Intermediate complexity problems – new upper and lower bounds

(Problemy o pośredniej złożoności obliczeniowej - nowe ograniczenia górne i dolne)

Aleksander Łukasiewicz

Praca magisterska

Promotor: dr Przemysław Uznański

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

14 września 2020

Abstract

There is a large family of graph and geometric problems for which the best known algorithms use fast matrix multiplication and their complexity is between n^ω and n^3. We call them *intermediate complexity problems*. One of these problems is *All-Pairs LCA in DAGs*. The fastest known algorithms for this problem have running times ranging from $\mathcal{O}(n^{2.687})$ [Bender et al. SODA'01] down to $\mathcal{O}(n^{2.615})$ [Kowaluk and Lingas ICALP'05] and $\mathcal{O}(n^{2.569})$ [Czumaj et al. TCS'07]. Somewhat surprisingly, all those bounds would still be $\Omega(n^{2.5})$ even if matrix multiplication could be solved optimally (i.e., $\omega = 2$). This appears to be an inherent barrier for all the currently known approaches, which raises the natural question on whether one could break through the $\mathcal{O}(n^{2.5})$ barrier for this problem.

In this work, we answer this question affirmatively: in particular, we present an $\widetilde{\mathcal{O}}(n^{2.447})$ ($\widetilde{\mathcal{O}}(n^{7/3})$ for $\omega = 2$) algorithm for finding an LCA for all pairs of vertices in a DAG, which represents the first improvement on the running times for this problem in the last 13 years.

Moreover, we show that any improvement in the complexity of the computation of the rectangular max-min matrix product (for specific matrix dimensions) implies an improvement in the complexity of the All-Pairs LCA problem.

Istnieje duża klasa problemów grafowych i geometrycznych, dla których najlepsze znane algorytmy używają szybkiego mnożenia macierzy oraz złożoność tych algorytmów znajduje się pomiędzy n^ω i n^3. Nazywamy je *problemami o złożoności pośredniej*. Jednym z takich problemów jest *Najniższy Wspólny Przodek Dla Wszystkich Par Wierzchołków w Acyklicznym Grafie Skierowanym*. Najszybsze znane algorytmy dla tego problemu mają złożoności od $\mathcal{O}(n^{2.687})$ [Bender et al. SODA'01], poprzez $\mathcal{O}(n^{2.615})$ [Kowaluk, Lingas ICALP'05], aż do $\mathcal{O}(n^{2.569})$ [Czumaj et al. TCS'07]. Co zaskakujące, wszystkie te algorytmy działałyby w czasie $\Omega(n^{2.5})$, nawet gdyby mnożenie macierzy można byłoby wykonywać optymalnie, tzn. gdyby $\omega = 2$. Wydaje się to być immanentną barierą wszystkich obecnie znanych podejść do tego problemu, co prowadzi do naturalnego pytania, czy jej złamanie jest możliwe.

W niniejszej pracy odpowiadamy na to pytanie twierdząco - w szczególności pokazujemy algorytm w czasie $\widetilde{\mathcal{O}}(n^{2.447})$ ($\widetilde{\mathcal{O}}(n^{7/3})$ dla $\omega = 2$) znajdujący najniższego wspólnego przodka dla wszystkich par wierzchołków w acyklicznych grafie skierowanym. Stanowi to pierwsze w ciągu 13 lat polepszenie złożoności dla tego problemu. Co więcej, pokazujemy również, że jakiekolwiek polepszenie złożoności liczenia max-min mnożenia prostokątnych macierzy (dla pewnych wymiarów macierzy) implikuje polepszenie złożoności problemu Najniższego Wspólnego Przodka Dla Wszystkich Par Wierzchołków w Acyklicznym Grafie Skierowanym.

Contents

Chapter 1

Introduction

1.1 Overview

Matrix multiplication is one of the most fundamental and most thoroughly studied algebraic operations. It admits an $\mathcal{O}(n^\omega)$ algorithm (with the current bound on ω being $\omega < 2.3728639$ [32]), much faster than the brute-force $\mathcal{O}(n^3)$ solution. This $\mathcal{O}(n^\omega)$ multiplication serves as a building block for many algorithms. In particular, there is a large family of graph and geometric problems for which the best known algorithms use fast matrix multiplication and their complexity is between n^ω and n^3. We call them *intermediate complexity problems*. One of these problems is *All-Pairs LCA in DAGs*, which is the main topic of interest in this thesis.

Let $G = (V, E)$ be a directed acyclic graph (DAG), with m edges and n vertices. Let u and v be any two vertices in G: if there is a path from u to v, we say that u is an *ancestor* of v and that v is a *descendant* of u. If u is an ancestor of v and $u \neq v$, we say that u is a *proper ancestor* of v (and v is a *proper descendant* of u). A *lowest common ancestor* (LCA) of u and v is the lowest (i.e., deepest) vertex w that is an ancestor of both u and v, i.e., no proper descendant of w is an ancestor of both u and v. In the special case of a tree, the lowest common ancestor of two vertices is always defined and is unique. In a DAG G, the existence of an LCA for a pair of vertices is not even guaranteed, and a pair of vertices can have as many as $(n-2)$ LCAs, where n is the total number of vertices in G.

In this thesis, we consider the problem of computing an LCA for all pairs of vertices in a DAG, which we refer to as the *All-Pairs LCA problem*. This is a fundamental problem and has many important applications, including inheritance in object-oriented programming languages, analysis of genealogical data and modeling the behavior of complex systems in distributed computing (see, e.g., [8, 14, 38] for a list of applications and especially [8] for further references).

The All-Pairs LCA problem for DAGs has been investigated in the last two decades, and many algorithms have been presented in the literature (see, e.g., [8, 9, 15, 27, 28, 30]). The problem was first considered by Bender et al. [8, 9], who proved an $\Omega(n^\omega)$ lower bound, by giving a reduction from the transitive closure problem, and presented an algorithm that runs in

$\mathcal{O}(n^{(\omega+3)/2})$ time, where ω is the exponent of the fastest known matrix multiplication algorithm. Later on, Kowaluk and Lingas [27] improved this bound to $\mathcal{O}(n^{2+1/(4-\omega)})$ by showing that the All-Pairs LCA problem can be reduced to finding maximum witnesses for Boolean matrix multiplication and by providing an efficient solution to the latter problem. The current best bound for the All-Pairs LCA problem is $\mathcal{O}(n^{2.569})$ by Czumaj et al. [15]. To achieve this bound, they solved the problem of finding maximum witnesses for Boolean matrix multiplication in time $\mathcal{O}(n^{2+\lambda})$, where λ satisfies the equation $\omega(1, \lambda, 1) = 1 + 2\lambda$. Here $\omega(1, x, 1)$ is the exponent of the (rectangular) multiplication of an $n \times n^x$ matrix by an $n^x \times n$ matrix. The currently best known bound on $\omega(1, x, 1)$ implies a bound of $\mathcal{O}(n^{2.569})$ for the All-Pairs LCA problem.

Somewhat surprisingly, all the currently known bounds for the All-Pairs LCA problem [8, 9, 15, 27] would still be $\Omega(n^{2.5})$ even if matrix multiplication could be solved optimally (i.e., $\omega = 2$). This appears to be an inherent barrier for all the currently known approaches, which raises the natural question on whether one could break through the $\mathcal{O}(n^{2.5})$ barrier for this problem.

1.2 Our result

In this work we answer this question affirmatively by presenting a new algorithm which runs in time $\widetilde{\mathcal{O}}(n^{2.447})$. This is the first improvement on the running time for this problem in the last 13 years. To this end we introduce some novel techniques, which differ substantially from previous approaches for the same problem. In particular, we develop a new technique for covering a DAG G with a small number of chains and antichains, which might also be of independent interest. Here, a chain is just a path in G, while an antichain is an independent set (i.e., a subset of vertices such that there is no edge between any two of them). In more detail, given a parameter $\ell \leq n$, we show how to partition the vertices of G into at most ℓ chains and $\frac{2n}{\ell}$ antichains in time $\mathcal{O}(n\ell^2)$. We refer to this as an $(\ell, \frac{2n}{\ell})$-decomposition of G.

Moreover, we further speed up the time needed for computing $(\ell, \frac{2n}{\ell})$-decomposition down to $\mathcal{O}(n^2)$. This enables us to show that computation of the max-min matrix product appears to be a barrier for faster solution to the All-Pairs LCA problem, i.e we show that any time improvement to the former problem implies an improvement to the latter.

We remark that if matrix multiplication could be solved optimally (i.e., $\omega = 2$), several graph algorithms based on fast matrix multiplication would take either time $\widetilde{\mathcal{O}}(n^2)$ or time $\widetilde{\mathcal{O}}(n^{2.5})$. As it was already mentioned, the previous algorithms by Bender et al. [8, 9], by Kowaluk and Lingas [27] and by Czumaj et al. [15] would all take time $\widetilde{\mathcal{O}}(n^{2.5})$. On the other side, under the same assumption, the running time of our algorithm would be $\widetilde{\mathcal{O}}(n^{2+\frac{1}{3}})$. Thus, our improvement suggests a possible separation between the All-Pairs LCA and the minimum / maximum witness for Boolean matrix multiplication (used by Czumaj et al. [15] as a reduction in their algorithm for All-Pairs LCA).

1.3 Structure of the thesis

In Chapter 2. we introduce necessary definitions and recall useful facts about fast matrix multiplication and max-min matrix product.

In Chapter 3. we define the notion of (a, b) decomposition of the DAG, discuss a greedy algorithm for computing $(\ell, \frac{n}{\ell})$ decomposition and construct faster algorithm that computes $(\ell, \frac{2n}{\ell})$ decomposition in $\mathcal{O}(n\ell^2)$ time.

In Chapter 4. we show how to use fast algorithm from the previous chapter in order to solve All-Pairs LCA problem in $\mathcal{O}(n^{2.447})$ time. After finding the $(\ell, \frac{2n}{\ell})$ decomposition, for each pair of vertices, we compute a candidate LCA among the chain and antichain vertices, separately. We show that the former case can be solved via a reduction to max-min matrix product, similarly to Bender et al. [8, 9] and to Czumaj et al. [15]. Then, we explain that the computation in the latter case can be reduced to several Boolean matrix multiplications. Finally, we discuss how to combine the two solutions, which is non-trivial and requires some extra care.

In Chapter 5. we show how to speed up $(\ell, \frac{2n}{\ell})$ decomposition down to $\mathcal{O}(n^2)$.

In Chapter 6. we provide applications of the $\mathcal{O}(n^2)$ decomposition from the previous chapter. We define S-pairs LCA problem and construct a fast algorithm that solves it. Next, we show that an improvement of the time complexity of computing max-min matrix product implies an improvement to the solution of the All-Pairs LCA problem.

1.4 Related work

The problem of finding LCAs in trees was first introduced by Aho et al. [1]. The first optimal (linear preprocessing and $\mathcal{O}(1)$ time per query) solution to this problem was presented by Harel and Tarjan [24], altough with a sophisticated data structure which is not practical. The first simple, near-optimal algorithm for LCAs in trees was introduced by Bender and Farach-Colton [7]. We remark that LCA problem in trees exemplifies a rather different structure than in DAGs.

Matrix multiplication is a fundamental problem, with a long line of algebraic approaches, with recent results by Stothers [40], Vassilevska-Williams [43] and finally Le Gall [32], which yielded $\omega < 2.3728639$. There are known faster (under the assumption that $\omega > 2$) algorithms for rectangular matrix multiplication, with current best bounds by Le Gall and Urrutia [33]. There is a long list of problems which are equivalent to matrix multiplication e.g., Boolean matrix multiplication witnesses with $\widetilde{\mathcal{O}}(n^\omega)$ [3] and All-Pairs Shortest Paths (APSP) in undirected unweighted graphs [2].

Examples of intermediate complexity problems other than All-Pairs LCA include All-Pair Bottleneck Paths (APBP): [19, 41], vertex APBP [39], unweighted directed APSP [49], All-Pair Nondecreasing Paths [17, 18, 44], and Dominance-, Hamming- and L_1- matrix products: [25, 36, 47]. Interestingly, for all the aforementioned problems, the best known algorithms would be of complexity $\widetilde{\mathcal{O}}(n^{2.5})$ if $\omega = 2$. For fine-grained complexity of intermediate complexity

problems and the relations between them, see recent results [6, 11, 20, 35].

For other results on All-Pairs LCA, see [28] on finding unique LCA. Other work in the area include [16].

For related problem of minimal witnesses of Boolean matrix multiplication, we refer to an algorithm for sparse matrices [13], and to a recent quantum algorithm [29]. We also refer to [30] which introduced path covering technique in the All-Pairs LCA problem. The authors observe that covering a DAG with a small number of paths might lead to faster algorithms, which is one of the key observations used in our algorithm. However, this observation alone does not lead to a faster algorithm.

The decomposition of a partially ordered set into disjoint chains and antichains can be seen as a special case of finding a *cocoloring* of a graph. A *cocoloring* of a graph is a partition of its vertices into cliques and independent sets. The *cochromatic number* of a graph is the cardinality of the smallest cocoloring. This problem has been originally studied by Lesniak and Straight [34]. The special case of partitioning sequence into monotonic subsequences has received considerable attention [5, 10, 21, 22, 42, 46], since it has many applications, including book embeddings [5], and geometric algorithms [4, 5, 12].

1.5 Contributions of the author and acknowledgments

This thesis is largely based on the paper [23] by Fabrizio Grandoni, Giuseppe F. Italiano, Aleksander Łukasiewicz, Nikos Parotsidis and Przemysław Uznański. This paper has been recently submitted to a conference.

The main conceptual contribution of the author of this thesis was construction of the Algorithm 2, thus improving the time needed for computing $(\ell, \frac{2n}{\ell})$ decomposition from the naive $\mathcal{O}(n^2\ell)$ to $\mathcal{O}(n\ell^2 \log n + n^2 \log n)$. This was the key missing ingredient required to improve upon the previously know algorithms for All-Pairs LCA problem (as described in Figure 9). Later on, the author derived a series of refinements to the Algorithm 2, improving the time down to $\mathcal{O}(n\ell^2 + n^2)$, then $\mathcal{O}(n^{2+\frac{1}{3}} \log n)$ and finally to $\mathcal{O}(n^2)$. Further simplifications and refinements made with the other co-authors led to the current presentation of the $\mathcal{O}(n^2)$ algorithm (Algorithm 6 described in Chapter 5). We remark that this further improvement of the decomposition enabled us to show that max-min product exemplifies the source of hardness in the All-Pairs LCA problem (which was non-obvious while we had decomposition in time $\mathcal{O}(n\ell^2 \log n + n^2 \log n)$). We discuss this matter in the Section 6.2.

Contents of the Section 6.2 is a result of an ongoing research, made by the author and the advisor in collaboration with Adam Polak.

Chapter 2

Preliminaries

2.1 All-Pairs LCA

Let $G = (V, E)$ be a DAG, with m edges and n vertices. Without loss of generality we assume that G is weakly connected (hence, $m \geq n - 1$). If $(u, v) \in E(G)$ we say that u is a *parent* of v and v is a child of u. If there is a path from u to v in G we say that u is an *ancestor* of v and that v is a *descendant* of u. If u is an ancestor of v and $u \neq v$, we say that u is a *proper ancestor* of v (and v is a *proper descendant* of u). A *lowest common ancestor* (LCA) of u and v is the lowest (i.e., deepest) vertex w that is an ancestor of both u and v, i.e., no proper descendant of w is an ancestor of both u and v. We use $LCA(u, v)$ to denote the set of LCAs of u and v. In case there is no common ancestor of u and v, $LCA(u, v) = \emptyset$. In this thesis, we consider the following problem.

Problem 2.1.1 (All-Pairs LCA)**.** *Let $G = (V, E)$ be a DAG. Compute a lowest common ancestor for all pairs of vertices $u, v \in V$.*

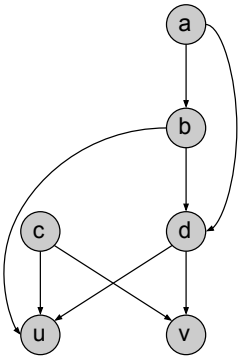

Figure 2.1: An ilustration of the definition of LCAs in DAGs - c and d are both valid LCAs for the pair u, v, while vertices a and b are not.

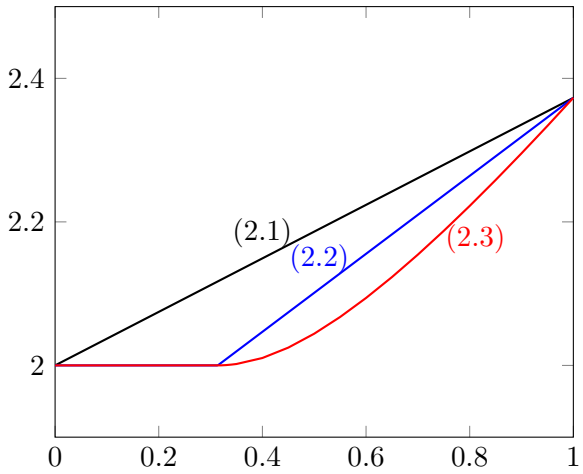

Figure 2.2: Bounds on $\omega(1, x, 1)$. We drew the whole curve of the bound 2.3 from [33].

2.2 Matrix multiplication

We use $\mathrm{MM}(X, Y, Z)$ to denote the time complexity of multiplying two matrices of dimensions $X \times Y$ and $Y \times Z$ respectively. We denote by ω the exponent of the fastest known matrix multiplication algorithm, i.e., $\mathrm{MM}(n, n, n) = \mathcal{O}(n^{\omega})$. The current best bound for ω is $\omega < 2.3728639$ [32]. We denote by $\omega(a, b, c)$ the rectangular matrix multiplication exponent, i.e., $\mathrm{MM}(n^a, n^b, n^c) = \mathcal{O}(n^{\omega(a,b,c)})$. The following is a standard bound derived from reducing rectangular matrix multiplication to square matrix multiplication:

$$\omega(1, x, 1) \leq 2 + x(\omega - 2) \qquad \text{for } 0 \leq x \leq 1. \tag{2.1}$$

We introduce the following definition:

Definition 2.2.1. *Let $\alpha > 0.31389$ be the maximum value satisfying $\omega(1, \alpha, 1) = 2$, and let $\beta = \frac{\omega - 2}{1 - \alpha}$.*

The following bound holds:

$$\omega(1, x, 1) \leq \begin{cases} 2 + \beta(x - \alpha) & \text{when } \alpha \leq x \leq 1, \\ 2 & \text{when } 0 \leq x \leq \alpha. \end{cases} \tag{2.2}$$

We remark that there are even better bounds on $\omega(1, x, 1)$ (see, e.g., [33]). In particular, the following bound is known:

$$\omega(1, x, 1) \leq 1.690383 + 0.66288 \cdot x \text{ for } 0.7 \leq x \leq 0.75 \tag{2.3}$$

For the plot of aforementioned bounds see Figure 2.2.

We will also make use of the following simple lemmas.

Lemma 2.2.2. $\omega(1, x + c, 1) \leq \omega(1, x, 1) + c$

Proof. Let A be an $n \times n^{x+c}$ matrix and B be an $n^{x+c} \times n$ matrix. Let A'_i denote the submatrix of A consisting of the columns of numbers $(i-1)n^x+1, (i-1)n^x+2, \ldots, in^x$ (so we have partitioned A into $A'_1, A'_2, \ldots, A'_{n^c}$). Analogously, let B'_i denote the submatrix of B consisting of the rows of numbers $(i-1)n^x+1, (i-1)n^x+2, \ldots, in^x$. It is now easy to see that $A \cdot B = \sum_{i=1}^{n^c} A'_i \cdot B'_i$. Each product $A'_i \cdot B'_i$ can be computed in $\mathcal{O}(n^{\omega(1,x,1)})$, so all the multiplications take $\mathcal{O}(n^{\omega(1,x,1)+c})$ time. Each product $A'_i \cdot B'_i$ is an $n \times n$ matrix, so the cost of summation is $\mathcal{O}(n^{2+c})$. Therefore, we have shown how to compute $A \cdot B$ in $\mathcal{O}(n^{\omega(1,x,1)+c} + n^{2+c})) = \mathcal{O}(n^{\omega(1,x,1)+c})$ time. □

Lemma 2.2.3. *$\omega(at, bt, ct) = t \cdot \omega(a, b, c)$ for all $a, b, c, t > 0$*

Proof. Let A, B be $n^{at} \times n^{bt}$ and $n^{bt} \times n^{ct}$ matrices, respectively. Let $m = n^t$. Then A is obviously $m^a \times m^b$ matrix and B is $m^b \times m^c$ matrix. Therefore we can compute $A \cdot B$ in $\mathcal{O}(m^{\omega(a,b,c)}) = \mathcal{O}(n^{t\omega(a,b,c)})$ time. □

2.3 Max-min matrix product

Definition 2.3.1. *Let A and B be matrices (of dimensions $n \times p$ and $p \times n$ respectively) with values from $\mathbb{Z} \cup \{-\infty\}$. The products $\circledcirc$ (dominance product) and $\textcircled{\vee}$ (max-min product) are defined as*

$$(A \circledcirc B)[i,j] = |\{k : A[i,k] < B[k,j]\}|$$
$$(A \textcircled{\vee} B)[i,j] = \max_k \min(A[i,k], B[k,j])$$

We use rectangular max-min product as one of the ingredients of our solution for All-Pairs LCA problem. Previously, Duan and Pettie have shown fast algorithms for square sparse dominance product and for square max-min product [19]. More specifically, given $n \times n$ matrices A, B with m_1 and m_2 non $(-\infty)$ values respectively, they compute $A \circledcirc B$ product in $\mathcal{O}(m_1 m_2/n + n^\omega)$ time. They later use this algorithm in order to construct an $\mathcal{O}(n^{\frac{\omega+3}{2}})$ algorithm for $A \textcircled{\vee} B$ product (where A, B are $n \times n$ matrices of arbitrary density). Below, we use simple modification of algorithms and analysis from [19], in order to obtain necessary generalizations for rectangular products. Note that we provide version with extra log factor, for the sake of simplifying and shortening the proof.

Lemma 2.3.2 (Sparse rectangular dominance product, c.f. Theorem 3.1 in [19])**.** *If A and B are respectively $n \times p$ and $p \times n$ matrices with m_1 and m_2 non $(-\infty)$ elements, then $A \circledcirc B$ can be computed in time $\tilde{\mathcal{O}}(MM(n,p,n) + m_1 m_2/p)$.*

Proof. By reductions presented in [45] (see [31] for alternative exposition), dominance product reduces to $\mathcal{O}(\log n)$ Hamming products with a reduction preserving dimensions and sparsity. We thus have to compute $\mathcal{O}(\log n)$ sparse Hamming products, with dimensions $n \times p$ and $p \times n$, and sparsity m_1 and m_2 respectively. By folklore reduction (see full version of [31] for exposition), each such product reduces to $n \times (np)$ vs $(np) \times n$ matrix product with sparsity m_1 and m_2 respectively. By techniques of [48], such product can be computed by decomposing into "dense"

matrix product with cost $MM(n,p,n)$ (packing p densest columns of first matrix, and p densest rows of second matrix), and "sparse" matrix product with total cost $m_1 m_2/p$. □

Theorem 2.3.3 (Rectangular max-min product, c.f. Theorem 3.3 in [19])**.** *If A and B are respectively $n \times p$ and $p \times n$ matrices, then the $A \otimes\!\!\!\!\bigcirc\, B$ product can be computed in time $\widetilde{\mathcal{O}}(\sqrt{MM(n,p,n) \cdot n^2 p}\,)$.*

Proof. Let L denote set of all the values in A and B of size $2np$ (w.l.o.g. all the values are distinct). We then partition L into $L_1, \ldots, L_t$, where each L_r contains at most $\lceil 2np/t \rceil$ consecutive values from L. We then construct sparse matrices $A_1, \ldots, A_t$ and $B_1, \ldots, B_t$ of dimensions $n \times p$ and $p \times n$, such that:

$$A_r[i,j] = \begin{cases} A[i,j] \text{ if } A[i,j] \in L_r \\ \infty \text{ otherwise} \end{cases}$$

$$B_r[i,j] = \begin{cases} B[i,j] \text{ if } B[i,j] \in L_r \\ -\infty \text{ otherwise} \end{cases}$$

For each A_r, a *row-balancing* operation is applied (see Definition 2.1, [19]), producing A'_r and A''_r, each of dimension $n \times p$ with $\mathcal{O}(p/t)$ elements in each row that are not ∞.

By construction from Theorem 3.3 in [19], we need to compute, for each r: $A_r \otimes\!\!\!\!\bigcirc\, B$, $A'_r \otimes\!\!\!\!\bigcirc\, B$ and $A''_r \otimes\!\!\!\!\bigcirc\, B$. Each such product reduces to: multiplication of Boolean matrices of dimension $n \times p$ with $p \times n$, and sparse dominance product of dimension $n \times p$ with $p \times n$ and density $m_1 = m_2 = \mathcal{O}(np/t)$. By Lemma 2.3.2, this takes $\widetilde{\mathcal{O}}(MM(n,p,n) + n^2 p/t^2)$ for each product. The postprocessing phase takes $\mathcal{O}(p/t)$ time for each n^2 elements of the output. The resulting time is then $\widetilde{\mathcal{O}}(MM(n,p,n)t + n^2 p/t)$, so by setting $t = \sqrt{n^2 p/MM(n,p,n)}$ the runtime bound is satisfied. □

Let also $\omega_{\otimes\!\!\!\!\bigcirc}(a,b,c)$ denote the rectangular max-min product exponent, i.e., max-min product of matrices of dimensions $a \times b$ and $b \times c$ can be computed in time $\widetilde{\mathcal{O}}(n^{\omega_{\otimes\!\!\!\!\bigcirc}(a,b,c)})$. Observe, that Theorem 2.3.3 yields immediately the following corollary.

Corollary 2.3.4. $\omega_{\otimes\!\!\!\!\bigcirc}(1,x,1) \leq \frac{\omega(1,x,1)+2+x}{2}$

Chapter 3

Fast Chain-Antichain Decomposition

3.1 Definition

Let $G = (V, E)$ be a DAG, with n vertices and m edges. A *chain* (of size k) of G is a subset of vertices $\{c_1, \ldots, c_k\}$ such that $c_1, \ldots, c_k$ is a directed path in G. An *antichain* (of size k) of G is a subset of vertices $\{a_1, \ldots, a_k\}$ such that there is no edge between them. Observe that if G is the graph induced by a partial order, then our definitions coincide with the usual definitions of chain and antichain in a partially ordered set.

Definition 3.1.1. *An (a, b)-decomposition of a DAG G consists of a collection $\mathcal{P}$ of chains of G, $|\mathcal{P}| \leq a$, and a collection $\mathcal{Q}$ of antichains of G, $|\mathcal{Q}| \leq b$, that together span all the vertices of G.*

3.2 Greedy algorithm

One could find an $(\ell, n/\ell)$-decomposition with a "simple minded" greedy method, as follows: find and remove the longest chain in G, for ℓ times in total. Next, cover whatever remains with at most n/ℓ antichains. We formalize this approach using the pseudocode below.

Lemma 3.2.1. *The Algorithm 1 computes an $(\ell, \frac{n}{\ell})$-decomposition.*

Proof. Let G' denote the graph that remains after execution of the loop that removes longest path ℓ times. Firstly, observe that, for all i, Q_i forms an antichain. Indeed, if $u \in Q_i$ and $(u, v) \in E(G')$, then the length of the longest path ending at v is at least $i + 1$, hence $v \notin Q_i$. Now observe that by construction $\mathcal{P}$ contains ℓ chains and $\mathcal{Q}$ contains $\lfloor \frac{n}{\ell} \rfloor$ antichains, so it remains to show that every vertex of G has been covered by some chain from $\mathcal{P}$ or antichain from $\mathcal{Q}$. Observe that the longest path in G' has to be shorter than $\frac{n}{\ell}$ (otherwise we would have already removed at least $\ell \frac{n}{\ell} = n$ vertices, and G' would be empty). Therefore, antichains $\{Q_1, \ldots, Q_{\lfloor \frac{n}{\ell} \rfloor}\}$ indeed cover all the vertices of G'. But vertices from $G \setminus G'$ has been already covered by chains, hence $(\mathcal{P}, \mathcal{Q})$ indeed form a decomposition. □

Algorithm 1: Naive $(\ell, \frac{n}{\ell})$-decomposition

Data: DAG G

Result: $(\ell, \frac{n}{\ell})$-decomposition of G.

$G' \leftarrow G, \mathcal{P} \leftarrow \emptyset, \mathcal{Q} \leftarrow \emptyset$

for $i = 1, \ldots, \ell$ **do**

  Find longest path P in G'

  Add P to $\mathcal{P}$

  Remove vertices of P from G'

end

Compute $Q_i = \{v \in G' :$ the longest path in G' ending at v has length $i\}$ for all $i \in \{1, \ldots, \lfloor \frac{n}{\ell} \rfloor\}$.

$\mathcal{Q} \leftarrow \{Q_1, \ldots, Q_{\lfloor \frac{n}{\ell} \rfloor}\}$

return $(\mathcal{P}, \mathcal{Q})$

Lemma 3.2.2. *The Algorithm 1 runs in time $\mathcal{O}(m\ell)$ in total.*

Proof. Computing the longest path in DAG can be done by a standard dynamic programming algorithm that goes through all vertices of the graph in topological order. While visiting vertex v, this algorithm computes the length of the longest path ending at v using already computed values for all parents of v. It is clear that this algorithm runs in time $\mathcal{O}(m)$. It is easy to see that it can be also used to compute sets $Q_1, \ldots, Q_{\lfloor \frac{n}{\ell} \rfloor}$. Therefore the total running time of Algorithm 1 is $\mathcal{O}((\ell + 1)m) = \mathcal{O}(m\ell)$. □

For dense graphs, Algorithm 1 runs in time $\mathcal{O}(n^2\ell)$, which would be too slow for our algorithm for All-Pairs LCA problem, that we will develop in section 4.

3.3 Faster algorithm for transitively-closed DAGs

We say that G is transitively-closed if for every $u, v \in G$ the fact that there is a path from u to v implies that there is an edge from u to v. Obviously, a transitive closure of a graph is transitively-closed. We also note that if G is a transitively closed DAG, then it is in fact a graph of some partial order.

In this section we use fairly simple techniques to develop an algorithm for chain-antichain decomposition in transitively-closed DAGs that runs faster than greedy Algorithm 1 (in a dense case). Since in chapter 4 we operate on transitive closure of a graphs, our algorithm will prove to be fast enough to develop faster solution for All-Pairs LCA problem.

Theorem 3.3.1. *Given transitively-closed DAG G and any $1 \leq \ell \leq n$, we can find a $\left(\ell, \frac{2n}{\ell}\right)$-decomposition of G in time $\mathcal{O}(n\ell^2 \log n + n^2 \log n)$.*

Proof. We start by computing a topological ordering of G which can be done in $\mathcal{O}(n^2)$ time.

Our algorithm will run in *phases*. In each phase we operate on the graph G' which is induced on the vertices of G that we didn't remove in the previous phases (so naturally in the first phase we let $G' = G$). At the end of each phase (except possibly the last one) we will remove some chain or antichain from the graph (and add it to our decomposition).

Let h_v denote the size of the longest chain in G' where v is the maximum element (the *height* of v). Let $v_1, v_2, \ldots, v_n$ be the topological ordering of the vertices of G. We denote $L_k^i = \{v \in \{v_1, \ldots, v_i\} \cap G' : h_v = k\}$ (we call these sets *layers*). Observe that each L_k^i forms an anti-chain - indeed, by definition if u is a parent of v, then $h_u + 1 \leq h_v$. Let us also denote p_v to be an arbitrary vertex u being a parent of v and satisfying $h_u = h_v - 1$ (if there is no such vertex we set $p_v = v$).

During each phase we iteratively compute values h_{v_i}, p_{v_i} and layers L'_k for $i = 1, 2, \ldots$ until we either run out of vertices or we decide to remove some chain or anti-chain from the graph, add it to the decomposition and start a new phase on the graph with the reduced number of vertices.

Namely, we continue computations in the phase until we meet one of the following conditions:

1. $L^i_{\frac{n}{\ell}} \neq \emptyset$, i.e we have found a chain of length $\frac{n}{\ell}$
2. there is k such that $|L_k^i| \geq \ell$, i.e we have found an anti-chain of length ℓ.
3. $i > n$, i.e we run out of vertices.

Assume we didn't run out of vertices but our computation of heights has stopped. This means that we have either found a chain of size $\frac{n}{\ell}$ or an anti-chain of size ℓ. In either case we take the corresponding chain or anti-chain to our decomposition and we remove its vertices from the graph. Removal of the vertices from the anti-chain is immediate and removal of the chain can be easily performed using p_v values (both taking constant time per vertex). We conclude the phase and proceed by running a new phase on the remaining graph.

We continue repeating the phases until one of them stops because of the condition (3). Observe that during preceding phases we could not have added more than ℓ chains and $\frac{n}{\ell}$ anti-chains to the decomposition (otherwise we would have already run out of the vertices). The vertices we have just processed in the last phase are not included in the decomposition yet. However, we can use non-empty anti-chains L_i^j from the last phase to cover them - the algorithm guarantees that there are at most $\frac{n}{\ell}$ many of them. This gives a desired $\left(\ell, \frac{2n}{\ell}\right)$-decomposition.

We give a formalized version of the algorithm we have just described in the pseudo-code of Algorithm 2.

We now analyze the computational cost of the algorithm. It is dominated by repeated computation of values h_{v_i}, p_{v_i} and construction of the layers L_k^i.

Observe that naive computation of h_{v_i} and p_{v_i} for a fixed v_i would take $\mathcal{O}(n)$ time. To speed it up, let $u_1, u_2, \ldots, u_{h_{v_i}} = v_i$ be any longest chain in G' ending at v_i. By definition all u_j's precede v_i in the topological ordering, therefore we already know their heights by the time

we want to compute h_v. It is also easy to see that $h_{u_j} = j$, hence $u_j \in L_j^{i-1}$. Also, because G is transitively-closed, $u_1, u_2, \ldots, u_{h_{v_i}-1}$ are all parents of v_i. So we can determine if $h_{v_i} > j$ for any j just by scanning through L_j^{i-1} and determining wherever there is some $u \in L_j^{i-1}$ that is a parent of v_i. If not, then surely $h_{v_i} \leq j$. This means that we can use binary search for computing h_{v_i} (and consequently p_{v_i}). Since we maintain $\forall_{i,k}|L_k^i| < \ell$ invariant throughout the phase, the complexity of one query in binary search is $\mathcal{O}(\ell)$. This yields $\mathcal{O}(\ell \log n)$ for computing h_{v_i} and p_{v_i} for a fixed v_i.

We have $L_{h_{v_i}}^i = L_{h_{v_i}}^{i-1} \cup \{v_i\}$ and $L_j^i = L_j^{i-1}$ for $j \neq h_{v_i}$, hence updating L_j^i's takes constant time per vertex.

Finally, since at each phase except possibly the last one we remove either l or $\frac{n}{l}$ vertices from the graph, we can't run more than $\max(\frac{n}{\ell}, \ell) + 1 \leq \frac{n}{\ell} + \ell + 1$ phases. Each phase takes $\mathcal{O}(n\ell \log(n))$ time. This yields $\mathcal{O}(n^2 \log n + n\ell^2 \log n + n^2) = \mathcal{O}(n\ell^2 \log n + n^2 \log n)$ total time cost.

□

Algorithm 2: Faster $(\ell, \frac{2n}{\ell})$-decomposition for transitively-closed DAGs

Data: Transitively-closed DAG G
Result: $(\ell, \frac{2n}{\ell})$-decomposition of G.

$G' \leftarrow G, \mathcal{P} \leftarrow \emptyset, \mathcal{Q} \leftarrow \emptyset$
while *True* **do**
  $L_i \leftarrow \emptyset$ for $1 \leq i \leq h$
  for $i = 1, \ldots, n$ **do**
    if $v_i \in G'$ **then**
      $j \leftarrow \max\{k : \exists_{u \in L_k}(u, v_i) \in E(G')\}$ or 0 if this maximum does not exist
      $h_{v_i} \leftarrow j + 1$
      Add v_i to L_{j+1}
      if $h_{v_i} = h$ **then**
        $u \leftarrow v_i$
        $C \leftarrow \emptyset$
        while $p_u \neq u$ **do**
          Add u to C
          $u \leftarrow p_u$
        end
        Add u to C
        Add chain C to $\mathcal{P}$ and remove all vertices of C from G'
        break
      end
      else if $|L_{h_{v_i}}| = \ell$ **then**
        Add antichain $L_{h_{v_i}}$ to $\mathcal{Q}$ and remove all vertices of $L_{h_{v_i}}$ from G'
        break
      end
    end
  end
  if *In the for loop didn't add a new chain or antichain to the decomposition* **then**
    break
  end
end
Compute $Q_i = \{v \in G' :$ the longest path in G' ending at v has length $i\}$ for all $i \in \{1, \ldots, \lfloor \frac{n}{\ell} \rfloor\}$.
$\mathcal{Q} \leftarrow \{Q_1, \ldots, Q_{\lfloor \frac{n}{\ell} \rfloor}\}$
return $(\mathcal{P}, \mathcal{Q})$

Chapter 4

Application: All-Pairs LCA in DAGs

4.1 Overview

In this chapter we present our improved algorithm for All-Pairs LCA in DAGs.

We start by sketching the high level ideas behind the algorithm. Let G_{input} be the input DAG and let G be the transitive closure of G_{input}. We compute G in $\mathcal{O}(n^\omega)$ time and solve the All-Pairs LCA problem on G (obviously the solution in the two cases is identical).

To do this, we first compute an $(n^x, 2n^{1-x})$-decomposition $(\mathcal{P}, \mathcal{Q})$ of G in $\mathcal{O}(n^{1+2x} \log n + n^2 \log n)$ time with the algorithm from Theorem 3.3.1. Recall that $\mathcal{P} = \{P_1, \ldots, P_p\}$ is a set of $p \leq n^x$ chains and $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ a set of $q \leq 2n^{1-x}$ antichains. Here $x \in [0, 1]$ is a parameter to be optimized later in order to minimize the overall running time.

We now define the notion of LCA restricted to a subset W of vertices as follows.

Definition 4.1.1. *Given a DAG $G = (V, E)$, a subset of vertices $W \subseteq V$, and a pair of vertices $u, v \in V$, $LCA_W(u, v)$ is the set of vertices $w \in W$ which are ancestors of both u and v and such that there is no descendent $w' \in W$ of w with the same property. Any $w \in LCA_W(u, v)$ is a W-restricted LCA of $\{u, v\}$. The W-restricted All-Pairs LCA problem is to compute $\mathsf{lca}_W(u, v) \in LCA_W(u, v)$ for all pairs of vertices $u, v \in V$ ($\mathsf{lca}_W(u, v) = -\infty$ if $LCA_W(u, v) = \emptyset$).*

We use $\mathcal{P}$-restricted and $\mathcal{Q}$-restricted as shortcuts for $(\cup_{P \in \mathcal{P}} P)$-restricted and $(\cup_{Q \in \mathcal{Q}} Q)$-restricted respectively, and also define analogously $\mathrm{LCA}_{\mathcal{P}}(\cdot, \cdot)$, $\mathsf{lca}_{\mathcal{P}}(\cdot, \cdot)$, etc. The next step is to solve the $\mathcal{P}$-restricted and $\mathcal{Q}$-restricted All-Pairs LCA problems. In particular, we plan to compute the values $\mathsf{lca}_{\mathcal{P}}(u, v)$ and $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for all pairs of vertices $u, v \in V$. This is explained in sections 4.2 and 4.3 respectively. In more detail, the first problem is solved in time $\widetilde{\mathcal{O}}(n^{\frac{\omega(1,x,1)+2+x}{2}})$ using a reduction to one max-min product. The second problem is solved in time $\widetilde{\mathcal{O}}(n^{1-x+\omega(1,x,1)})$ by performing one Boolean matrix product of cost $\widetilde{\mathcal{O}}(n^{\omega(1,x,1)})$ for each $Q \in \mathcal{Q}$.

At this point for every pair or vertices $\{u, v\}$ we have two candidates for $lca(u, v)$, so we need to combine the two solutions together. A naive approach might be as follows. Let us label the

vertices from 1 to n according to some arbitrary topological order. Then, for any pair of vertices u, v, we simply set $\mathsf{lca}(u,v) = \max\{\mathsf{lca}_{\mathcal{P}}(u,v), \mathsf{lca}_{\mathcal{Q}}(u,v)\}$ (in total time $\mathcal{O}(n^2)$). Unfortunately, as discussed in Section 4.4, there exist topological orderings for which this approach fails. In the same section we show how to compute a specific topological ordering in $\mathcal{O}(n^2)$ time such that the above combination indeed works. Then, it will be sufficient to optimize over the parameter x.

Throughout this chapter we assume that vertices are labeled with integers between 1 and n (according to some given order to be specified later).

4.2 Computing $\mathcal{P}$-Restricted LCAs

In this section we present our algorithm for the $\mathcal{P}$-restricted All-Pairs LCA problem. It works as follows (see also the pseudo-code of Algorithm 3). For each vertex v and each chain P_i, we compute the parent $w_i(v)$ of v in P_i with largest index ($w_i(v) = -\infty$ if there is no such ancestor). Observe, that all vertices of P_i with indices smaller than $w_i(v)$ are ancestors of $w_i(v)$ because P_i is a chain, and therefore they are also parents of v (because G is transitively-closed). Next, for each pair of vertices u, v and each P_i, we compute $w_i(u,v) = \min\{w_i(u), w_i(v)\}$. Therefore if $w_i(u,v) \neq -\infty$, then it is a common ancestor of u and v. Finally we set $\mathsf{lca}_{\mathcal{P}}(u,v) = \max_{1 \leq i \leq p}\{w_i(u,v)\}$.

Recall that, given two matrices A and B, their max-min product $C = A \otimes B$ is specified by $C[i,j] = \max_k \min\{A[i,k], B[k,j]\}$.

In order to implement the above algorithm, it is sufficient to construct an $n \times n^x$ matrix A whose rows are indexed by vertices in V and whose columns are indexes by chains P_i. The entry $A[v, P_i]$ corresponds to the value $w_i(v)$ defined above. Then it is sufficient to compute $C = A \otimes A^T$ and set $\mathsf{lca}_{\mathcal{P}}(u,v) = C[u,v]$ for all pairs $u, v \in V$.

Lemma 4.2.1. *The $\mathcal{P}$-restricted All-Pairs LCA problem can be solved in time $\widetilde{\mathcal{O}}(n^{\frac{\omega(1,x,1)+2+x}{2}})$.*

Proof. Consider the above algorithm (pseudo-code in Algorithm 3). To analyze its running time, we observe that the matrix A can be built in time $\mathcal{O}(n^2)$ by scanning the vertices $v \in V$ and the vertices w in $\mathcal{P}$. The rest of the computation takes time $\mathcal{O}(n^{\frac{\omega(1,x,1)+2+x}{2}})$ by Corollary 2.3.4.

For the correctness observe that, if P_i contains a vertex in $LCA_{\mathcal{P}}(u,v)$, then this vertex has to be $w = w_i(u,v)$. Indeed, by construction w is an ancestor of both u and v. Since $w_i(u,v) = \min\{w_i(u), w_i(v)\}$, any successor w' of w along P_i is not an ancestor of u or of v. Vice versa, any ancestor w' of w along P_i cannot be in $LCA_{\mathcal{P}}(u,v)$ due to the existence of w. Therefore the set $W := \{w_i(u,v)\}_i$ contains $LCA_{\mathcal{P}}(u,v)$. Notice also that $W = \{-\infty\}$ iff u and v do not have a common ancestor in $\mathcal{P}$, in which case $LCA_{\mathcal{P}}(u,v) = \emptyset$. Therefore we can w.l.o.g. assume that the algorithm returns some $w \in W$, $w \neq -\infty$. In particular, w is a vertex with the largest index in W according to the considered topological order. Assume by contradiction that $w \notin LCA_{\mathcal{P}}(u,v)$. This implies that there exists some other vertex $w' \in LCA_{\mathcal{P}}(u,v)$ which is a descendant of w. But vertex w' must be contained in W, which implies $w' < w$ (otherwise the

algorithm would not return w). This is a contradiction since w' is a descendant of w and at the same time has a smaller index in some topological order. □

Algorithm 3: Compute $\mathsf{lca}_{\mathcal{P}}(u, v)$ for all pairs of vertices $u, v \in V$.

Data: Transitive closure graph $G = (V, E)$, and a family of chains $\mathcal{P} = \{P_1, \ldots, P_p\}$ of G where $p \leq n^x$.

Result: $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u, v)$ for each pair of vertices $u, v \in V$.

```
Initialize lca_P(·,·) with −∞
Let A be an n × p matrix initialized with −∞'s
for (u, v) ∈ V × V do
    if (u, v) ∈ E and u ∈ ⋃_{P∈P} P then
        i ← index of chain P_i such that u ∈ P_i
        Update A[v, i] ← max(A[v, i], u)
    end
end
Compute the (max,min)-product A ⊛ A^T
for all u, v ∈ V do
    lca_P(u, v) ← (A ⊛ A^T)[u, v]
end
```

4.3 Computing $\mathcal{Q}$-Restricted LCAs

In this section we present our algorithm for the $\mathcal{Q}$-restricted All-Pairs LCA problem. For notational convenience let us rename $\mathcal{Q}$ as $\mathcal{Q}' = \{Q'_1, \ldots, Q'_{q'}\}$. Recall that $q' \leq 2n^{1-x}$. The first step in our construction is to transform $\mathcal{Q}'$ into a more convenient family of antichains $\mathcal{Q}$ as follows.

Definition 4.3.1. *Let $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ be a collection of disjoint antichains of a transitive closure graph $G = (V, E)$. $\mathcal{Q}$ is* path-respecting *if for any two vertices $x \in Q_i, y \in Q_j$ such that $(x, y) \in E$ it holds that $i < j$.*

Lemma 4.3.2 (Folklore)**.** *Given a transitive closure graph $G = (V, E)$ and a collection of q' disjoint antichains $\mathcal{Q}' = \{Q'_1, \ldots, Q'_{q'}\}$ over the vertex set $W \subseteq V$, there exists a greedy algorithm that computes a partition of W into a collection of $q \leq q'$ disjoint antichains $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ in time $\mathcal{O}(n^2)$.*

Proof. Let us initialize W' to W. The greedy algorithm proceeds in rounds. In round i we set $Q_i = \{$all vertices with indegree 0 in $G[W']\}$. Then Q_i is added to $\mathcal{Q}$, and its vertices are removed from W'. We halt when $W' = \emptyset$. It is easy to see that each Q_i is indeed an antichain. Let q be the number of antichains produced by the algorithm and let h be the height of $G[W]$, that is the size of its longest chain. We have that $h \leq q'$, since by Mirsky's theorem (c.f. [37])

size of any antichain cover of $G[W]$ is at least h. We also have $h = q$, since greedy algorithm reduces the length of longest chain in $G[W']$ by exactly one at each iteration.

The above algorithm can be easily implemented in time $\mathcal{O}(n^2)$. Indeed, it is sufficient to maintain the in-degree of the vertices and update them each time a vertex is removed. Whenever during an iteration the in-degree of some vertex v becomes 0 because of the removal of other vertices, we add v to a list of vertices to be used in the next round. □

We use Lemma 4.3.2 to transform $\mathcal{Q}'$ into a path-respecting family of $q \leq q' \leq 2n^{1-x}$ antichains $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$. It remains to solve the $\mathcal{Q}$-restricted All-Pairs LCA problem. To this end, we use a relatively simple reduction to Fast Boolean Matrix Multiplication. Let $C = A \cdot B$ be the product of an $n \times p$ Boolean (i.e., 0-1) matrix A and a $p \times n$ Boolean matrix B. The *witness matrix* W of this product is an $n \times n$ matrix where $W[i, j]$ is any index k such that $A[i, k] = B[k, j] = 1$. We conventionally set $W[i, j] = -\infty$ if no such index exists. Recall that the time needed to compute C is denoted by $\mathrm{MM}(n, p, n)$. A mild adaptation of the algorithm and analysis in [3] shows that we can compute W in roughly the same time.

Theorem 4.3.3 (Folklore, corollary of [3])**.** *The witness matrix W of the product $C = A \cdot B$ of an $n \times p$ Boolean matrix A and a $p \times n$ Boolean matrix B can be computed in time $\widetilde{\mathcal{O}}(MM(n, p, n))$ by a deterministic algorithm.*

Our algorithm works as follows (see also the pseudo-code of Algorithm 4). We initialize $\mathsf{lca}_{\mathcal{Q}}(\cdot, \cdot)$ with $-\infty$. Then we consider the antichains $Q_q, \ldots, Q_1$ in this order. For each Q_i and each pair of vertices u, v with $\mathsf{lca}_{\mathcal{Q}}(u, v) = -\infty$, we check if Q_i contains a common ancestor w of v and u, in which case we set $\mathsf{lca}_{\mathcal{Q}}(u, v) = w$. In order to perform efficiently this step we build a $n \times |Q_i|$ matrix A whose rows are indexed by vertices in V and whose columns are indexed by vertices in Q_i. We set entry $A[v, w]$ to 1 if w is an ancestor of v and to 0 otherwise[1]. We compute the product $A \cdot A^T$ and its witness matrix W. Notice that the pair u, v has a common ancestor w in Q_i iff $A \cdot A^T[u, v] \neq 0$, in which case $W[u, v]$ contains one such vertex. Thus it is sufficient to set $\mathsf{lca}_{\mathcal{Q}}(u, v) = W[u, v]$.

Lemma 4.3.4. *The $\mathcal{Q}$-restricted All-Pairs LCA problem can be solved in time $\widetilde{\mathcal{O}}(n^{1-x+\omega(1,x,1)})$.*

Proof. Consider the above algorithm (pseudo-code in Algorithm 4). Its running time is upper bounded by $\widetilde{\mathcal{O}}(\sum_{i=1}^{q} \mathrm{MM}(n, |Q_i|, n))$. Assume w.l.o.g. that $|Q_i|$ is non-increasing, then $|Q_i| \leq n/i$, and by monotonicity of $\mathrm{MM}(n, \cdot, n)$

$$\begin{aligned}
\sum_{i=1}^{q} \mathrm{MM}(n, |Q_i|, n) &\leq \sum_{i=1}^{q} \mathrm{MM}(n, n/i, n) \\
&\leq \sum_{j=0}^{\log q} 2^j \mathrm{MM}(n, n/2^j, n) \\
&\leq (1 + \log q) \cdot q \cdot \mathrm{MM}(n, n/q, n) \\
&\in \widetilde{\mathcal{O}}(n^{1-x+\omega(1,x,1)}).
\end{aligned}$$

[1]Padding with zeros the columns not corresponding to vertices in Q_i.

Algorithm 4: Compute $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for all pairs of vertices $u, v \in V$.

Data: Transitive closure graph $G = (V, E)$, and a family of antichains $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ of G that is path-respecting such that $q \leq 2n^{1-x}$.

Result: $\mathcal{Q}$-restricted LCA $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for each pair of vertices $u, v \in V$.

```
1  Initialize lca_Q(·,·) with −∞.
2  for i = q, ..., 1 do
3  |  Initialize a n × |Q_i| matrix A with zeros.
4  |  Let φ_i : Q_i --1:1--> {1, ..., |Q_i|} be an arbitrary bijection and φ_i^{-1}(·) be its inverse
   |   function.
5  |  for all x ∈ V, y ∈ Q_i such that (y, x) ∈ E do
6  |  |  A[x, φ_i(y)] ← 1
7  |  end
8  |  Compute A · A^T, and its witness matrix W
9  |  for all u, v ∈ V do
10 |  |  if lca_Q(u, v) = −∞ and A · A^T[u, v] ≠ 0 then
11 |  |  |  lca_Q(u, v) ← φ_i^{-1}(W[u, v]).
12 |  |  end
13 |  end
14 end
```

For the correctness, assume by contradiction that for some pair of vertices u, v the computed value $\mathsf{lca}_{\mathcal{Q}}(u, v)$ is not correct. Notice that $\mathsf{lca}_{\mathcal{Q}}(u, v) = -\infty$ iff u and v have no common ancestor in $\mathcal{Q}$, hence we can assume w.l.o.g. $\mathsf{lca}_{\mathcal{Q}}(u, v) = w$ for some w in some Q_i. The contradiction implies that there exists a common ancestor $w' \in Q_j$ of u and v which is a descendant of w (in particular, $(w, w') \in E$ since G is a transitively-closed). Notice that $j \neq i$ since Q_i is an anti-chain. By construction u and v do not have any common ancestor in $Q_{i+1}, \ldots, Q_q$ since otherwise at the time when Q_i is considered we would have $\mathsf{lca}_{\mathcal{Q}}(u, v) \neq -\infty$. Hence it must be the case that $j < i$. This is a contradiction since the existence of the pair w, w' shows that $\mathcal{Q}$ is not path-respecting. □

4.4 Patching the LCAs Together

Suppose we are given values $\mathsf{lca}_{\mathcal{P}}(\cdot, \cdot)$ and $\mathsf{lca}_{\mathcal{Q}}(\cdot, \cdot)$ as computed in previous sections. Let us also assume that vertices are labeled from 1 to n according to an *arbitrary* topological ordering. The following approach to solve All-Pairs LCA might be tempting: for each pair $u, v \in V$, we simply set $\mathsf{lca}(u, v) = \max\{\mathsf{lca}_{\mathcal{P}}(u, v), \mathsf{lca}_{\mathcal{Q}}(u, v)\}$. Unfortunately this approach does not work, as illustrated in Figure 4.1. Intuitively, the issue is that in the computation of $\mathsf{lca}_{\mathcal{Q}}(u, v)$ the algorithm can return any vertex w in some Q_i which is a common ancestor of u and v, not necessarily the one with largest index in Q_i. This flexibility is essential to achieve the claimed running time: computing w with the largest index in Q_i would require a *max-witness*

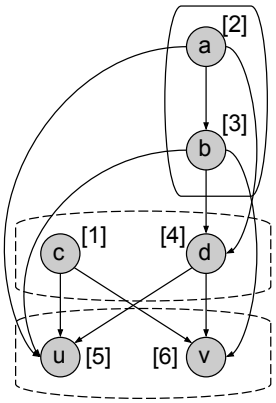

Figure 4.1: An example of a topological order in a transitive closure graph G that is not suitable for combining the $\mathcal{Q}$-restricted LCA and the $\mathcal{P}$-restricted LCA for the pair of vertices u, v. The family of chains in G is $\mathcal{P} = \{\{a, b\}\}$ and the path-respecting family of antichains is $\mathcal{Q} = \{\{c, d\}, \{u, v\}\}$. The index of each vertex in the topological order is in brackets next to each vertex. It might happen that $\mathsf{lca}_{\mathcal{P}}(u, v) = b$ and $\mathsf{lca}_{\mathcal{Q}}(u, v) = c$, in which case the algorithm would return the incorrect answer $\mathsf{lca}(u, v) = b$.

computation, and the best-known algorithms for the latter problem are substantially slower than Boolean matrix multiplication.

In order to circumvent this problem, we will compute (in $\mathcal{O}(n^2)$ time) a more structured topological order. Using this particular order rather than an arbitrary topological order, will guarantee that the above approach works. In particular, our goal is to define a topological order such that, if $\mathsf{lca}_{\mathcal{P}}(u, v)$ appears later than $\mathsf{lca}_{\mathcal{Q}}(u, v)$ in this order, then there is no path from $\mathsf{lca}_{\mathcal{P}}(u, v)$ to any $\mathcal{Q}$-restricted LCA for u, v, and vice versa.

Definition 4.4.1. *Let $G = (V, E)$ be a transitive closure graph and $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ a path-respecting family of antichains of G. A $\mathcal{Q}$-compact topological order of the vertices in V is a topological order such that all vertices in an antichain $Q_i \in \mathcal{Q}$ appear consecutively and a vertex in Q_i appears earlier than a vertex in Q_j, for $i < j$.*

Lemma 4.4.2. *Given a transitive closure graph $G = (V, E)$ and a path-respecting family of antichains $\mathcal{Q}$, we can compute a $\mathcal{Q}$-compact topological order of G in time $\mathcal{O}(n^2)$.*

Proof. For notational convenience, let us define a dummy set $Q_0 = \emptyset$. The algorithm proceeds in rounds. At the beginning of round $i \geq 0$ we are given a current subset of vertices W and a partial topological ordering R (implemented as list) of the remaining vertices $V \setminus W$. Initially $W = V$ and R is empty. During round i we append the vertices of Q_i to R in any order and remove them from W. Then we iteratively identify the sources S_i in $G[W \setminus \left(\bigcup_{i<j\leq q} Q_j\right)]$, append the vertices S_i to R in any order, and finally remove them from W.

In order to implement the above algorithm in $\mathcal{O}(n^2)$ time, we can use an approach similar to the proof of Lemma 4.3.2, where we keep track of vertices whose in-degree becomes zero because of the removal of other vertices.

For the correctness, trivially by construction the indexes of the vertices in the same antichain Q_i are consecutive, and the indexes in Q_i are smaller than the indexes in Q_j for $j > i$. Hence it remains to show that R defines a topological order at the end of the algorithm. Suppose by contradiction that there exists an edge $(x, y) \in E$ such that x is placed after y in R. Let $y \in S_i \cup Q_i$ for some i. Assume by contradiction that $y \in S_i$. Then it must be the case that $x \in Q_i$ or x was removed in some earlier iteration. Indeed otherwise y would not be a source. In both cases x would appear earlier than y in R. It therefore remains to consider the case $y \in Q_i$, $i \geq 1$. Assume that $x \in Q_j$ for some j. The fact that $\mathcal{Q}$ is path-respecting implies that $j < i$. This means that x is added to R in some earlier iteration, a contradiction. So the remaining case is that $x \in S_j$ for some $j \geq i$. Notice that vertices in S_j become sources right after the removal of vertices in Q_j (otherwise they would be sources at some earlier round). In particular, there must exist some parent w of y in Q_j. Notice that $w \in Q_j$ is an ancestor of $y \in Q_i$ (hence $(w, y) \in E$), and $j \geq i$. This contradicts the fact that $\mathcal{Q}$ is path-respecting. □

This concludes the description of our algorithm for the All-Pairs LCA problem in DAGs (see also the pseudo-code of Algorithm 5).

Algorithm 5: Compute $\mathsf{lca}_{\mathcal{V}}(u, v)$ for all pairs of vertices $u, v \in V$.

Data: DAG $G_{input} = (V, E_{input})$
Result: $\mathsf{lca}(u, v)$ for each pair of vertices $u, v \in V$

1 Compute the transitive closure graph $G = (V, E)$ of G_{input}
2 Use Algorithm 2 to compute a $(n^x, 2n^{1-x})$-decomposition into a family of chains $\mathcal{P} = \{P_1, \ldots, P_p\}$ with $p \leq n^x$ and a family of antichains $\mathcal{Q}' = \{Q'_1, \ldots, Q'_{q'}\}$ with $q' \leq 2n^{1-x}$.
3 Use Lemma 4.3.2 with input $\mathcal{Q}'$ to compute a path-respecting family of antichains $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ of G where $q \leq 2n^{1-x}$.
4 Compute a $\mathcal{Q}$-compact topological order of G using Lemma 4.4.2 and rename vertices so that they are $1, \ldots, n$ according to this order
5 Use Algorithm 3 to compute $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u, v)$ for each pair of vertices $u, v \in V$.
6 Use Algorithm 4 to compute $\mathcal{Q}$-restricted LCA $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for each pair of vertices $u, v \in V$
7 **for** *all* $u, v \in V, u \neq v$ **do**
8 $\quad \mathsf{lca}(u, v) \leftarrow \max\{\mathsf{lca}_{\mathcal{Q}}(u, v), \mathsf{lca}_{\mathcal{P}}(u, v)\}$
9 **end**

Theorem 4.4.3 (Main Theorem)**.** *All-Pairs LCA in DAGs can be solved in time $\widetilde{\mathcal{O}}(n^\gamma)$, where $\gamma = 1 + 2x$ and x is the solution of the equation $3x = \omega(1, x, 1)$.*

Proof. Consider the above All-Pairs LCA algorithm (for pseudo-code see Algorithm 5). We begin with time complexity analysis. The running time of the algorithm is $\widetilde{\mathcal{O}}(n^\omega + n^2 + n^{1+2x} + n^{\frac{\omega(1,x,1)+2+x}{2}} + n^{1-x+\omega(1,x,1)})$ for a fixed $x \in [0, 1]$. Firstly, observe that we can drop n^2, since it is dominated by n^ω. In fact we can also drop n^ω. Indeed, Lemma 2.2.2 implies $\omega = \omega(1, 1, 1) \leq$

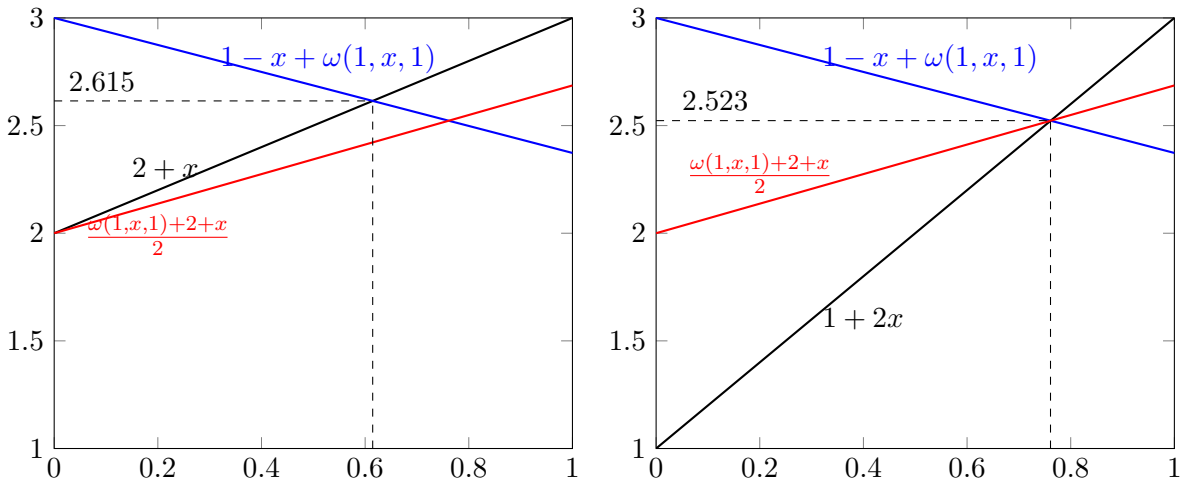

Figure 4.2: Naive decomposition (Algorithm 1) runs in time $\mathcal{O}(n^{2+x})$ and is not fast enough to make an improvement over $\mathcal{O}(n^{2.569})$ time of [15], as shown on the left. We speed up the solution to All-Pairs LCA problem only after applying better decomposition in time $\mathcal{O}(n^{1+2x})$ (Algorithm 2). For clarity of the pictures we used simple bound 2.2 when approximating $\omega(1, x, 1)$.

$\omega(1, x, 1) + 1 - x$ for $x \in [0, 1]$, hence $n^\omega \leq n^{1-x+\omega(1,x,1)}$. Now observe that $n^{\frac{\omega(1,x,1)+2+x}{2}}$ is a geometric mean of n^{1+2x} and $n^{1-x+\omega(1,x,1)}$, so in particular $n^{\frac{\omega(1,x,1)+2+x}{2}} \leq n^{1+2x} + n^{1-x+\omega(1,x,1)}$. Therefore, the running time of our algorithm becomes $\widetilde{\mathcal{O}}(n^{1+2x} + n^{1-x+\omega(1,x,1)})$. Finally, the claimed running time is obtained by imposing $1 + 2x = 1 - x + \omega(1, x, 1)$.

For the correctness assume by contradiction that $w = \mathsf{lca}(u, v)$ is not a correct answer. Notice that if u and v have no common ancestor, by construction $w = -\infty$ and the answer is correct. So we can assume that w is an index of some vertex. Assume first $w = \mathsf{lca}_{\mathcal{P}}(u, v)$. By contradiction assume that w' is some descendant of w which is also a common ancestor of u and v. Notice that $w' > w$ since we consider a topological order. The correctness of the $\mathcal{P}$-restricted All-Pairs LCA algorithm implies that w' is contained in $\mathcal{Q}$. In particular $w' \in Q_i$ for some i. Since the considered topological order is $\mathcal{Q}$-compact, all vertices in Q_i appear after w in the topological order (in particular, they have larger indices than w). Since Q_i contains at least one common ancestor of u and v (namely, w'), by construction $\mathsf{lca}_{\mathcal{Q}}(u, v)$ is contained in Q_j for some $j \geq i$. Since the topological order is $\mathcal{Q}$-compact, this implies $\mathsf{lca}_{\mathcal{Q}}(u, v) > w$. Hence we get a contradiction $w = \mathsf{lca}_{\mathcal{P}}(u, v) \geq \mathsf{lca}_{\mathcal{Q}}(u, v) > w$.

The case that $w = \mathsf{lca}_{\mathcal{Q}}(u, v)$ is symmetric. In particular, any descendant w' of w which is a common ancestor of u and v must be contained in $\mathcal{P}$, and $w' > w$. But by construction $\mathsf{lca}_{\mathcal{P}}(u, v) \geq w'$. Hence we get a contradiction $w = \mathsf{lca}_{\mathcal{Q}}(u, v) \geq \mathsf{lca}_{\mathcal{P}}(u, v) \geq w' > w$. □

We now use bounds on $\omega(1, x, 1)$ from Section 2.2 to derive numerical bounds on γ. If we simply use square matrix multiplication as a subroutine to implement rectangular matrix multiplication (i.e., the bound (2.1)), combining this with equation $3x = \omega(1, x, 1)$, we obtain $x \leq \frac{2}{5-\omega}$ and $\gamma \leq \frac{9-\omega}{5-\omega}$. In this case, the running time of our algorithm would be $\widetilde{\mathcal{O}}(n^{\frac{9-\omega}{5-\omega}}) \in \mathcal{O}(n^{2.522571})$, which is already an improvement over the algorithm by Czumaj et al. [15].

This bound can be further improved by using more sophisticated rectangular matrix multiplication algorithms. In particular, using the bound in (2.2), we get $x \leq \frac{2-\omega\alpha}{5-\omega-3\alpha}$ and $\gamma = \frac{2(2-\omega\alpha)}{5-3\alpha-\omega} + 1$, so the running time of our algorithm becomes $\mathcal{O}(n^{2.489418})$, which means breaking through the $\mathcal{O}(n^{2.5})$ barrier. By applying the bound (2.3), we finally get $x \leq 0.7232761$, and $\gamma \leq 2.4465522$, which yields our claimed running time of $\mathcal{O}(n^{2.4465522})$.

Chapter 5

Faster Chain-Antichain Decomposition

5.1 Improved algorithm

In this chapter we improve upon Algorithm 2 and present a faster algorithm for computing an $(\ell, \frac{2n}{\ell})$-decomposition of a DAG G, as stated in the following theorem.

Theorem 5.1.1. *Let $G = (V, E)$ be a DAG with n vertices, and let $\ell \in [1, n]$ be an integer parameter. There exists an $\mathcal{O}(n^2)$ time deterministic algorithm to compute an $(\ell, \frac{2n}{\ell})$-decomposition of G.*

In the case of dense graphs we clearly cannot go below $\mathcal{O}(n^2)$, as this is the time required to read the input. We discuss the case of sparse graphs later in Section 5.2.

We assume that G is represented via an adjacency matrix (otherwise, we can construct it in $\mathcal{O}(n^2)$ time). The high level idea is as follows. Let $v_1, \ldots, v_n$ be a topological ordering of the vertices of G (which can be computed in $\mathcal{O}(n^2)$ time). Let $V_i = \{v_1, \ldots, v_i\}$, $1 \leq i \leq n$, denote the first i vertices in the topological order. The algorithm consists of $(n + 1)$ iterations. At the beginning of iteration $t \geq 1$ we are given an input graph $G_{t-1} = G[W_{t-1}]$ induced in G by a set of vertices $W_{t-1} \subseteq V_{t-1}$. Initially G_0 is the empty graph (and $W_0 = \emptyset$). For $1 \leq t \leq n$, graph G_t is obtained from G_{t-1} as follows. We first add vertex v_t. Then we remove (and add to our decomposition) possibly one chain of size at least n/ℓ and possibly some antichains of size ℓ each. After iteration n, there is a final special iteration $n + 1$ where G_n is decomposed into at most n/ℓ antichains which are added to our decomposition. Clearly, this process produces at most ℓ chains and at most $2n/\ell$ antichains, as required.

As mentioned earlier, during a given iteration we insert and remove sets of vertices in a form of chains and antichains. We let $G' = G[W']$ denote the current graph. The set of vertices that are present in G' is implicitly maintained by using a Boolean vector that indicates the existence of a vertex in W'. Since each vertex is added and removed from G' at most once, the maintenance of this vector takes total time $\mathcal{O}(n)$.

Furthermore, we let $L'_1, \ldots, L'_h$, $h := \lceil n/\ell \rceil$, be disjoint (initially empty) sets of vertices that we will call *layers*. We say that a vertex $v \in L'_i$ is at *level* i. Intuitively, the level of vertex $v \in G'$ will be the length of the longest chain ending at v. This will immediately imply that all layers will form antichains. During the execution of our algorithm it holds that $|L'_i| \leq \ell$, $1 \leq i \leq h$, at all times: as a consequence, whenever at any point during the execution of the algorithm, we identify $|L'_i| = \ell$ for some set L'_i, we can remove from W' the vertices in L'_i since they form an antichain of size ℓ. We say that the algorithm is in a *stable state* when the set W' is partitioned into the sets $L'_i, 1 \leq i \leq h$, such that each vertex $v \in L'_i$, for $i \geq 2$, has a parent $u \in L'_{i-1}$. Therefore, once we have that $L'_h \neq \emptyset$ during a stable state of the algorithm, it can be seen (as we will show later) that starting from a vertex $v \in L'_h$ and following any path by traversing a parent of each visited vertex produces (the reverse of) a chain of G' of length exactly h. After the removal of some set of vertices (either a chain or antichain) from G', the algorithm might enter into an *unstable state* (i.e., not a stable state), and hence our algorithm will work to restore a stable state by suitably modifying the partitioning of W' into the sets $L'_1, \ldots, L'_h$. We next give the low level details of our algorithm.

We maintain all sets L'_i in an array of size h of doubly-linked lists. We will guarantee that each $v \in G'$ is contained in precisely one such set L'_i, and maintain bi-directional pointers between the corresponding two copies of v. We also maintain the sizes $|L'_i|$, and maintain the following quantities for each vertex $v \in G'$:

- the level $h(v)$ of v;
- a list $L_{\text{next}}(v)$ of pointers to parents of v in $L_{h(v)-1}$ ($L_{\text{next}}(v) = \emptyset$ for $h(v) = 1$).
- a list $L_{\text{prev}}(v)$ of pointers to children of v in $L_{h(v)+1}$.

The lists L_{next} and L_{prev} are used to assist fast insertions (resp., deletions) of vertices to (resp., from) a list L'_i. We note that the lists L_{prev} are not required for the correctness of the algorithm, but only for efficiency reasons. In order to be able to quickly update lists $L_{\text{prev}}(v)$ and $L_{\text{next}}(v)$ after we delete or move a vertex we store together with each entry of $w \in L_{\text{prev}}(v)$ a pointer to the occurrence of v in the list $L_{\text{next}}(w)$. We also store with each entry of $v \in L_{\text{next}}(w)$ a pointer to the occurrence of w in the list $L_{\text{next}}(v)$. This way, for some vertex $w \in L_{\text{prev}}(v)$ we can remove v from $L_{\text{next}}(w)$ in constant time, and vice versa. For the sake of simplifying the presentation, these pointers are updated implicitly and we assume that we can execute the relevant insertions and removals in constant time.

During each iteration $t \leq n$ we perform three main operations:

- $\mathsf{insert}(v)$: adds vertex v to G'. This is applied once to v_t at the beginning of iteration t.
- $\mathsf{delete}(v)$: deletes vertex v from G'. This is used to remove chains and antichains from G'.
- $\mathsf{move}(v)$: moves v from some L'_i to some L'_j, $j < i$. This is used to modify the assignment of the vertices of G' to the layers $L'_1, \ldots, L'_h$ in order to restore a stable state of the algorithm.

The latter two operations can be performed multiple times in each iteration. Throughout, we will maintain the following invariant (we prove it in Lemma 5.1.3):

Invariant 5.1.2. *After each execution of* insert(), delete() *or* move(), *the following holds:*

1. *Each vertex $v \in G'$ belongs to one L'_i and, right before an* insert() *(or after the last iteration), $L'_h = \emptyset$;*
2. *Each L'_i has size at most ℓ, and size at most $\ell - 1$ right before an* insert() *or* move().
3. *For each $v \in G'$, each parent $w \in G'$ of v belongs to some lower layer L'_j, $j < h(v)$.*
4. *There is no edge (u, v) for u and v belonging to the same layer L'_i.*
5. *Right before an* insert() *each vertex $v \in L'_i$, for $i \geq 2$, has a parent $u \in L'_{i-1}$ (i.e., the algorithm is at a stable state).*

We next describe in more detail a given iteration $t \leq n$, modulo a detailed description of the operations insert(), delete() and move() which will be given later. We create two empty lists DEL and MOVE. Intuitively, DEL contains vertices that have to be deleted from G', while MOVE contains vertices that need to be moved to a lower layer (unless they are deleted earlier) in order to restore a stable state of the algorithm. Initially we execute insert(v_t). This way we add v_t to G' to some L'_i. Then we add some vertices to DEL if one of the following two cases happens: (a) $v_t \in L'_h$ or (b) $v_t \in L'_i$ and $|L'_i| = \ell$. In case (a) we compute a set C_t iteratively as follows. Initially $u = v_t$. We add u to C_t, then update u to any vertex in $L_{\text{next}}(u)$ and iterate. We halt when $L_{\text{next}}(u) = \emptyset$. C_t is added to the set $\mathcal{P}$ of chains in the decomposition under construction and its vertices are added to DEL. Notice that by Invariant 5.1.2 the algorithm is at a stable state right before the insert() operations is executed. We will later show that, indeed, C_t is a chain in G' of size precisely h. In case (b) we add L'_i (interpreted as a set of vertices) to the set of antichains $\mathcal{Q}$ in the decomposition and its vertices to DEL. By Invariant 5.1.2, there is no edge between any two vertices in L'_i, and thus, is an antichain in G' of size precisely ℓ.

Now we perform the following steps while DEL $\cup$ MOVE $\neq \emptyset$[1]. If DEL $\neq \emptyset$, we extract v from DEL and call delete(v). This procedure removes v from G' and from the corresponding layer L'_i, and it might add some vertices to MOVE. In particular, if v used to be the only parent of w, then w is added to MOVE. Notice that at that point the algorithm is in an unstable state and cannot return to a stable state before all vertices in MOVE are re-assigned to appropriate layers.

Otherwise (i.e. DEL $= \emptyset$), we extract v from MOVE and, if $v \in G'$ (i.e., v was not deleted in some previous step), we call move(v). This procedure will move v from its current layer L'_i to some lower layer L'_j, $j < i$. If after this step it happens that $|L'_j| = \ell$, then L'_j is added to $\mathcal{Q}$ and its vertices are added to DEL. Again, by Invariant 5.1.2, there is no edge between any two vertices in L'_j, and thus, it is an antichain in G' of size ℓ.

Procedure insert(v) works as follows. We consider the layers $j = h - 1, \ldots, 1$ in this order, and check whether v has some parent in L'_j. Notice that all parents of v must have been inserted at some previous iteration, however they might not belong to G' any longer due to deletions. As soon as one such parent is found, v is added to L'_{j+1} (and $|L'_{j+1}|$ is incremented). We initialize $L_{\text{next}}(v)$ with the parents of v in L'_j and add v to $L_{\text{prev}}(u)$ for each $u \in L_{\text{next}}(v)$. We also set

[1]Notice that if cases (a) and (b) above do not happen, we stop at this point.

$L_{\text{prev}}(v) = \emptyset$ (the children of v still need to be inserted). If no parent is found, v is inserted in L'_1 (and $|L'_1|$ is incremented) and we set $L_{\text{next}}(v) = L_{\text{prev}}(v) = \emptyset$. In any case v is added to G'.

Procedure $\mathsf{delete}(v)$ works as follows. Assume $v \in L'_i$. The first step is to remove v from G' and L'_i (decrementing $|L'_i|$). Then, for each vertex $u \in L_{\text{next}}(v)$ (i.e. a parent of v) that is still in G', we remove v from $L_{\text{prev}}(u)$. Next we scan the list $L_{\text{prev}}(v)$ and for each vertex $w \in G'$ in such list we remove v from $L_{\text{next}}(w)$ and w from $L_{\text{prev}}(v)$. If $L_{\text{next}}(w) = \emptyset$ after the removal of v, we add w to MOVE.

It remains to describe $\mathsf{move}(v)$. Again assume $v \in L'_i$. Notice that by construction $i \geq 2$ since we never add to MOVE vertices in L'_1. We initially consider the vertices $w \in L_{\text{prev}}(v)$, and remove v from $L_{\text{next}}(w)$ and w from $L_{\text{prev}}(v)$. Notice that, similarly to the $\mathsf{delete}(v)$ case, if $L_{\text{next}}(w) = \emptyset$, we need to add w to MOVE. Then we consider the layers $j = i-2, \ldots, 1$ one by one, and check whether L'_j contains at least one parent of v. If such a parent is found, v is moved from L'_i to L'_{j+1} (updating $|L'_i|$ and $|L'_{j+1}|$, and setting $h(v) \leftarrow j+1$, consequently). All the parents of v in L'_j are added to $L_{\text{next}}(v)$. If no parent is found, v is moved to L'_1 and the procedure sets $L_{\text{next}}(v) = \emptyset$, and $h(v) \leftarrow 1$. In either case (that is, either $L_{\text{next}}(v) = \emptyset$ or $L_{\text{next}}(v) \neq \emptyset$), we scan all vertices of $L'_{h(v)+1}$ for children of v and for each such child u we add u to $L_{\text{prev}}(v)$ and v to $L_{\text{next}}(u)$. Notice that the above procedure moves a vertex only to a strictly lower level.

For the pseudo-code of operations $\mathsf{insert}(v), \mathsf{delete}(v), \mathsf{move}(v)$ see Algorithms 8, 9, and 7, respectively.

Observe that, by giving priority to the $\mathsf{delete}()$ operations over the $\mathsf{move}()$ operations, we avoid increasing the size of any layer above ℓ (that is, we preserve case 4 of Invariant 5.1.2). This not only allows us to identify antichains of length ℓ during an unstable state but also, most importantly, limits the size of the vertices to test for identifying parents and children during the subsequent $\mathsf{insert}()$ and $\mathsf{move}()$ operations.

At the end of the last iteration by Invariant 5.1.2 all vertices still in G' are contained in some L'_i, $i = 1, \ldots, h-1$. We execute a special final iteration $n+1$ where we add each such set L'_i as an antichain to our decomposition. This concludes the algorithm.

Lemma 5.1.3. *After each execution of $\mathsf{insert}()$, $\mathsf{delete}()$ or $\mathsf{move}()$, Invariant 5.1.2 is satisfied.*

Proof. We prove the claim by induction on the number of operations. In particular we will assume that the considered operation is the k-th one, and the invariant holds before its execution. Notice that the invariant is trivially satisfied before the first execution of any such operation (when G' and the layers L'_i are empty).

(1) Consider the first part of the claim. Clearly if the k-th operation is $\mathsf{delete}(v)$ or $\mathsf{move}(v)$, the claim holds. In case of $\mathsf{insert}(v)$, the inductive hypothesis guarantees that all parents of v in G' are in level $h-1$ or lower. Hence v is inserted in layer h or lower. For the second part of the claim, assume inductively that L'_h is empty before the execution of some $\mathsf{insert}()$ (this is true at the beginning). Observe that the only operation that can add some vertex w to L'_h is $\mathsf{insert}(w)$. Vertex w is deleted right after the $\mathsf{insert}(w)$ operation, since we always test whether $L'_h \neq \emptyset$

Algorithm 6: Compute chain/antichain decomposition

Data: DAG G with the topological ordering $v_1, \ldots, v_n$ of its vertices and parameter ℓ, with $h = \lceil n/\ell \rceil$.

Result: $(\ell, \frac{2n}{\ell})$-decomposition of G.

```
Initialize G' ← ∅, P ← ∅, Q ← ∅, L'_i ← ∅ for 1 ≤ i ≤ h
for t = 1, ..., n do
    MOVE ← ∅, DEL ← ∅
    insert(v_t)
    if |L'_{h(v_t)}| ≥ ℓ then
        Add antichain L'_{h(v_t)} to Q and all its vertices to DEL
    else if h(v_t) = h then
        u ← v_t
        C_t ← {u}
        while L_next(u) ≠ ∅ do
            u ← any element of L_next(u)
            Add u to C_t
        Add chain C_t to P and all its vertices to DEL

    while DEL ∪ MOVE ≠ ∅ do
        if DEL ≠ ∅ then
            Extract w from DEL and execute delete(w)
        else
            Extract w from MOVE and execute move(w)
            if |L'_{h(w)}| ≥ ℓ then
                Add antichain L'_{h(w)} to Q and all its vertices to DEL
return (P, Q)
```

and, if so, the algorithm retrieves and removes (the reverse of) a path that is traversed starting from w and following a parent of each visited vertex (this path always includes w). Furthermore delete() and move() never add vertices to L'_h. Hence before the execution of the next insert() the set L'_h is empty as required.

(2) Clearly if the k-th operation is delete(v) the claim holds. If the k-th operation is move(v) or insert(v) by inductive hypothesis at most one layer L'_i can reach size ℓ, while all other layers have the same or smaller size after the operation. Notice that we give priority to the delete() operations over the move() operations, and hence once $|L'_i| = \ell$ and all vertices of L'_i are inserted into DEL, no further points are inserted into L'_i until it is fully empty. Thus, all vertices of L'_i are deleted before the next execution of a move() or insert(). The claim then holds.

(3) This is the most delicate claim. The claim trivially holds if the k-th operation is delete(v),

and it holds by construction if it is insert(v). Next assume that the k-th operation is move(v), and let $v \in L'_i$ at the time of its execution. By inductive hypothesis all parents of v are of level at most $i-1$ at that time, and the procedure considers all the parents of v in G' of level at most $i-2$. Hence assume by contradiction that v has some parent in G' of level $i-1$ when move(v) is executed (in which case the invariant is violated). Suppose that the operation that added v to MOVE is the k'-th one, $k' < k$. Observe that this operation is either a move(w) or a delete(w) for some $w \in L'_{i-1}$. Since by assumption $v \in G'$ at the time of execution of move(v), by construction the level of v remains i during all the intermediate operations $k'+1, \ldots, k-1$. Furthermore, L'_{i-1} does not contain any parent of v before the execution of the first such intermediate operation, hence at that time the parents of v are in level $i-2$ or lower. Therefore, any intermediate operation which is a move() or delete() keeps the invariant that the parents of v are in level $i-2$ or lower as every move() operation can only decrease the level of a vertex. Any intermediate operation which is an insert() cannot add a parent of v at all, since vertices are inserted in topological order. Hence L'_{i-1} does not contain parents of v at the time of the execution of move(v), a contradiction.

(4) This case trivially follows by case (3) of the invariant.

(5) By induction, the invariant was satisfied right before the last execution of insert(v). We claim that after any operation the list MOVE contains all vertices for which this invariant is not satisfied. Right before the last insert(v) operation, MOVE $= \emptyset$. The insert(v) operation simply adds v either to layer L'_1 if there is no parent of v in G', or to layer $L_{h(v)}$ such that $L_{h(v)-1}$ contains a parent of v. Clearly, the claim holds as MOVE $= \emptyset$ and the invariant is satisfied for v. Consider now a delete(v) operation on a vertex $v \in L'_i$. The additional vertices that violate the invariant after this operation are the vertices $u \in L'_{i+1}$ whose only parent in L'_i is v. Recall, we keep track of the parent of u in L'_i in the list $L_{\text{next}}(w)$. Since *delete*(v) tests whether $L_{\text{next}}(u) = \emptyset$ after removing v from L'_i for all children u of v in L'_{i+1}, all these vertices are correctly inserted to MOVE. Thus, the claim holds also after a delete(v) operation. Finally, we consider the case of a move(v) operation. The move(v) first removes a vertex v from a list L_i. At the first stage of move(v), similarly to the delete(v) operation, the additional vertices that violate the invariant are the vertices $u \in L'_{i+1}$ whose only parent in L'_i is v. Arguing in the exact same way as the delete(v) operation, all the additional vertices that violate the invariant (i.e., the ones that are not already in MOVE) are correctly added to MOVE. To complete our claim, notice that (similarly to insert()) move(v) adds v to either adds v to layer L'_1 if there is no parent of v in G', or to $L'_{h(v)}$ such that $L'_{h(v)-1}$ contains a parent of v. Hence, the invariant is satisfied for v after move(v), and therefore our claim holds. The proof of the invariant follows by the fact that MOVE $= \emptyset$ right before an insert(v) operation. □

Lemma 5.1.4. *The algorithm described above (pseudo-code in Algorithm 6) computes an* $(\ell, \frac{2n}{\ell})$*-decomposition.*

Proof. By construction each vertex which is included in a chain or antichain in the first n iterations is deleted from G', hence it is not included in any following chain or antichain. The antichains added to $\mathcal{Q}$ in iteration $n+1$ are disjoint by Invariant 5.1.2. Furthermore, all vertices

are added at some point to G', hence they are included by construction in some chain or antichain at some later point. Thus the chains and antichains induce a partition of the vertex set V.

Each list $L_{next}(v)$ by construction contains parents of v only. Furthermore, right before the $\mathsf{insert}(v_t)$ operation that leads to the construction of some set C_t, each vertex $w \in L'_i$, $i \geq 2$, must have at least one parent in L'_{i-1} by construction (by Invariant 5.1.2), hence $L_{\text{next}}(w)$ is not empty and contains vertices in G'. Consequently C_t is a chain in G' of size precisely h.

Similarly, by Invariant 5.1.2, vertices in each set L'_i that are added to the set $\mathcal{Q}$ of antichains are not parents of each other, hence they form a correct antichain. Notice also that all the sets L'_i that are added to $\mathcal{Q}$ in the first n iterations have size precisely ℓ and consist of vertices in G' only. Indeed, the condition $|L'_i| = \ell$ happens after an $\mathsf{insert}()$ or $\mathsf{move}()$ operation. In both cases there are no vertices in DEL, hence any vertex in L'_i is also present in G'.

It remains to bound the number of chains and antichains. As argued before, each chain C_t has size precisely $h \geq n/\ell$. Hence disjointness implies that there are at most ℓ such chains. Similarly, each antichain that we add in the first n iterations has size precisely ℓ, hence disjointness implies that there are at most n/ℓ such antichains. In the final iteration $n+1$ we add at most $h - 1 \leq n/\ell$ extra antichains. The claim follows. □

Lemma 5.1.5. *The above algorithm (pseudo-code in Algorithm 6) takes $\mathcal{O}(n^2)$ time.*

Proof. The running time is dominated by the execution of the operations $\mathsf{insert}()$, $\mathsf{delete}()$ and $\mathsf{move}()$. We execute $\mathsf{delete}(v)$ at most once on each $v \in V$. Assume $v \in L'_i$ at time of execution. This operation requires to remove v from $|L_{\text{prev}}(v)|$ lists $L_{\text{next}}(w)$ of vertices $w \in L'_{i+1}$: notice that we maintain pointers to the occurrence of v in each of these lists $L_{\text{next}}(w)$, hence this operation can be performed in time $\mathcal{O}(\ell)$ since by Invariant 5.1.2 $|L_{\text{prev}}(v)| \leq |L'_{i+1}| \leq \ell$. For $i \geq 1$, we also need to remove v from the list $L_{\text{prev}}(u)$ of some $u \in L_{i-1}$. The same invariant guarantees that $|L_{\text{prev}}(u)| \leq |L'_i| \leq \ell$, and the fact that we store pointers of the occurrence of v in each of these lists $L_{\text{prev}}(u)$, and hence this step also takes $\mathcal{O}(\ell)$ time. Thus the total cost of $\mathsf{delete}()$ operations is $\mathcal{O}(n\ell)$.

Similarly, $\mathsf{insert}(v)$ is executed at most once on each $v \in V$, and this operation can be easily performed in time $\mathcal{O}(n)$. Hence the total cost of $\mathsf{insert}()$ operations is $\mathcal{O}(n^2)$.

It remains to consider the cost of $\mathsf{move}()$ operations. Let us focus on the operations of type $\mathsf{move}(v)$ for a specific vertex v (notice that the same vertex v can be moved multiple times). Assume $v \in L'_i$ at that time, and v is moved to layer L'_j. Recall that by construction $j < i$. Similarly to the $\mathsf{delete}(v)$ case, we spend $\mathcal{O}(\ell)$ time to remove v from affected lists $L_{\text{next}}(w)$, $w \in L'_{i+1}$, and $L_{\text{prev}}(u)$, $u \in L'_{i-1}$. Analogously, we spend $\mathcal{O}(\ell)$ time to create the new list $L_{\text{next}}(v)$ and $\mathcal{O}(1)$ time to update $L_{\text{prev}}(u)$ for some $u \in L'_{j-1}$. The rest of the operations can be easily performed in time $\mathcal{O}(\ell)$ for each level between $i+2$ and $j-1$. Hence the cost of this operation $move(v)$ is $\mathcal{O}((j-i)\ell)$. Since the largest possible level of a vertex v on which we execute $\mathsf{move}(v)$ is $h-1$, a simple sum argument shows that the total cost of $\mathsf{move}(v)$ operations involving the same vertex v is $\mathcal{O}(h\ell) = \mathcal{O}(n)$. Hence the total cost of $\mathsf{move}()$ operations is $\mathcal{O}(n^2)$. □

Algorithm 7: Move vertex v to a lower layer

```
1  Procedure move(v)
2      Remove v from L'_{h(v)}, decrement |L'_{h(v)}|
3      foreach w ∈ L_prev(v) do
4          Remove v from L_next(w)
5          if L_next(w) = ∅ then
6              Add w to MOVE
7      L_next(v) ← ∅, L_prev(v) ← ∅
8      for j = h(v) − 2, ..., 1 do
9          foreach u ∈ L'_j do
10             if (u, v) ∈ E(G') then
11                 Add u to L_next(v)
12         if L_next(v) ≠ ∅ then
13             Add v to L'_{j+1}, increment |L'_{j+1}|, and set h(v) ← j + 1
14             for u ∈ L_next(v) do
15                 Add v to L_prev(u)
16             return
17     Add v to L'_1, increment |L'_1|, and set h(v) ← 1
18     foreach u ∈ L'_2 do
19         if (v, u) ∈ E(G') then
20             Add u to L_prev(v) and v to L_next(u)
```

Algorithm 8: Insert vertex v

```
1  Procedure insert(v)
2      L_prev(v) ← ∅, L_next(v) ← ∅
3      for i = h − 1, h − 2, ..., 1 do
4          foreach u ∈ L'_i do
5              if (u, v) ∈ E(G') then
6                  Add u to L_next(v)
7          if L_next(v) ≠ ∅ then
8              Add v to L'_{i+1}, increment |L'_{i+1}|, and set h(v) ← i + 1
9              for u ∈ L_next(v) do
10                 Add v to L_prev(u)
11             return
12     Add v to L'_1, increment |L'_1|, and set h(v) ← 1
```

Algorithm 9: Delete vertex v

```
1 Procedure delete(v)
2     Remove v from G' and from L'_{h(v)}, decrement |L'_{h(v)}|
3     foreach u ∈ L_next(v) do
4         Remove v from L_prev(u) and u from L_next(v)
5     foreach w ∈ L_prev(v) do
6         Remove v from L_next(w) and w from L_prev(v)
7         if L_next(w) = ∅ then
8             Add w to MOVE
```

5.2 Faster decomposition in sparse graphs

We observe that our decomposition algorithm can be implemented more efficiently in sparse graphs (more precisely, whenever $m/\ell \ll n$). This is not critical in our application, since the number of edges will be $\Theta(n^2)$ in our case. However, since this might be helpful in other applications, we give the details in the following.

Theorem 5.2.1. *Let $G = (V, E)$ be a DAG with n vertices and m edges, represented via adjacency lists, and let $\ell \in [1, n]$ be an integer parameter. Then there exists an $\mathcal{O}(\frac{mn}{\ell})$ time deterministic algorithm to compute an $(\ell, \frac{2n}{\ell})$ decomposition of G.*

Proof. We modify the above algorithm as follows. We do not compute the adjacency matrix of G, and we compute the topological order of G in time $\mathcal{O}(m + n)$. In each $\mathsf{insert}(v)$ operation, we simply scan the in-neighbors of v and check in which layer they are to identify the layer where v has to be inserted. We similarly modify the involved lists $L_{\text{next}}()$ and $L_{\text{prev}}()$. Hence we can perform this operation in time $\mathcal{O}(\mathsf{deg}(v))$, where $\mathsf{deg}(v)$ is the degree of v in G. Similarly, for each $\mathsf{delete}(v)$ operation, $v \in L'_i$, we consider the out-neighbors of G and check which ones belong to L'_{i+1}. Hence also this operation can be performed in time $\mathcal{O}(\mathsf{deg}(v))$. Thus $\mathsf{insert}()$ and $\mathsf{delete}()$ operations cost $\mathcal{O}(m)$ in total. In each $\mathsf{move}(v)$ operation we consider all parents of v and identify the lowest layer of any such parent. Hence this operation can be implemented in $\mathcal{O}(\mathsf{deg}(v))$ time. By construction each time we execute $\mathsf{move}(v)$, v is moved to a strictly lower level. Since the largest level of a vertex v on which we execute $\mathsf{move}(v)$ is $h - 1$, we can perform this operation at most $h - 2$ times. So the total cost of $\mathsf{move}()$ operations is $\mathcal{O}(\sum_{v \in V} \mathsf{deg}(v)\frac{n}{\ell}) = \mathcal{O}(\frac{mn}{\ell})$. The claim follows. □

Recall that we have already shown a greedy Algorithm 1, that calculates $(\ell, \frac{n}{\ell})$ decomposition in $\mathcal{O}(m\ell)$ time, which is better than $\mathcal{O}(\frac{mn}{\ell})$ iff $\ell < \sqrt{n}$. This immediately yields the following corollary.

Corollary 5.2.2. *Let $G = (V, E)$ be a DAG with n vertices and m edges, represented via adjacency lists, and let $\ell \in [1, n]$ be an integer parameter. Then there exists an $\mathcal{O}(m\sqrt{n})$ time deterministic algorithm to compute an $(\ell, \frac{2n}{\ell})$ decomposition of G.*

Chapter 6

Applications of the faster decomposition

6.1 Faster LCA for smaller set of queries

We now show that if one is interested in computing the LCA for all pairs of vertices from a subset $S \subset V$ of the vertices, where $|S| = \mathcal{O}(n^\delta), \delta \leq 1$, we can modify the Algorithm 5 from Chapter 4 and get a better running time. We refer to this problem as the *S-pairs LCA* problem. We remark that obtaining faster (than the solution for the global problem) algorithm for S-pairs LCA problem is possible when using $\mathcal{O}(n\ell^2)$ decomposition (Algorithm 2). However, by computing decomposition in $\mathcal{O}(n^2)$ (Algorithm 6) we obtain even better bound on the running time.

On a high-level, the algorithm remains the same. Let G_{input} be the input DAG. We first compute in $\mathcal{O}(n^\omega)$ the transitive closure of G, and solve the S-pairs LCA problem on G. Then, we compute in $\mathcal{O}(n^2)$ time an $(n^x, 2n^{1-x})$-decomposition $(\mathcal{P}, \mathcal{Q})$ of G with the algorithm from Theorem 3.3.1. Again, the parameter x will be fixed later on to optimize the running time of the algorithm.

Incorporating the W-restricted LCAs to the context of the S-pairs LCA problem, we give the following definition.

Definition 6.1.1. *Given a DAG $G = (V, E)$, and two subsets of vertices $W, S \subseteq V$, the W-restricted S-Pairs LCA problem is to compute $\mathsf{lca}_W(u, v) \in LCA_W(u, v)$ for all pairs of vertices $u, v \in S$ ($\mathsf{lca}_W(u, v) = -\infty$ if $LCA_W(u, v) = \emptyset$).*

Similarly like in the solution for the global problem, we compute a solution to the $\mathcal{P}$-restricted and $\mathcal{Q}$-restricted S-Pairs LCA problems (that is, the values $\mathsf{lca}_\mathcal{P}(u, v)$ and $\mathsf{lca}_\mathcal{Q}(u, v)$ for all pairs of vertices $u, v \in S$), and later on combine these solutions to compute a solution to the S-pairs LCA problem. Again, we set $\mathsf{lca}(u, v) = \max\{\mathsf{lca}_\mathcal{P}(u, v), \mathsf{lca}_\mathcal{Q}(u, v)\}$, where the labels of the vertices respect a $\mathcal{Q}$-compact topological order, where $\mathcal{Q}$ is a path-respecting family of antichains of G. Throughout the section, we assume the vertices are are labeled with integers $1, \ldots, n$ according to a $\mathcal{Q}$-compact topological order.

While it is straightforward to modify our solution to the global All-Pairs LCA problem in order to get a solution to the S-pairs LCA problem, and the proof of correctness is essentially the same, we still spell-out the details for completeness.

The first modifications are in the algorithms that compute the solutions to the $\mathcal{P}$-restricted and $\mathcal{Q}$-restricted S-Pairs LCA problems. The modified version of Algorithm 3 is presented in Algorithm 10 and its proof in Lemma 6.1.2, while the modified version of Algorithm 4 is presented in Algorithm 11 and its proof in Lemma 6.1.3.

Algorithm 10: Compute $\mathsf{lca}_{\mathcal{P}}(u, v)$ for all pairs of vertices $u, v \in S$.

Data: Transitive closure graph $G = (V, E)$, a subset $S \subset V$ of vertices, and a family of chains $\mathcal{P} = \{P_1, \ldots, P_p\}$ of G where $p \leq n^x$.

Result: $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u, v)$ for each pair of vertices $u, v \in S$.

```
Initialize lca_P(·,·) with −∞
Let A be a |S| × p matrix initialized with −∞'s ;
for (u, v) ∈ V × S do
    if (u, v) ∈ E and u ∈ ⋃_{P∈P} P then
        i ← index of chain P_i such that u ∈ P_i ;
        Update A[v, i] ← max(A[v, i], u) ;
    end
end
Compute the (max,min)-product A ⍟ A^T
for all u, v ∈ S do
    lca_P(u, v) ← (A ⍟ A^T)[u, v]
end
```

Lemma 6.1.2. *Algorithm 10 computes for each pair of vertices $u, v \in S$, $|S| = n^\delta$, a $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u, v)$. The algorithm runs in $\widetilde{\mathcal{O}}(n^2 + n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}})$ time.*

Proof. The proof of correctness follows from Lemma 4.2.1. We now analyze the running time. It is clear from the description of the algorithm that the construction of matrix A takes $\mathcal{O}(n^2)$ time. Now, the (max,min)-matrix multiplication $A \circledcirc A^T$, where A is a $n^\delta \times p$ matrix, can be performed in time $\widetilde{\mathcal{O}}(n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}})$, by Theorem 2.3.3. The last step of Algorithm 10 takes $\mathcal{O}(|S|^2) = \mathcal{O}(n^{2\delta})$ time. □

Lemma 6.1.3. *Algorithm 11 computes the $\mathcal{Q}$-restricted LCA $lca_{\mathcal{Q}}(u, v) \in Q_i$ for each pair of vertices $u, v \in S \subseteq V, |S| = n^\delta$. The algorithm runs in time $\widetilde{\mathcal{O}}(n^{1+\delta} + n^{1-x+\omega(\delta,x,\delta)})$.*

Proof. The proof of correctness follows from Lemma 6.1.3. Initializing the $n^\delta \times |Q_i|$ dimensional matrix A for all q iterations takes $\mathcal{O}(\sum_{i=1}^{q} n^\delta \cdot |Q_i|) = \mathcal{O}(n^\delta n)$ time. The running time of the Boolean matrix multiplications and witness calculations is upperbounded by $\widetilde{\mathcal{O}}(\sum_{i=1}^{q} \mathrm{MM}(n^\delta, |Q_i|, n^\delta))$. Using similar reasoning like in the proof of the Lemma 4.3.4, we can show that $\widetilde{\mathcal{O}}(\sum_{i=1}^{q} \mathrm{MM}(n^\delta, |Q_i|, n^\delta)) \leq \widetilde{\mathcal{O}}(q \cdot \mathrm{MM}(n^\delta, n/q, n^\delta)) \in \mathcal{O}(n^{1-x+\omega(\delta,x,\delta)})$. □

Algorithm 11: Compute $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for all pairs of vertices $u, v \in V$.

Data: Transitive closure graph $G = (V, E)$, a subset $S \subset V$ of the vertices, and a family of antichains $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ of G that is path-respecting such that $q \leq 2n^{1-x}$.

Result: $\mathcal{Q}$-restricted LCA $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for each pair of vertices $u, v \in V$.

```
Initialize lca_Q(·,·) with −∞.
for i = q, ..., 1 do
    Initialize a |S| × |Q_i| matrix A with zeros.
    Let φ_i : Q_i --1:1--> {1, ..., |Q_i|} be an arbitrary bijection and φ_i^{-1}(·) be its inverse
     function.
    for all x ∈ |S|, y ∈ Q_i such that (y, x) ∈ E do
        A[x, φ_i(y)] ← 1
    end
    Compute A · A^T, and its witness matrix W
    for all u, v ∈ S do
        if lca_Q(u, v) = −∞ and A · A^T[u, v] ≠ 0 then
            lca_Q(u, v) ← φ_i^{-1}(W[u, v]).
        end
    end
end
```

Theorem 6.1.4. *Algorithm 12 computes for all pairs of vertices $u, v \in S$ a LCA $\mathsf{lca}(u, v)$. If $|S| = n^{\delta}$, then the algorithm runs in time $\mathcal{O}(n^{\omega} + n^{1-x+\omega(\delta,x,\delta)} + n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}})$.*

Proof. Let $|S| = \mathcal{O}(n^{\delta})$. The proof for correctness follows from Theorem 4.4.3. The running time of the algorithm is trivially $\widetilde{\mathcal{O}}(n^{\omega} + n^2 + n^{1+\delta} + n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}} + n^{1-x+\omega(\delta,x,\delta)}) = \widetilde{\mathcal{O}}(n^{\omega} + n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}} + n^{1-x+\omega(\delta,x,\delta)})$ for a fixed $x \in [0, 1]$. □

What is now left is to find optimal value of x as a function of δ. Balancing the cost terms, we need to have $1 - x + \omega(\delta, x, \delta) = \frac{\omega(\delta,x,\delta)+2\delta+x}{2}$ which is equivalent to $\omega(\delta, x, \delta) = 3x + 2\delta - 2$. By Lemma 2.2.3 $\omega(\delta, x, \delta) = \delta\omega(1, \frac{x}{\delta}, 1)$, so we get the equation $\delta\omega(1, \frac{x}{\delta}, 1) = 3x + 2\delta - 2$. The time complexity becomes $\mathcal{O}(n^{1-x+\omega(\delta,x,\delta)}) = \mathcal{O}(n^{2\delta+2x-1})$. We denote the exponent of the time complexity as $\gamma' = 2\delta + 2x - 1$.

To get numerical bounds on the time complexity we can use bounds on $\omega(1, x, 1)$ outlined in the Section 2.2. Using square matrix multiplication as a subroutine to implement rectangular matrix multiplication (i.e., the bound in (2.1)), one obtains $\delta(2 + \frac{x}{\delta}(\omega - 2)) \geq 3x + 2\delta - 2$. This, given current bound $\omega < 2.3728639$, yields $x \leq \frac{2}{5-\omega}$ and $\gamma' \leq 2\delta + \frac{4}{5-\omega} - 1 \leq 2\delta + 0.522571$.

Similarly, using the bound (2.2), one gets $x \leq \frac{2-\beta\alpha\delta}{3-\beta}$, $\gamma' \leq \frac{2(2-\beta\alpha\delta)}{3-\beta} + 2\delta - 1 \leq 0.6282973594 + 1.86112061279\delta$. If instead we apply the bound (2.3), we get $\gamma' \leq 0.711508 + 1.73504\delta$, with $x \leq 0.855754 - 0.132478\delta$.

Algorithm 12: Compute $\mathsf{lca}(u, v)$ for all pairs of vertices $u, v \in S$.

Data: DAG $G_{input} = (V, E_{input})$
Result: $\mathsf{lca}(u, v)$ for each pair of vertices $u, v \in S$

1 Compute the transitive closure graph $G = (V, E)$ of G_{input}
2 Use Algorithm 2 to compute an $(n^x, 2n^{1-x})$-decomposition into a family of chains $\mathcal{P} = \{P_1, \ldots, P_p\}$ with $p \leq n^x$ and $|P_i| \leq n^{1-x}$ and a family of antichains $\mathcal{Q}' = \{Q'_1, \ldots, Q'_{q'}\}$ with $q' \leq 2n^{1-x}$.
3 Use Lemma 4.3.2 with input $\mathcal{Q}'$ to compute a path-respecting family of antichains $\mathcal{Q} = \{Q_1, \ldots, Q_q\}$ of G where $q \leq 2n^{1-x}$.
4 Compute a $\mathcal{Q}$-compact topological order of G using Lemma 4.4.2 and rename vertices so that they are $1, \ldots, n$ according to this order
5 Use Algorithm 10 to compute $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u, v)$ for each pair of vertices $u, v \in S$.
6 Use Algorithm 11 to compute $\mathcal{Q}$-restricted LCA $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for each pair of vertices $u, v \in S$ **for** *all* $u, v \in S$ **do**
7 | $\mathsf{lca}(u, v) \leftarrow \max\{\mathsf{lca}_{\mathcal{Q}}(u, v), \mathsf{lca}_{\mathcal{P}}(u, v)\}$
8 **end**

One might ask how small δ must be in order to bring the time complexity down to $\widetilde{\mathcal{O}}(n^\omega)$ (i.e when $\gamma' \leq \omega$). When using the bound (2.1), we had $\gamma' \leq 2\delta + \frac{4}{5-\omega} - 1$, therefore we need $\delta \leq \frac{\omega+1}{2} - \frac{2}{5-\omega}$, which is bounded by 0.925146 given the current bound on ω. Similarly, we get $\delta \leq \frac{(\omega+1)(3-\beta)-4}{2(3-\beta)-2\beta\alpha} \leq 0.937374$ using the bound in (2.2) and $\delta \leq 0.957531$ if we apply the bound (2.3).

6.2 Faster max-min product implies faster All-Pairs LCA

Let $\lambda' = 0.7232761$ be the approximation of λ (the solution of equation $3x = \omega(1, x, 1)$) that we used to get the $\widetilde{\mathcal{O}}(n^{2.447})$ time for All-Pairs LCA. Recall also that we use $\omega_{\mathbb{V}}(a, b, c)$ to denote the rectangular max-min product exponent.

By Corollary 2.3.4 we have $\omega_{\mathbb{V}}(1, \lambda', 1) \leq (\omega(1, \lambda', 1) + 2 + \lambda')/2 \leq 2.4465522$. In this section we will prove that any improvement of this bound immediately implies improvement over algorithm for All-Pairs LCA problem, derived in chapter 4.

Lemma 6.2.1. $\omega_{\mathbb{V}}(1, x + c, 1) \leq \omega_{\mathbb{V}}(1, x, 1) + c$

Proof. The proof is completely analogous to the proof of Lemma 2.2.2. □

Theorem 6.2.2. *If* $\omega_{\mathbb{V}}(1, \lambda', 1) \leq \frac{\omega(1,\lambda',1)+2+\lambda'}{2} - \varepsilon$ *for some* $\varepsilon > 0$*, then there exists* $\delta > 0$ *such that All-Pairs LCA problem can be solved in* $\widetilde{\mathcal{O}}(n^{2.447-\delta})$ *time.*

Proof. By Theorem 5.1.1 we have an algorithm for All-Pairs LCA problem running in time

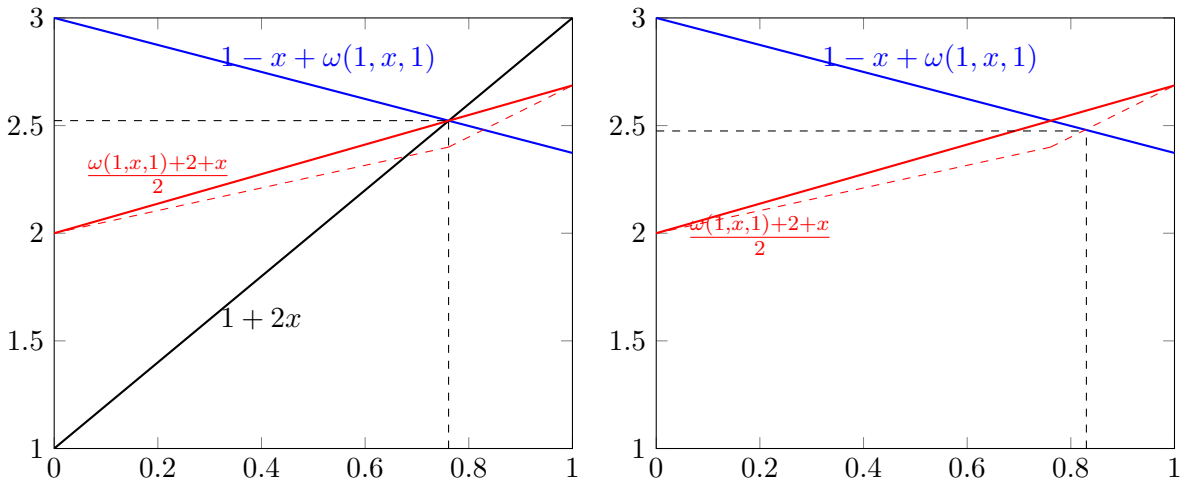

Figure 6.1: A better bound on $\omega_{\mathbb{O}}(1,\lambda',1)$ gives a better approximation of $\omega_{\mathbb{O}}(1,\cdot,1)$ (drew in dashed line). However it does not lead to a faster algorithm to All-Pairs LCA problem if we compute chain/antichain decomposition in $\widetilde{\mathcal{O}}(n^{1+2x})$ time (as pictured on the left). When we speed up decomposition down to $\mathcal{O}(n^2)$, we only need to balance two running times, and an improvement appears (pictured on the right).

$\widetilde{\mathcal{O}}(n^{\omega_{\mathbb{O}}(1,x,1)}+n^{1-x+\omega(1,x,1)})$ for a fixed $x\in[0,1]$. In particular it can run in time $n^{\omega_{\mathbb{O}}(1,\lambda'+t,1)}+n^{1-\lambda'-t+\omega(1,\lambda'+t,1)})$ for a fixed $t\in[0,1-\lambda']$.

Usinga Lemma 6.2.1 we get that $\omega_{\mathbb{O}}(1,\lambda'+t,1)\leq\frac{\omega(1,\lambda',1)+2+\lambda'}{2}+t-\varepsilon\leq 2.4465522+t-\varepsilon$. Using bound (2.3) we also get that $1-\lambda'-t+\omega(1,\lambda'+t,1)\leq 2.4465522-0.33712\cdot t$ for $t\leq 0.75-\lambda'=0.0267239$. Setting $t=\min(\frac{\epsilon}{1.33712},0.0267239)$ and $\delta=0.33712\cdot t$ we get the claimed $\widetilde{\mathcal{O}}(n^{2.447-\delta})$ running time. □

Theorem 6.2.2 shows that computation of max-min product is the source of hardness for All-Pairs LCA problem. We stress that the usage of Algorithm 6 (i.e. decomposition in time $\mathcal{O}(n^2)$) is crucial for the proof of this theorem. The reason is that now we only had to balance the running time of two algorithms (i.e computation of $\mathcal{P}$-restricted LCAs and $\mathcal{Q}$-restricted LCAs), whereas when using decomposition in time n^{1+2x}, we would need to balance the running time of three algorithms simultaneously. An improvement in running time for one of them would not lead to change in the optimal value of parameter x for the other two. We show the intuition behind this phenomenon in Figure 6.2.

Chapter 7

Conclusions and Open Problems

To the best of our knowledge, All-Pairs LCA is the first example of a natural graph problem with an algorithm based on fast matrix multiplication, which has a running time strictly between $\Omega(n^2)$ and $\mathcal{O}(n^{2.5})$, under the assumption $\omega = 2$. This might suggest that a faster algorithm exists (e.g., with a running time of $\widetilde{\mathcal{O}}(n^\omega)$). Alternatively, it would be interesting to derive fine-grained lower bounds based on All-Pairs LCAs in DAGs.

A simple greedy algorithm for decomposing a DAG into $\mathcal{O}(\sqrt{n}\,)$ chains and antichains runs in time $\mathcal{O}(n^{2.5})$ for dense graphs. Our algorithm improves this bound to $\mathcal{O}(n^2)$. In the similar problem of decomposing a sequence into $\mathcal{O}(\sqrt{n}\,)$ monotonic subsequences, a naive greedy algorithm works in time $\mathcal{O}(n^{1.5}\log(n))$. Yehuda and Fogel improved this to $\mathcal{O}(n^{1.5})$ [5] and there has been no further progress ever since. It was also noted by Jørgensen and Pettie that this is a natural example of a problem with a large $(\widetilde{\Omega}(\sqrt{n}\,))$ gap between the current algorithmic and decision-tree complexity [26]. Therefore, it would be interesting to see if the techniques developed in this work can be used to improve the time complexity of sequence decomposition. Alternatively, one could further investigate the relationship between the two problems in order to prove some lower bounds.

In Section 5.2 we showed faster algorithm for computing $(\ell, \frac{2n}{\ell})$ decomposition in sparse graphs, in particular we observed that for graphs with m edges and arbitrary value of parameter ℓ we can compute the decomposition in $\mathcal{O}(m\sqrt{n})$ time. It is an interesting open question, whenever this can be improved e.g. down to $\widetilde{\mathcal{O}}(m)$.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *STOC 1973*, pages 253–265.

[2] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997.

[3] N. Alon, Z. Galil, O. Margalit, and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *FOCS 1992*, pages 417–426.

[4] D. Arroyuelo, F. Claude, R. Dorrigiv, S. Durocher, M. He, A. López-Ortiz, J. I. Munro, P. K. Nicholson, A. Salinger, and M. Skala. Untangled monotonic chains and adaptive range search. In *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2009.

[5] R. Bar-Yehuda and S. Fogel. Partitioning a sequence into few monotone subsequences. *Acta Informatica*, 35(5):421–440, 1998.

[6] H. Barr, T. Kopelowitz, E. Porat, and L. Roditty. $\{-1, 0, 1\}$-APSP and (min, max)-product problems. *CoRR*, abs/1911.06132, 2019.

[7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94.

[8] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[9] M. A. Bender, G. Pemmasani, S. Skiena, and P. Sumazin. Finding least common ancestors in directed acyclic graphs. In *SODA 2001*, pages 845–854.

[10] A. Brandstädt and D. Kratsch. On partitions of permutations into increasing and decreasing subsequences. *J. Inf. Process. Cybern.*, 22(5/6):263–273, 1986.

[11] K. Bringmann, M. Künnemann, and K. Węgrzycki. Approximating APSP without scaling: equivalence of approximate min-plus and exact min-max. In *STOC 2019*, pages 943–954.

[12] F. Claude, J. I. Munro, and P. K. Nicholson. Range queries over untangled chains. In *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 82–93. Springer, 2010.

[13] K. Cohen and R. Yuster. On minimum witnesses for boolean matrix multiplication. *Algorithmica*, 69(2):431–442, 2014.

[14] R. W. Cottingham, R. M. Idury, and A. A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[15] A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theor. Comput. Sci.*, 380(1-2):37–46, 2007.

[16] S. K. Dash, S. Scholz, S. Herhut, and B. Christianson. A scalable approach to computing representative lowest common ancestor in directed acyclic graphs. *Theor. Comput. Sci.*, 513:25–37, 2013.

[17] R. Duan, Y. Gu, and L. Zhang. Improved time bounds for all pairs non-decreasing paths in general digraphs. In *ICALP 2018*, pages 44:1–44:14.

[18] R. Duan, C. Jin, and H. Wu. Faster algorithms for all pairs non-decreasing paths problem. In *ICALP 2019*, pages 48:1–48:13.

[19] R. Duan and S. Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *SODA 2009*, pages 384–391, 2009.

[20] L. Duraj, K. Kleiner, A. Polak, and V. V. Williams. Equivalences between triangle and range query problems. In *SODA 2020*, pages 30–47.

[21] P. Erdös, J. G. Gimbel, and D. Kratsch. Some extremal results in cochromatic and dichromatic theory. *Journal of Graph Theory*, 15(6):579–585, 1991.

[22] F. V. Fomin, D. Kratsch, and J. Novelli. Approximating minimum cocolorings. *Inf. Process. Lett.*, 84(5):285–290, 2002.

[23] F. Grandoni, G. F. Italiano, A. Łukasiewicz, N. Parotsidis, and P. Uznański. All-pairs LCA in dags: Breaking through the $O(n^{2.5})$ barrier. *CoRR*, abs/2007.08914, 2020.

[24] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[25] P. Indyk, M. Lewenstein, O. Lipsky, and E. Porat. Closest pair problems in very high dimensions. In *ICALP 2004*, pages 782–792.

[26] A. G. Jørgensen and S. Pettie. Threesomes, degenerates, and love triangles. In *FOCS 2014*, pages 621–630. IEEE Computer Society.

[27] M. Kowaluk and A. Lingas. LCA queries in directed acyclic graphs. In *ICALP 2005*, pages 241–248.

[28] M. Kowaluk and A. Lingas. Unique lowest common ancestors in dags are almost as easy as matrix multiplication. In *ESA 2007*, pages 265–274.

[29] M. Kowaluk and A. Lingas. Quantum and approximation algorithms for maximum witnesses of boolean matrix products. *CoRR*, abs/2004.14064, 2020.

[30] M. Kowaluk, A. Lingas, and J. Nowak. A path cover technique for LCAs in dags. In *SWAT 2008*, pages 222–233.

[31] K. Labib, P. Uznański, and D. Wolleb-Graf. Hamming distance completeness. In *CPM 2019*, pages 14:1–14:17.

[32] F. Le Gall. Algebraic complexity theory and matrix multiplication. In *ISSAC 2014*, page 23.

[33] F. Le Gall and F. Urrutia. Improved rectangular matrix multiplication using powers of the Coppersmith-Winograd tensor. In *SODA 2018*, pages 1029–1046.

[34] L. Lesniak and H. J. Straight. The cochromatic number of a graph. *Ars Combinatoria*, 3:39–46, 1977.

[35] A. Lincoln, A. Polak, and V. V. Williams. Monochromatic triangles, intermediate matrix products, and convolutions. In *ITCS 2020*, pages 53:1–53:18.

[36] K. Min, M. Kao, and H. Zhu. The closest pair problem under the hamming metric. In *COCOON 2009*, pages 205–214.

[37] L. Mirsky. A dual of dilworth's decomposition theorem. *The American Mathematical Monthly*, 78(8):876–877, 1971.

[38] A. A. Schäffer, S. K. Gupta, K. Shriram, and R. W. Cottingham. Avoiding recomputation in linkage analysis. *Human Heredity*, 44:225–237, 1994.

[39] A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. In *SODA 2007*, pages 978–985.

[40] A. J. Stothers. On the complexity of matrix multiplication. 2010.

[41] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *STOC 2007*, pages 585–589.

[42] K. W. Wagner. Monotonic coverings of finite sets. *J. Inf. Process. Cybern.*, 20(12):633–639, 1984.

[43] V. V. Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC 2012*, pages 887–898.

[44] V. V. Williams. Nondecreasing paths in a weighted graph or: How to optimally read a train schedule. *ACM Trans. Algorithms*, 6(4):70:1–70:24, 2010.

[45] V. V. Williams and R. Williams. Finding, minimizing, and counting weighted subgraphs. *SIAM J. Comput.*, 42(3):831–854, 2013.

[46] B. Yang, J. Chen, E. Lu, and S. Q. Zheng. A comparative study of efficient algorithms for partitioning a sequence into monotone subsequences. In *TAMC 2007*, volume 4484 of *Lecture Notes in Computer Science*, pages 46–57.

[47] R. Yuster. Efficient algorithms on sets of permutations, dominance, and real-weighted APSP. In *SODA 2009*, pages 950–957.

[48] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.

[49] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.