All-Pairs LCA in DAGs: Breaking through the $O(n^{2.5})$ barrier[*]

Fabrizio Grandoni[†] Giuseppe F. Italiano[‡] Aleksander Łukasiewicz[§] Nikos Parotsidis[¶]
Przemysław Uznański[‖]

Abstract

Let $G = (V, E)$ be an n-vertex directed acyclic graph (DAG). A *lowest common ancestor* (LCA) of two vertices u and v is a common ancestor w of u and v such that no descendant of w has the same property. In this paper, we consider the problem of computing an LCA, if any, for all pairs of vertices in a DAG. The fastest known algorithms for this problem exploit fast matrix multiplication subroutines and 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 paper, 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. A key tool in our approach is a fast algorithm to partition the vertex set of the transitive closure of G into a collection of $\mathcal{O}(\ell)$ chains and $\mathcal{O}(n/\ell)$ antichains, for a given parameter ℓ. As usual, a chain is a path while an antichain is an independent set. We then find, for all pairs of vertices, a *candidate* LCA among the chain and antichain vertices, separately. The first set is obtained via a reduction to $(\max, \min)$ matrix multiplication. The computation of the second set can be reduced to Boolean matrix multiplication similarly to previous results on this problem. We finally combine the two solutions together in a careful (non-obvious) manner.

[*]Fabrizio Grandoni is partially supported by the SNF Excellence Grant 200020B 182865/1. Giuseppe F. Italiano is partially supported by MUR, the Italian Ministry for University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data). Aleksander Łukasiewicz and Przemysław Uznański are Supported by the Polish National Science Centre grant 2019/33/B/ST6/00298.

[†]IDSIA, Lugano, Switzerland

[‡]LUISS University, Rome, Italy

[§]University of Wrocław, Wrocław, Poland

[¶]Google Research

[‖]University of Wrocław, Wrocław, Poland

1 Introduction

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 paper, 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, 37] 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, 26, 27, 29]). 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 [26] 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 $\mathcal{O}(n^{2.5286})$ 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 $\mathcal{O}(n^{2.5286})$ for the All-Pairs LCA problem.

Somewhat surprisingly, all the currently known bounds for the All-Pairs LCA problem [8, 9, 15, 26] 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.

Our result. In this paper 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 $2n/\ell$ antichains in time $\mathcal{O}(n^2)$. We refer to this as an $(\ell, 2n/\ell)$-decomposition of G.

We now sketch how to exploit this decomposition in order to compute efficiently all-pairs LCAs. Let G^T be the transitive closure of G, which can be computed in time $\mathcal{O}(n^\omega)$. Note that we can solve the All-Pairs LCA problem in G by solving the same problem in G^T. We first find an $(n^x, 2n^{1-x})$-decomposition of G^T in time $\mathcal{O}(n^2)$ for a parameter x, $0 \leq x \leq 1$, to be fixed later. Next, for each pair of vertices, we find a candidate LCA among the chain and antichain vertices, separately. The first set can be obtained in time $\widetilde{\mathcal{O}}(n^{\frac{\omega(1,x,1)+2+x}{2}})$ via a reduction to $(\max, \min)$ matrix multiplication, similarly to Bender et al. [8, 9] and to Czumaj et al. [15]. The computation of the second set can be reduced to Boolean matrix multiplication in time $\widetilde{\mathcal{O}}(n^{1-x+\omega(1,x,1)})$. Combining the two solutions is non-trivial and requires some extra care. Putting all pieces together yields a total running time of $\widetilde{\mathcal{O}}(n^\omega + n^{\frac{\omega(1,x,1)+2+x}{2}} + n^{1-x+\omega(1,x,1)})$. Balancing the last two terms implies $\omega(1, x, 1) = 3x$, and thus a running time of $\widetilde{\mathcal{O}}(n^\omega + n^{1+2x})$. Using the equation $\omega(1, x, 1) = 3x$ to tune the parameter x, yields $x = 0.7232761$, and hence a total running time of $\mathcal{O}(n^{2.447})$.

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 [26] 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).

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 [23], although 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 [39], Vassilevska-Williams [42] and finally Le Gall [31], 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 [32]. 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].

There is a large family of graph and geometric problems for which the best known algorithms use matrix-multiplication and their complexity is between n^ω and n^3. Such problems include All-Pair Bottleneck Paths (APBP): [19, 40], vertex APBP [38], unweighted directed APSP [48], All-Pair Nondecreasing Paths [17, 18, 43], and Dominance-, Hamming- and L_1-matrix products: [24, 35, 46]. 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, 34].

For other results on All-Pairs LCA, see [27] on finding unique LCA. Other work in the area include [16].

For related problems of minimal witnesses of Boolean matrix multiplication, we refer to an algorithm for sparse matrices [13], and to a recent quantum algorithm [28]. We also refer to [29] which introduced path covering techniques 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 imply 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 [33]. The special case of partitioning a sequence into monotonic subsequences has received considerable attention [5, 10, 21, 22, 41, 45], since it has many applications, including book embeddings [5], and geometric algorithms [4, 5, 12].

2 Preliminaries

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 paper, we consider the following problem.

Problem 2.1. (All-Pairs LCA) *Let $G = (V, E)$ be a DAG. Compute a lowest common ancestor for all pairs of vertices $u, v \in V$.*

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 $\omega < 2.3728639$ [31]. 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. *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 better bounds on $\omega(1, x, 1)$ (see, e.g., [32]). In particular, the following bound is known:

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

We now sketch how all those bounds influence the $\widetilde{\mathcal{O}}(n^\omega + n^{1+2x})$ running time of our algorithm. If we simply use square matrix multiplication as a subroutine to implement rectangular matrix multiplication (i.e., the bound in (2.1)), combining this with equation $3x = \omega(1, x, 1)$, we obtain $x = \frac{2}{5-\omega}$. 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 $x = \frac{2-\omega\alpha}{5-\omega-3\alpha}$ and 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 (2.3), we finally get $x = 0.7232758$, which yields our claimed running time of $\mathcal{O}(n^{2.4465522})$.

Max-min matrix product. We exploit fast algorithms for the max-min matrix product. In more detail, let A be an $n \times p$ matrix and B be a $p \times n$ matrix. The entries of A and B are assumed to come from $\mathbb{Z} \cup \{-\infty\}$. The max-min product $C = A \mathbin{\overset{\vee}{\wedge}} B$ is specified by $C[i, j] = \max_k \min\{A[i, k], B[k, j]\}$. A simple modification of the algorithm and analysis in [19] implies the following complexity (for which we provide a short proof in Section 6 for completeness).

Theorem 2.1. (Corollary of [19]) *If A and B are respectively $n \times p$ and $p \times n$ matrices, then the $A \mathbin{\overset{\vee}{\wedge}} B$ product can be computed in time $\widetilde{\mathcal{O}}(\sqrt{MM(n, p, n) \cdot n^2 p})$.*

3 Fast Chain-Antichain Decomposition

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. *An (a, b)-decomposition of a DAG G consists of a collection $\mathcal{P}$ of chains of G, $|\mathcal{P}| \leq a$, and*

Algorithm 1 Compute chain/antichain decomposition

Input: DAG G with the topological ordering $v_1, \ldots, v_n$ of its vertices and parameter ℓ, with $h = \lceil n/\ell \rceil$.
Output: $(\ell, \frac{2n}{\ell})$-decomposition of G.

```
1: Initialize $G' \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$, $\mathcal{Q} \leftarrow \emptyset$, $L'_i \leftarrow \emptyset$ for $1 \le i \le h$
2: for $t = 1, \ldots, n$ do
3:    $\mathsf{MOVE} \leftarrow \emptyset$, $\mathsf{DEL} \leftarrow \emptyset$
4:    $\mathsf{insert}(v_t)$
5:    if $|L'_{h(v_t)}| \ge \ell$ then
6:       Add antichain $L'_{h(v_t)}$ to $\mathcal{Q}$ and all its vertices to $\mathsf{DEL}$
7:    else if $h(v_t) = h$ then
8:       $u \leftarrow v_t$
9:       $C_t \leftarrow \{u\}$
10:      while $L_{\text{next}}(u) \neq \emptyset$ do
11:         $u \leftarrow$ any element of $L_{\text{next}}(u)$
12:         Add $u$ to $C_t$
13:      end while
14:      Add chain $C_t$ to $\mathcal{P}$ and all its vertices to $\mathsf{DEL}$
15:   end if
16:   while $\mathsf{DEL} \cup \mathsf{MOVE} \neq \emptyset$ do
17:      if $\mathsf{DEL} \neq \emptyset$ then
18:         Extract $w$ from $\mathsf{DEL}$ and execute $\mathsf{delete}(w)$
19:      else
20:         Extract $w$ from $\mathsf{MOVE}$ and execute $move(w)$
21:         if $|L'_{h(w)}| \ge \ell$ then
22:            Add antichain $L'_{h(w)}$ to $\mathcal{Q}$ and all its vertices to $\mathsf{DEL}$
23:         end if
24:      end if
25:   end while
26: end for
27: return $(\mathcal{P}, \mathcal{Q})$
```

a collection $\mathcal{Q}$ of antichains of G, $|\mathcal{Q}| \le b$, that together span all the vertices of G.

One could find an $(\ell, n/\ell)$-decomposition with a simple greedy method, as follows: find and remove the longest chain in G, for ℓ times in total. Next, cover whatever remains with n/ℓ antichains (since the remaining graph has depth at most n/ℓ). Such an algorithm takes time $\mathcal{O}(m\ell)$ in total ($\mathcal{O}(n^2\ell)$ for dense graphs), since finding "naively" the longest chain in a DAG can be accomplished by a single graph traversal. Unfortunately, this running time would be too slow for our purposes.

We next present a faster algorithm for computing an $(\ell, \frac{2n}{\ell})$-decomposition of a DAG G, as stated in the following theorem, which will be proven in this section.

THEOREM 3.1. *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.*

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 $V_i = \{v_1, \ldots, v_i\}$, $1 \le i \le n$, denote the first i vertices in the topological order. The algorithm consists of $(n+1)$ iterations. At the beginning of iteration $t \ge 1$ 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 $1 \le t \le n$, 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 $|L'_i| \le \ell$, $1 \le i \le h$, 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 $L'_i, 1 \le i \le h$, such that each vertex $v \in L'_i$, for $i \ge 2$, 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 into 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 $w \in L_{\text{prev}}(v)$ a pointer to the occurrence of v in the list $L_{\text{next}}(w)$. We also store with each entry $v \in L_{\text{next}}(w)$ a pointer to the occurrence of w in the list $L_{\text{prev}}(v)$. 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:

- insert(v): adds vertex v to G'. This is applied once to v_t at the beginning of iteration t.
- delete(v): deletes vertex v from G'. This is used to remove chains and antichains from G'.
- 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:

Invariant 3.2. *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).*

Procedure 2 insert(v): Insert vertex v

```
1: $L_{\text{prev}}(v) \leftarrow \emptyset$, $L_{\text{next}}(v) \leftarrow \emptyset$
2: for $i = h-1, h-2, \ldots, 1$ do
3:     for each $u \in L'_i$ do
4:         if $(u, v) \in E(G')$ then
5:             Add $u$ to $L_{\text{next}}(v)$
6:         end if
7:     end for
8:     if $L_{\text{next}}(v) \neq \emptyset$ then
9:         Add $v$ to $L'_{i+1}$, increment $|L'_{i+1}|$, and set $h(v) \leftarrow i+1$
10:        for $u \in L_{\text{next}}(v)$ do
11:            Add $v$ to $L_{\text{prev}}(u)$
12:        end for
13:        return
14:    end if
15: end for
16: Add $v$ to $L'_1$, increment $|L'_1|$, and set $h(v) \leftarrow 1$
```

Procedure 3 delete(v): Delete vertex v

```
1: Remove $v$ from $G'$ and from $L'_{h(v)}$, decrement $|L'_{h(v)}|$
2: for each $u \in L_{next}(v)$ do
3:     Remove $v$ from $L_{\text{prev}}(u)$ and $u$ from $L_{\text{next}}(v)$
4: end for
5: for each $w \in L_{\text{prev}}(v)$ do
6:     Remove $v$ from $L_{\text{next}}(w)$ and $w$ from $L_{\text{prev}}(v)$
7:     if $L_{\text{next}}(w) = \emptyset$ then
8:         Add $w$ to MOVE
9:     end if
10: end for
```

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 3.2 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 3.2, 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 3.2, 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 $L_{\text{prev}}(v) = \emptyset$ (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 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.

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

Procedure 4 move(v): Move vertex v to a lower layer

```
1: Remove $v$ from $L'_{h(v)}$, decrement $|L'_{h(v)}|$
2: for each $w \in L_{\text{prev}}(v)$ do
3:     Remove $v$ from $L_{\text{next}}(w)$
4:     if $L_{\text{next}}(w) = \emptyset$ then
5:         Add $w$ to MOVE
6:     end if
7: end for
8: $L_{\text{next}}(v) \leftarrow \emptyset$, $L_{\text{prev}}(v) \leftarrow \emptyset$
9: for $j = h(v)-2, \ldots, 1$ do
10:     for each $u \in L'_j$ do
11:         if $(u, v) \in E(G')$ then
12:             Add $u$ to $L_{\text{next}}(v)$
13:         end if
14:     end for
15:     if $L_{\text{next}}(v) \neq \emptyset$ then
16:         Add $v$ to $L'_{j+1}$, increment $|L'_{j+1}|$, and set
            $h(v) \leftarrow j+1$
17:         for $u \in L_{\text{next}}(v)$ do
18:             Add $v$ to $L_{\text{prev}}(u)$
19:         end for
20:         return
21:     end if
22: end for
23: Add $v$ to $L'_1$, increment $|L'_1|$, and set $h(v) \leftarrow 1$
24: for each $u \in L'_2$ do
25:     if $(v, u) \in E(G')$ then
26:         Add $u$ to $L_{\text{prev}}(v)$ and $v$ to $L_{\text{next}}(u)$
27:     end if
28: end for
```

It remains to describe 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 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.

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 3.2). 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. *We defer to the Section 5 the proof that after each execution of* $\mathsf{insert}()$, $\mathsf{delete}()$ *or* $\mathsf{move}()$ *Invariant 3.2 is satisfied.*

At the end of the last iteration by Invariant 3.2 all vertices still in G' are contained in some L'_i, $i = 1, \dots, h-1$. We execute a special final iteration $n+1$ where we add each such set L'_i as an antichain to our decomposition.

Lemma 3.1. *The algorithm described above (pseudocode in Algorithm 1) 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 3.2. 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 3.2), 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 3.2, 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'.

Finally, we 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. $\square$

We defer to the Section 5 the proof that the running time of the algorithm is $\mathcal{O}(n^2)$.

4 All-Pairs LCA in DAGs

In this section 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 $\mathcal{O}(n^2)$ time with the algorithm from Theorem 3.1. Recall that $\mathcal{P} = \{P_1, \dots, P_p\}$ is a set of $p \leq n^x$ chains and $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ 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. *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 resp., and also define analogously $\text{LCA}_{\mathcal{P}}(\cdot,\cdot)$, $\mathsf{lca}_{\mathcal{P}}(\cdot,\cdot)$ 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 4.1 and 4.2 resp. 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 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 4.3, 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.

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

Input: Transitive closure graph $G = (V, E)$, and a family of chains $\mathcal{P} = \{P_1, \dots, P_p\}$ of G where $p \leq n^x$.
Output: $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u, v)$ for each pair of vertices $u, v \in V$.

```
1:  Initialize $\mathsf{lca}_{\mathcal{P}}(\cdot,\cdot)$ with $-\infty$
2:  Let $A$ be an $n \times p$ matrix
3:  for $P_i \in \mathcal{P}$ do
4:      for $v \in V$ do
5:          Let $w_i(v)$ be the parent of $v$ in $P_i$ with the
            largest index, otherwise $w_i(v) = -\infty$
6:          Set $A[v, i] \leftarrow w_i(v)$
7:      end for
8:  end for
9:  Compute the (max,min)-product $A \otimes A^T$
10: for all $u, v \in V, u \neq v$ do
11:     $\mathsf{lca}_{\mathcal{P}}(u, v) \leftarrow (A \otimes A^T)[u, v]$
12: end for
```

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

4.1 Computing $\mathcal{P}$-Restricted LCAs In this section we present our algorithm for the $\mathcal{P}$-restricted All-Pairs LCA problem. We next assume that vertices are labeled with integers $1, \dots, n$ according to some topological order. In the next section we will specify such ordering in a more careful way in order to achieve our final result.

Our algorithm works as follows (see also the pseudo-code in Algorithm 5). For each vertex v and each chain P_i, we compute the ancestor $w_i(v)$ of v in P_i with largest index ($w_i(v) = -\infty$ if there is no such ancestor). Next, for each pair of vertices u, v and each P_i, we compute $w_i(u, v) = \min\{w_i(u), w_i(v)\}$. 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 4.1. *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 5). 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 Theorem 2.1. The claim follows.

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. $\square$

4.2 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 $\mathcal{Q}' = \{Q'_1, \dots, Q_{q'}\}$. 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 4.2. *Let $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ 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 4.2. (FOLKLORE) *Given a transitive closure graph $G = (V, E)$ and a collection of q' disjoint antichains $\mathcal{Q}' = \{Q'_1, \dots, Q'_{q'}\}$ over the vertex set $W \subseteq V$, a greedy algorithm computes a partition of W into a collection of $q \leq q'$ disjoint antichains $\mathcal{Q} = \{Q_1, \dots, Q_q\}$ 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. [36]) 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 4.2 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 aim, 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 roughly in the same amount of time.

Theorem 4.1. (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 in Algorithm 6). 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 an $n \times n^x$ 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[2]. 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. *The $\mathcal{Q}$-restricted All-Pairs LCA problem can be solved in time $\widetilde{\mathcal{O}}(n^{1-x+\omega(1,x,1)})$.*

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

Algorithm 6 Compute $\mathsf{lca}_{\mathcal{Q}}(u,v)$ for all pairs of vertices $u, v \in V$.

Input: 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 $q \leq 2n^{1-x}$.
Output: $\mathcal{Q}$-restricted LCA $\mathsf{lca}_{\mathcal{Q}}(u,v)$ for each pair of vertices $u, v \in V$. Initialize $\mathsf{lca}_{\mathcal{Q}}(\cdot,\cdot)$ with $-\infty$.

```
1:  for $i = q, \ldots, 1$ do
2:      Initialize an $n \times n^x$ matrix $A$ with zeros
3:      Let $\phi_i : Q_i \xrightarrow{1:1} \{1, \ldots, |Q_i|\}$ be an arbitrary
        bijection and $\phi_i^{-1}(\cdot)$ be its inverse function
4:      for all $x \in V, y \in Q_i$ such that $(x,y) \in E$ do
5:          $A[x, \phi_i(y)] \leftarrow 1$
6:      end for
7:      Compute $A \cdot A^T$, and its witness matrix $W$
8:      for all $u, v \in V, u \neq v$ do
9:          if $\mathsf{lca}_{\mathcal{Q}}(u,v) = -\infty$ and $A \cdot A^T[u,v] \neq 0$ then
10:             $\mathsf{lca}_{\mathcal{Q}}(u,v) \leftarrow \phi_i^{-1}(W[u,v])$
11:         end if
12:     end for
13: end for
```

Proof. Consider the above algorithm (pseudo-code in Algorithm 6). 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 $\mathrm{MM}(n,\cdot,n)$

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

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 transitive closure). 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. ∎

4.3 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 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* 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, guarantees 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 4.3. *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. *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 4.2, where we keep track of vertices whose in-degree becomes zero after the removal of other vertices.

Algorithm 7 Compute $\mathsf{lca}_{\mathcal{V}}(u,v)$ for all pairs of vertices $u, v \in V$.

```
Input: DAG $G_{input} = (V, E_{input})$
Output: $\mathsf{lca}(u,v)$ for each pair of vertices $u, v \in V$
1: Compute the transitive closure graph $G = (V, E)$ of $G_{input}$
2: Use Algorithm 1 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 4.2 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 and rename vertices so that they are $1, \ldots, n$ according to this order
5: Use Algorithm 5 to compute $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u,v)$ for each pair of vertices $u, v \in V$
6: Use Algorithm 6 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:     $\mathsf{lca}(u,v) \leftarrow \max\{\mathsf{lca}_{\mathcal{Q}}(u,v), \mathsf{lca}_{\mathcal{P}}(u,v)\}$
9: end for
```

For the correctness, trivially by construction the indexes of the vertices in the same anti-chain 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 x 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 in Algorithm 7).

Theorem 4.2. (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 7). The running time of the algorithm is $\widetilde{\mathcal{O}}(n^{\omega} + n^{\frac{\omega(1,x,1)+2+x}{2}} + n^{1-x+\omega(1,x,1)})$ for a fixed $x \in [0, 1]$. The claimed running time is obtained by imposing $\frac{\omega(1,x,1)+2+x}{2} = 1-x+\omega(1, x, 1)$, and observing that $\omega \leq \omega(1, x, 1) + 1 - x$ for any $x \geq 0$.

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 $w' > w$. By construction we have $\mathsf{lca}_{\mathcal{P}}(u, v) \geq w'$. Hence we get a contradiction $w = \mathsf{lca}_{\mathcal{Q}}(u, v) \geq \mathsf{lca}_{\mathcal{P}}(u, v) \geq w' > w$. ☐

5 Missing proofs from Section 3.

Lemma 5.1. *After each execution of insert(), delete() or move(), Invariant 3.2 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

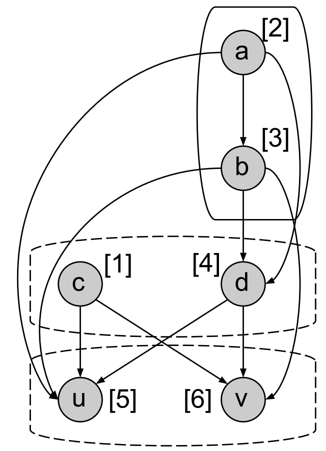

Figure 1: An example of a topological order in a transitive closure graph G that is not suitable for combining the $\mathcal{Q}$-restricted LCA and the $\mathcal{P}$-restricted LCA for the pair of vertices u, v. The family of chains in G is $\mathcal{P} = \{\{a, b\}\}$ and the path-respecting family 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 $\mathsf{lca}(u, v) = b$.

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 $L'_h \neq \emptyset$ 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 $\mathsf{delete}()$ and $\mathsf{move}()$ never add vertices to L'_h. Hence before the execution of the next $\mathsf{insert}()$ the set L'_h is empty as required.

(2) Clearly if the k-th operation is $\mathsf{delete}(v)$ the claim holds. If the k-th operation is $\mathsf{move}(v)$ or $\mathsf{insert}(v)$ 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 $\mathsf{delete}()$ operations over the $\mathsf{move}()$ 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 $\mathsf{move}()$ or $\mathsf{insert}()$. The claim then holds.

(3) This is the most delicate claim. The claim trivially holds if the k-th operation is $\mathsf{delete}(v)$, and it holds by construction if it is $\mathsf{insert}(v)$. Next assume that the k-th operation is $\mathsf{move}(v)$, 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 $\mathsf{move}(v)$ 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 $\mathsf{move}(w)$ or a $\mathsf{delete}(w)$ for some $w \in L'_{i-1}$. Since by assumption $v \in G'$ at the time of execution of $\mathsf{move}(v)$, by construction the level of v remains i during all the intermediate operations $k'+1,\dots,k-1$. 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 $\mathsf{move}()$ or $\mathsf{delete}()$ keeps the invariant that the parents of v are in level $i-2$ or lower as every $\mathsf{move}()$ operation can only decrease the level of a vertex. Any intermediate operation which is an $\mathsf{insert}()$ 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 $\mathsf{move}(v)$, 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 $\mathsf{insert}(v)$. We claim that after any operation the list MOVE contains all vertices for which this invariant is not satisfied. Right before the last $\mathsf{insert}(v)$ operation, $\mathsf{MOVE} = \emptyset$. The $\mathsf{insert}(v)$ 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 $\mathsf{MOVE} = \emptyset$ and the invariant is satisfied for v. Consider now a $\mathsf{delete}(v)$ 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 $\mathsf{delete}(v)$ operation. Finally, we consider the case of a $\mathsf{move}(v)$ operation. The $\mathsf{move}(v)$ first removes a vertex v from a list L_i. At the first stage of $\mathsf{move}(v)$, similarly to the $\mathsf{delete}(v)$ 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 $\mathsf{delete}(v)$ 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 $\mathsf{insert}()$) $\mathsf{move}(v)$ 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 $\mathsf{move}(v)$, and therefore our claim holds. The proof of the invariant follows by the fact that $\mathsf{MOVE} = \emptyset$ right before an $\mathsf{insert}(v)$ operation. □

LEMMA 5.2. *The above algorithm (pseudo-code in Algorithm 1) 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 3.2 $|L_{\text{prev}}(v)| \le |L'_{i+1}| \le \ell$. For $i \ge 1$, 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 $|L_{\text{prev}}(u)| \le |L'_i| \le \ell$, 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 $\mathcal{O}(n^2)$. □

6 Rectangular $\max, \min$-product

Recall the definition of dominance product of two matrices A, B: $(A \otimes B)[i,j] = \max_k \min(A[i,k], B[k,j])$. In the following, we find useful the dominance product: $A \otimes B$ as $(A \otimes B)[i,j] = |\{k : A[i,k] < B[k,j]\}|$.

LEMMA 6.1. (C.F. THM 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* $A \otimes B$ *can be computed in time* $\widetilde{\mathcal{O}}(MM(n,p,n) + m_1m_2/p)$.

Proof. By reductions presented in [44] (see [30] for alternative exposition), dominance product reduces to $\mathcal{O}(\log n)$ Hamming products with the 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 [30] 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 [47], 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_1m_2/p. □

We provide following theorem for completeness (note, that we provide version with extra log factor, for the sake of shortening the proof).

Theorem 2.1. (Corollary of [19]) *If A and B are respectively $n \times p$ and $p \times n$ matrices, then the $A \otimes B$ product can be computed in time* $\widetilde{\mathcal{O}}(\sqrt{MM(n,p,n) \cdot n^2p})$.

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 $L_1, \dots, L_t$, where each L_r contains at most $\lceil 2np/t \rceil$ consecutive values from L. We then construct sparse matrices $A_1, \dots, A_t$ and $B_1, \dots, B_t$ 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: $A_r \otimes B$, $A'_r \otimes B$ and $A''_r \otimes B$. 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 6.1, this takes $\widetilde{\mathcal{O}}(MM(n,p,n) + n^2p/t^2)$ for each product. The postprocessing phase takes $\mathcal{O}(p/t)$ time for each n^2 elements of the output. The total time is then $\widetilde{\mathcal{O}}(MM(n,p,n)t + n^2p/t)$, so setting $t = \sqrt{n^2p/MM(n,p,n)}$ implies the runtime bound. □

7 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 7.1. *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. □

8 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 from Section 4 to run faster. We refer to this problem as the *S-pairs LCA* problem.

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 3.1. 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.

Algorithm 8 Compute $\mathsf{lca}_{\mathcal{P}}(u,v)$ for all pairs of vertices $u,v \in S$.

Input: 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 $p \le n^x$.
Output: $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u,v)$ for each pair of vertices $u,v \in S$.

```
 1: Initialize $\mathsf{lca}_{\mathcal{P}}(\cdot,\cdot)$ with $-\infty$
 2: Let $A$ be a $|S| \times p$ matrix
 3: for $P_i \in \mathcal{P}$ do
 4:     for $v \in S$ do
 5:         Let $w_i(v)$ be the parent of $v$ in $P_i$ with the
            largest index, otherwise $w_i(v)=-\infty$
 6:         Set $A[v,i] \leftarrow w_i(v)$
 7:     end for
 8: end for
 9: Compute the (max,min)-product $A \circledvee A^T$
10: for all $u,v \in S, u \ne v$ do
11:     $\mathsf{lca}_{\mathcal{P}}(u,v) \leftarrow (A \circledvee A^T)[u,v]$
12: end for
```

Definition 8.1. *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 to Section 4, 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 the modifications to the algorithm from Section 4 are straightforward, 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 5 is presented in Algorithm 8 and its proof in Lemma 8.1, while the modified version of Algorithm 6 is presented in Algorithm 9 and its proof in Lemma 8.2.

Lemma 8.1. *Algorithm 8 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 $\widetilde{\mathcal{O}}(n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}})$.*

Algorithm 9 Compute $\mathsf{lca}_{\mathcal{Q}}(u,v)$ for all pairs of vertices $u,v \in V$.

Input: 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 $q \le 2n^{1-x}$.
Output: $\mathcal{Q}$-restricted LCA $\mathsf{lca}_{\mathcal{Q}}(u,v)$ for each pair of vertices $u,v \in V$.

```
 1: Initialize $\mathsf{lca}_{\mathcal{Q}}(\cdot,\cdot)$ with $-\infty$
 2: for $i=q,\ldots,1$ do
 3:     Initialize a $|S| \times n^x$ matrix $A$ with zeros
 4:     Let $\phi_i : Q_i \xrightarrow{1:1} \{1,\ldots,|Q_i|\}$ be an arbitrary
        bijection and $\phi_i^{-1}(\cdot)$ be its inverse function
 5:     for all $x \in |S|, y \in Q_i$ such that $(x,y) \in E$ do
 6:         $A[x,\phi_i(y)] \leftarrow 1$
 7:     end for
 8:     Compute $A \cdot A^T$, and its witness matrix $W$
 9:     for all $u,v \in S, u \ne v$ do
10:         if $\mathsf{lca}_{\mathcal{Q}}(u,v)=-\infty$ and $A \cdot A^T[u,v] \ne 0$ then
11:             $\mathsf{lca}_{\mathcal{Q}}(u,v) \leftarrow \phi_i^{-1}(W[u,v])$
12:         end if
13:     end for
14: end for
```

Proof. The proof of correctness follows from Lemma 4.1. We now show the proof for the running time. For each vertex v this can be done in $\mathcal{O}(n)$ time for all P_i by scanning all incoming edges from v, and in $\mathcal{O}(n^2)$ for all vertices in S and all $P_i, 1 \le i \le p$.

The $(\max,\min)$ matrix multiplication $A \circledvee A^T$, where A is an $n^\delta \times n^x$, can be performed in time $\widetilde{\mathcal{O}}(n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}})$, by Theorem 2.1. The time to compute the $(\max,\min)$ matrix product dominates the running time of Algorithm 5. ◻

Lemma 8.2. *Algorithm 9 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 $\widetilde{\mathcal{O}}(n^{1-x+\omega(\delta,x,\delta)})$.*

Proof. The proof of correctness follows from Lemma 8.2. Initializing the $n^\delta \times n^x$ dimensional matrix A for all $q=2n^{1-x}$ iterations take $\mathcal{O}(n^\delta n) \in \mathcal{O}(n^2)$ time. We apply n^{1-x} rectangular Boolean matrix multiplications $A \cdot A^T$ and witnesses. The running time of the algorithm is upper bounded by $\widetilde{\mathcal{O}}(\sum_{i=1}^{q} \mathrm{MM}(n^\delta,|Q_i|,n^\delta))$. Assume w.l.o.g. that $|Q_i|$ is non-increasing, then $|Q_i| \le n/i$, and

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

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

```
1: Compute the transitive closure graph $G = (V, E)$ of $G_{input}$
2: Use Algorithm 1 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 $|P_i| \leq n^{1-x}$ and a family of antichains $\mathcal{Q}' = \{Q'_1, \ldots, Q'_{q'}\}$ with $q' \leq 2n^{1-x}$
3: Use Lemma 4.2 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 and rename vertices so that they are $1, \ldots, n$ according to this order
5: Use Algorithm 5 to compute $\mathcal{P}$-restricted LCA $\mathsf{lca}_{\mathcal{P}}(u, v)$ for each pair of vertices $u, v \in S$
6: Use Algorithm 6 to compute $\mathcal{Q}$-restricted LCA $\mathsf{lca}_{\mathcal{Q}}(u, v)$ for each pair of vertices $u, v \in S$
7: for all $u, v \in S, u \neq v$ do
8:     $\mathsf{lca}(u, v) \leftarrow \max\{\mathsf{lca}_{\mathcal{Q}}(u, v), \mathsf{lca}_{\mathcal{P}}(u, v)\}$
9: end for
```

by monotonicity of $\mathrm{MM}(n^\delta, \cdot, n^\delta)$

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

□

Theorem 8.1. *Algorithm 10 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 4.2. The running time of the algorithm is trivially $\widetilde{\mathcal{O}}(n^\omega + n^{\frac{\omega(\delta,x,\delta)+2\delta+x}{2}} + n^{1-x+\omega(\delta,x,\delta)})$ 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 $\frac{2-2\delta}{\delta} + \omega(1, \frac{x}{\delta}, 1) = 3\frac{x}{\delta}$. (Here we are using $\omega(at, bt, ct) = t \cdot \omega(a, b, c)$ property which holds for any $a, b, c, t \geq 0$.) Using square matrix multiplication as a subroutine to implement rectangular matrix multiplication (i.e., the bound in (2.1)), one obtains $x = \frac{2}{5-\omega}$ and $\gamma' = 2\delta + \frac{4}{5-\omega} - 1 \leq 2\delta + 0.522571$. As usual, one can do better using more refined rectangular matrix multiplication algorithms. In particular, using the bound in (2.2), one gets $x = \frac{2-\beta\alpha\delta}{3-\beta}$, $\gamma' = \frac{2(2-\beta\alpha\delta)}{3-\beta} + 2\delta - 1 \leq 0.6282973594 + 1.861120612798\delta$. If instead we apply the bound (2.3), we get $\gamma' \leq 0.711508 + 1.73504\delta$ for $x = 0.855754 - 0.132478\delta$.

The running time becomes $\widetilde{\mathcal{O}}(n^\omega)$ for $|S| = n^\delta$ and: $\delta \leq \frac{\omega+1}{2} - \frac{2}{5-\omega} < 0.925146$ if bound (2.1) is used, $\delta \leq \frac{(\omega+1)(3-\beta)-4}{2(3-\beta)-2\beta\alpha} < 0.937374$ using the bound in (2.2) and $\delta \leq 0.957531$ if we apply the bound (2.3) (given current bounds on $\omega(1, \cdot, 1)$).

9 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 [25]. Therefore, it would be interesting to see if the techniques developed in this paper 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.

Acknowledgments The authors would like to thank Adam Polak for insightful discussions.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[40] 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.

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

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

[43] 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.

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

[45] 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.

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

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

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