing the longest increasing subsequence of a sequence subject to dynamic insertion. *CoRR*, abs/1309.7724, 2013.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[CGKT08] Bruno Courcelle, Cyril Gavoille, Mamadou Moustapha Kanté, and Andrew Twigg. Connectivity check in 3-connected planar graphs with obstacles. *Electron. Notes Discret. Math.*, 31:151–155, 2008.

[Che13] Shiri Chechik. Compact routing schemes with improved stretch. In *PODC*, pages 33–41. ACM, 2013.

[CHL00] H.S. Chao, F.R. Hsu, and R.C.T. Lee. An optimal algorithm for finding the minimum cardinality dominating set on permutation graphs. *Discret. Appl. Math.*, 102(3):159–173, 2000.

[Chu90] Fan R. K. Chung. Universal graphs and induced-universal graphs. *Journal of Graph Theory*, 14(4):443–454, 1990.

[CKL18] Diptarka Chakraborty, Lior Kamma, and Kasper Green Larsen. Tight cell probe bounds for succinct boolean matrix-vector multiplication. In *STOC*, pages 1297–1306. ACM, 2018.

[CKM02] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic XML trees. In *PODS*, pages 271–281. ACM, 2002.

[CKM20] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes. Dynamic string alignment. In *CPM*, volume 161 of *LIPIcs*, pages 9:1–9:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Col81] Charles J. Colbourn. On testing isomorphism of permutation graphs. *Networks*, 11(1):13–21, 1981.

[Cow01] Lenore Cowen. Compact routing with minimum stretch. *J. Algorithms*, 38(1):170–183, 2001.

[CP10] Maxime Crochemore and Ely Porat. Fast computation of a longest increasing subsequence and application. *Inf. Comput.*, 208(9):1054–1059, 2010.

[DEG+21] Vida Dujmovic, Louis Esperet, Cyril Gavoille, Gwenaël Joret, Piotr Micek, and Pat Morin. Adjacency labelling for planar graphs (and beyond). *J. ACM*, 68(8):42:1–42:33, 2021.

[DJM+20] 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.

[DKR14] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Noy Rotbart. Dynamic and multi-functional labeling schemes. In *ISAAC*, volume 8889 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2014.

[DKR15] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Noy Rotbart. A simple and optimal ancestry labeling scheme for trees. In *42nd ICALP*, pages 564–574, 2015.

[DP21] Michal Dory and Merav Parter. Fault-tolerant labeling and compact routing schemes. In *PODC*, pages 445–455. ACM, 2021.

[DS87] Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *STOC*, pages 365–372. ACM, 1987.

[DSST86] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *STOC*, pages 109–121. ACM, 1986.

[EGP03] Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schemes with low stretch factor. *J. Algorithms*, 46(2):97–114, 2003.

[EJ08] Funda Ergün and Hossein Jowhari. On distance to monotonicity and longest increasing subsequence of a data stream. In *SODA*, pages 730–736. SIAM, 2008.

[EJM20] Louis Esperet, Gwenaël Joret, and Pat Morin. Sparse universal graphs for planarity. *CoRR*, abs/2010.05779, 2020.

[Ell22] David Ellis. Intersection problems in extremal combinatorics: theorems, techniques and questions old and new. *Surveys in Combinatorics*, pages 115–173, 2022.

[EPL72] Shimon Even, Amir Pnueli, and Abraham Lempel. Permutation graphs and transitive graphs. *J. ACM*, 19(3):400–410, 1972.

[ES35] Paul Erdös and George Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

[FG01] Pierre Fraigniaud and Cyril Gavoille. Routing in trees. In *28th ICALP*, pages 757–772, 2001.

[FG02] Pierre Fraigniaud and Cyril Gavoille. A space lower bound for routing in trees. In *19th STACS*, pages 65–75, 2002.

[FGNW17] Ofer Freedman, Paweł Gawrychowski, Patrick K. Nicholson, and Oren Weimann. Optimal distance labeling schemes for trees. In *36th PODC*, pages 185–194, 2017.

[FK10a] Pierre Fraigniaud and Amos Korman. Compact ancestry labeling schemes for XML trees. In *21st SODA*, pages 458–466, 2010.

[FK10b] Pierre Fraigniaud and Amos Korman. An optimal ancestry scheme and small universal posets. In *42th STOC*, pages 611–620, 2010.

[FM71] Michael J. Fischer and Albert R. Meyer. Boolean matrix multiplication and transitive closure. In *SWAT (FOCS)*, pages 129–131. IEEE Computer Society, 1971.

[Fre75] Michael L. Fredman. On computing the length of longest increasing subsequences. *Discret. Math.*, 11(1):29–35, 1975.

[Fre76] Michael L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5(1):83–89, 1976.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic barrier with fusion. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[GG10] Anna Gál and Parikshit Gopalan. Lower bounds on streaming algorithms for approximating the length of the longest increasing subsequence. *SIAM J. Comput.*, 39(8):3463–3479, 2010.

[GIu+21] Fabrizio Grandoni, Giuseppe F. Italiano, Aleksander Łukasiewicz, Nikos Parotsidis, and Przemysław Uznański. All-pairs LCA in dags: Breaking through the $O(n^2.5)$ barrier. In *SODA*, pages 273–289. SIAM, 2021.

[GJ21a] Paweł Gawrychowski and Wojciech Janczewski. Conditional lower bounds for variants of dynamic LIS. *CoRR*, abs/2102.11797, 2021.

[GJ21b] Paweł Gawrychowski and Wojciech Janczewski. Fully dynamic approximation of lis in polylogarithmic time. In *STOC*, 2021.

[GJ22] Paweł Gawrychowski and Wojciech Janczewski. Simpler adjacency labeling for planar graphs with b-trees. In *SOSA*, pages 24–36. SIAM, 2022.

[GJ24] Paweł Gawrychowski and Wojciech Janczewski. Optimal distance labeling for permutation graphs. *CoRR*, abs/2407.12147, 2024.

[GJKK07] Parikshit Gopalan, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Estimating the sortedness of a data stream. In *SODA*, pages 318–327. SIAM, 2007.

[GJL21] Paweł Gawrychowski, Wojciech Janczewski, and Jakub Lopuszanski. Shorter labels for routing in trees. In *32th SODA*, pages 2174–2193. SIAM, 2021.

[GKo+18] Paweł Gawrychowski, Fabian Kuhn, Jakub Łopuszański, Konstantinos Panagiotou, and Pascal Su. Labeling schemes for nearest common ancestors through minor-universal trees. In *29th SODA*, pages 2604–2619, 2018.

[GKU16] Paweł Gawrychowski, Adrian Kosowski, and Przemysław Uznański. Sublinear-space distance labeling using hubs. In *30th DISC*, pages 230–242, 2016.

[GL07] Cyril Gavoille and Arnaud Labourel. Shorter implicit representation for planar graphs and bounded treewidth graphs. In *15th ESA*, volume 4698 of *Lecture Notes in Computer Science*, pages 582–593. Springer, 2007.

[GP08] Cyril Gavoille and Christophe Paul. Optimal distance labeling for interval graphs and related graph families. *SIAM J. Discret. Math.*, 22(3):1239–1258, 2008.

[GP18] Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *J. ACM*, 65(4):22:1–22:25, 2018.

[GPPR04] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, October 2004.

[Gra16] Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discret. Appl. Math.*, 212:96–103, 2016.

[GU23] Paweł Gawrychowski and Przemysław Uznański. Better distance labeling for unweighted planar graphs. *Algorithmica*, 85(6):1805–1823, 2023.

[GW20a] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Decremental SSSP in weighted digraphs: Faster and against an adaptive adversary. In *SODA*, pages 2542–2561. SIAM, 2020.

[GW20b] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In *SODA*, pages 2522–2541. SIAM, 2020.

[GW20c] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In *SODA*, pages 2562–2574. SIAM, 2020.

[GWW20] Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In *STOC*, pages 153–166. ACM, 2020.

[Har20] Nathaniel Harms. Universal communication, universal graphs, and graph labeling. In *ITCS*, volume 151 of *LIPIcs*, pages 33:1–33:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[HdLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[HH22] Hamed Hatami and Pooya Hatami. The implicit graph conjecture is false. In *FOCS*, pages 1134–1137. IEEE, 2022.

[HK01] Petr Hliněnỳ and Jan Kratochvıl. Representing graphs by disks and balls (a survey of recognition-complexity results). *Discret. Math.*, 229(1-3):101–124, 2001.

[HKNS15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30. ACM, 2015.

[HMN+20] Meng He, J. Ian Munro, Yakov Nekrich, Sebastian Wild, and Kaiyu Wu. Distance oracles for interval graphs via breadth-first rank/select in succinct trees. In *31st ISAAC*, pages 25:1–25:18, 2020.

[HW24] Meng He and Kaiyu Wu. Closing the gap: Minimum space optimal time distance labeling scheme for interval graphs. In *35th CPM*, pages 17:1–17:18, 2024.

[HWZ22] Nathaniel Harms, Sebastian Wild, and Viktor Zamaraev. Randomized communication and implicit graph representations. In *STOC*, pages 1220–1233. ACM, 2022.

[IEWM23] Taisuke Izumi, Yuval Emek, Tadashi Wadayama, and Toshimitsu Masuzawa. Deterministic fault-tolerant connectivity labeling scheme. In *PODC*, pages 190–199. ACM, 2023.

[KKKP04] Michal Katz, Nir A. Katz, Amos Korman, and David Peleg. Labeling schemes for flow and connectivity. *SIAM J. Comput.*, 34(1):23–40, 2004.

[KKP05] Michal Katz, Nir A. Katz, and David Peleg. Distance labeling schemes for well-separated graph classes. *Discret. Appl. Math.*, 145(3):384–402, 2005.

[KLM19] Daniel M. Kane, Shachar Lovett, and Shay Moran. Near-optimal linear decision trees for k-sum and related problems. *J. ACM*, 66(3):16:1–16:18, 2019.

[KNR92] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.

[KOO+18] Masashi Kiyomi, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, and Jun Tarui. Space-efficient algorithms for longest increasing subsequence. In *STACS*, volume 96 of *LIPIcs*, pages 44:1–44:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[KPR10] Amos Korman, David Peleg, and Yoav Rodeh. Constructing labeling schemes through universal matrices. *Algorithmica*, 57(4):641–652, 2010.

[KS21] Tomasz Kociumaka and Saeed Seddighin. Improved dynamic algorithms for longest increasing subsequence. In *STOC*, 2021.

[ŁOP+15] Jakub Łącki, Jakub Oćwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych. The power of dynamic distance oracles: Efficient dynamic algorithms for the steiner tree. In *STOC*, pages 11–20. ACM, 2015.

[Lue78] George S. Lueker. A data structure for orthogonal range queries. In *FOCS*, pages 28–34. IEEE Computer Society, 1978.

[LW17] Kasper Green Larsen and R. Ryan Williams. Faster online matrix-vector multiplication. In *SODA*, pages 2182–2189. SIAM, 2017.

[MM99] Terry A. McKee and F. R. McMorris. *Topics in Intersection Graph Theory*. Society for Industrial and Applied Mathematics, 1999.

[Möh85] Rolf H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In *Graphs and Order: The Role of Graphs in the Theory of Ordered Sets and Its Applications*, pages 41–101. Springer Netherlands, Dordrecht, 1985.

[Moo65] J. W. Moon. On minimal n-universal graphs. *Proceedings of the Glasgow Mathematical Association*, 7(1):32–33, 1965.

[MP80] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[MS99] Ross M. McConnell and Jeremy P. Spinrad. Modular decomposition and transitive orientation. *Discret. Math.*, 201(1-3):189–241, 1999.

[MS20a] Michael Mitzenmacher and Saeed Seddighin. Dynamic algorithms for LIS and distance to monotonicity. In *STOC*, pages 671–684. ACM, 2020.

[MS20b] Michael Mitzenmacher and Saeed Seddighin. Erdös-szekeres partitioning problem. *CoRR*, abs/2011.10870, 2020.

[Mul88] John H. Muller. Local structure in graph classes. *PhD thesis, School of Information and Computer Science*, 1988.

[NP24] Moni Naor and Eugene Pekel. Adjacency sketches in adversarial environments. In *SODA*, pages 1067–1098. SIAM, 2024.

[NS15] Timothy Naumovitz and Michael E. Saks. A polylogarithmic space deterministic streaming algorithm for approximating distance to monotonicity. In *SODA*, pages 1252–1262. SIAM, 2015.

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2-\epsilon})$-time. In *STOC*, pages 1122–1129. ACM, 2017.

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961. IEEE Computer Society, 2017.

[NV20] Ilan Newman and Nithin Varma. New algorithms and lower bounds for LIS estimation. *CoRR*, abs/2010.05805, 2020.

[Pal13] Madhumangal Pal. Intersection graphs: An introduction. *Annals of Pure and Applied Mathematics*, 4(1):43–91, 2013.

[Pel00] David Peleg. Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33(3):167–176, 2000.

[Pel05] David Peleg. Informative labeling schemes for graphs. *Theor. Comput. Sci.*, 340(3):577–593, 2005.

[PPP24] Merav Parter, Asaf Petruschka, and Seth Pettie. Connectivity labeling and routing with multiple vertex failures. In *STOC*, pages 823–834. ACM, 2024.

[Rad64] Richard Rado. Universal graphs and universal functions. *Acta Arithmetica*, 9(4):331–340, 1964.

[Ram97] Prakash Ramanan. Tight $\Omega(n \lg n)$ lower bound for finding a longest increasing subsequence. *Int. J. Comput. Math.*, 65(3-4):161–164, 1997.

[Rot16] Noy Galil Rotbart. *New Ideas on Labeling Schemes*. PhD thesis, University of Copenhagen, 2016.

[RSSS19] Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for LCS and LIS with truly improved running times. In *FOCS*, pages 1121–1145. IEEE Computer Society, 2019.

[Spi03] Jeremy P. Spinrad. *Efficient graph representations*, volume 19 of *Fields Institute monographs*. American Mathematical Society, 2003.

[SR24] Arseny M. Shur and Mikhail Rubinchik. Distance labeling for families of cycles. In *49th SOFSEM*, pages 471–484, 2024.

[SS13] Michael E. Saks and C. Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *SODA*, pages 1698–1709. SIAM, 2013.

[SS17] Michael E. Saks and C. Seshadhri. Estimating the longest increasing sequence in polylogarithmic time. *SIAM J. Comput.*, 46(2):774–823, 2017.

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[Tis07] Alexandre Tiskin. Semi-local string comparison: algorithmic techniques and applications. *CoRR*, abs/0707.3619, 2007.

[TWZ23] Konstantinos Tsakalidis, Sebastian Wild, and Viktor Zamaraev. Succinct permutation graphs. *Algorithmica*, 85(2):509–543, 2023.

[TZ01] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *13th SPAA*, pages 1–10, 2001.

[UWY21] Torsten Ueckerdt, David R. Wood, and Wendy Yi. An improved planar graph product structure theorem. *CoRR*, abs/2108.00198, 2021.

[vEB77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[Wul17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *STOC*, pages 1130–1143. ACM, 2017.