Faster Algorithms for All Pairs Non-Decreasing Paths Problem

Ran Duan
Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
duanran@mail.tsinghua.edu.cn

Ce Jin
Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
jinc16@mails.tsinghua.edu.cn

Hongxun Wu
Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
wuhx18@mails.tsinghua.edu.cn

Abstract

In this paper, we present an improved algorithm for the All Pairs Non-decreasing Paths (APNP) problem on weighted simple digraphs, which has running time $\tilde{O}(n^{\frac{3+\omega}{2}}) = \tilde{O}(n^{2.686})$. Here n is the number of vertices, and $\omega < 2.373$ is the exponent of time complexity of fast matrix multiplication [Williams 2012, Le Gall 2014]. This matches the current best upper bound for $(\max, \min)$-matrix product [Duan, Pettie 2009] which is reducible to APNP. Thus, further improvement for APNP will imply a faster algorithm for $(\max, \min)$-matrix product. The previous best upper bound for APNP on weighted digraphs was $\tilde{O}(n^{\frac{1}{2}(3+\frac{3-\omega}{\omega+1}+\omega)}) = \tilde{O}(n^{2.78})$ [Duan, Gu, Zhang 2018]. We also show an $\tilde{O}(n^2)$ time algorithm for APNP in undirected simple graphs which also reaches optimal within logarithmic factors.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases graph optimization, matrix multiplication, non-decreasing paths

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.48

Category Track A: Algorithms, Complexity and Games

Related Version A full version of the paper is available at `https://arxiv.org/abs/1904.10701`.

1 Introduction

Given a directed or undirected graph $G = (V, E)$ with arbitrary real edge weights, a *non-decreasing path* is a path on which edge weights form a non-decreasing sequence [11]. We define the weight of a non-decreasing path to be the weight of its last edge, which we want to minimize. The motivation of this definition comes from train scheduling [13]. Suppose each train station is mapped to a vertex of a directed graph, and a train from station v_1 to station v_2 scheduled at time w is mapped to a directed edge (v_1, v_2) with weight w. If we neglect the time trains spent on their way, a trip from s to t is just a non-decreasing path from s to t in the constructed graph, and the earliest time arriving at t equals the minimum weight of such non-decreasing path. If we consider the train starts from v_1 at time w_1 and arrives at v_2 at time w_2, we can add a vertex u and two edges $(v_1, u), (u, v_2)$, then gives edge weights w_1, w_2 on them, respectively.

The *Single Source Non-decreasing Paths* (SSNP) problem, first studied by Minty [11] in 1958, is to find the minimum weight non-decreasing path from a given source s to all $t \in V$. The first linear time algorithm for SSNP problem in RAM model was given by Williams [13]. She also gave an $O(m \log \log n)$ time algorithm in the standard addition-comparison model. (m is the number of edges.)

46th International Colloquium on Automata, Languages, and Programming (ICALP 2019).
Editors: Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi;
Article No. 48; pp. 48:1–48:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The *All Pairs Non-decreasing Paths* (APNP) problem is the all-pairs version of the above problem, asking for the minimum weight non-decreasing path between *all* pairs of vertices in the graph. Williams [13] gave the first truly sub-cubic time algorithm of APNP. The original time complexity of the algorithm was $\tilde{O}(n^{(15+\omega)/6})$, obtained by using an $\tilde{O}(n^{2+\omega/3})$-time $(\min, \leq)$-matrix product[1] algorithm [12] as a subroutine. Here $\omega < 2.373$ is the exponent of time complexity of fast matrix multiplication [2, 14, 8]. Recently, Duan et al. [5] obtained a faster algorithm for APNP in $\tilde{O}(n^{\frac{1}{2}(3+\frac{3-\omega}{\omega+1}+\omega)}) = \tilde{O}(n^{2.78})$, by using a simple balancing technique introduced in [6]. We also adapt this technique in our algorithm.

Computing APNP is at least as hard as the *All Pair Bottleneck Paths* (APBP) problem [13], which asks for the maximum bottleneck path between every pair of vertices, where the bottleneck of a path is defined as the minimum weight (capacity) among all edges on this path [12, 15, 6, 9]. APBP is shown to have the same complexity as the $(\max, \min)$-matrix product ($C_{i,j} = \max_k\{\min(A_{i,k}, B_{k,j})\}$) [1, 12]. The current fastest algorithm for $(\max, \min)$-matrix product runs in $\tilde{O}(n^{(3+\omega)/2}) = \tilde{O}(n^{2.686})$ time [6]. Our algorithm for APNP matches this running time, so any further improvement on APNP will imply a faster algorithm for APBP and $(\max, \min)$-matrix product as well.

The *vertex-weighted* APNP problem on directed graphs, a restricted version of APNP, is computationally equivalent to the problem of *Maximum Witness for Boolean Matrix Multiplication* (MWBMM) [13]. An algorithm of $O(n^{2+\mu})$ time for MWBMM was given by Czumaj et al. [3], where μ satisfies the equation $\omega(1, \mu, 1) = 1 + 2\mu$ and $\omega(1, \mu, 1)$ is the exponent of multiplying an $n \times n^\mu$ matrix with an $n^\mu \times n$ matrix. Currently, the best bounds on rectangular matrix multiplication by Le Gall and Urrutia [7] imply that $\mu < 0.5286$.

1.1 Our results

In this paper we describe a faster algorithm for directed APNP problem running in $\tilde{O}(n^{(3+\omega)/2})$ time, which reaches optimal if the algorithm for APBP cannot be improved.

▶ **Theorem 1.** *The all pairs non-decreasing paths (APNP) problem on directed simple graphs can be solved in $\tilde{O}(n^{(3+\omega)/2})$ time.*

As in Dijkstra search [4] we can maintain a priority queue of current non-decreasing paths we have found, then the minimum unvisited one is the optimal paths between its endpoints. Every time we visit an optimal path, we "relax" all edges following that path. In [5] they partition the edge set into some parts by increasing order. For low-degree vertices in one part, trivially relax all of its outgoing edges, while for high-degree ones, use matrix multiplication to accelerate. Our algorithm adapt this idea, but we recursively divide the edges to $O(\log n)$ levels. In order the optimize the running time, a careful analysis of matrix multiplication is needed, and we need new ideas to use fast matrix multiplication to "predetermine" some of the useless edges from high-degree vertices.

We also give a $\tilde{O}(n^2)$ time algorithm for undirected APNP using the dynamic sequence data structure [10], which also reaches optimal within logarithmic factors.

▶ **Theorem 2.** *The all pairs non-decreasing paths (APNP) problem on undirected simple graphs can be solved in $\tilde{O}(n^2)$ time.*

[1] The $(\min, \leq)$-product of two matrices A, B is defined as $C_{i,j} = \min_k\{B_{k,j} : A_{i,k} \leq B_{k,j}\}$.

1.2 Organization of this paper

In Section 2, we introduce some terminologies used throughout this paper, and discuss how to recursively divide the edges. Then Section 3 and 4 will discuss our APNP algorithm for directed graphs in detail, where Section 3 explains the main techniques used, and Section 4 describes our whole algorithm and its analysis using procedures in Section 3 as subroutines. Due to page limit, the algorithm for undirected graphs will be discussed in the full version of this paper[2].

2 Preliminaries

2.1 Basic definitions

APNP problem

Let $G = (V, E)$ be a directed simple graph with edge weight $w(e)$ for each edge $e \in E$. We denote $n = |V|$ and $m = |E| = O(n^2)$.

A path is a sequence of edges $e_1, e_2, \ldots, e_l$. A non-decreasing path is a path satisfying $\forall 1 \leq i \leq l-1$, $w(e_i) \leq w(e_{i+1})$, and the weight of this non-decreasing path is defined to be $w(e_l)$, the weight of the last edge. All pairs non-decreasing paths problem asks to determine the minimum weight non-decreasing path between every pair of vertices. Let $OPT(i,j)$ denote the optimal (minimum) non-decreasing path between i and j. In contrast, during the algorithm, we use $d(i,j)$ to denote the current minimum non-decreasing path that our algorithm has found so far. In our algorithm, instead of explicitly maintaining the paths, we only need to maintain the weights of the paths $d(i,j)$ and $OPT(i,j)$ (and their last edges if we want to retrieve the paths), so we will also use $d(i,j)$ and $OPT(i,j)$ to denote the weights of corresponding paths $d(i,j)$ and $OPT(i,j)$, respectively. (Remember that $w(j,k)$ denotes the weight of the edge (j,k).)

Notation for String and Subgraph

Without loss of generality[3], we assume all edges have distinct edge weights ranged from 0 to $2^b - 1$, where $b = \lceil \log_2 |E| \rceil \leq 2\log_2 n + O(1)$, so $2^b = O(n^2)$. Then every edge weight $w(e)$ corresponds to a b-bit 0-1 string $[w(e)]$ which is its binary representation, with the rightmost bit being the lowest bit.

To distinguish between values and strings, string s is written as $[s]$. $LCP([w_1],[w_2])$ (which is also a binary string) is the longest common prefix of $[w_1]$ and $[w_2]$. For example $LCP([100],[101]) = [10]$, and $LCP([000],[100]) = [\,]$(empty string). $|[w]|$ denotes the length of the string $[w]$, and $[x][y]$ denotes the concatenation of two strings $[x]$ and $[y]$. $[x] < [y]$ means that the lexicographical order of $[x]$ is smaller. For example, $[0111] < [101] < [1010]$.

We define $E_{[x]} = \{e \in E \mid [w(e)] \text{ has prefix } [x]\}$ to be the set of all edges whose weight has prefix $[x]$ (also call the edges have prefix $[x]$), and similarly the subgraph $G_{[x]} = (V, E_{[x]})$. For convenience, an edge set or a subgraph with subscript $[x]$ means all of the edges in it have prefix $[x]$.

2 See `https://arxiv.org/abs/1904.10701`.
3 See the full version of this paper for the proof.

Rectangular Matrix Multiplication

We use $M(m,n,p)$ to denote the asymptotic time complexity of multiplying an $m \times n$ matrix with an $n \times p$ matrix. We denote $M(n,n,n) = O(n^\omega)$, where $\omega < 2.373$ [2, 14, 8]. In this paper, rectangular matrix multiplications are straightforwardly reduced to square matrix multiplications. So we will only use the following fact. (Here $\min(x_1, \cdots, x_k)$ means the minimum number in $\{x_1, \cdots, x_k\}$.)

▶ **Lemma 3.** $M(m,n,p) = O\left(\dfrac{mnp}{\min(m,n,p)^{3-\omega}}\right)$.

Proof. The rectangular matrices are decomposed into $\min(m,n,p) \times \min(m,n,p)$ square matrices, then standard fast square matrix multiplication is applied. ◀

2.2 Naïve Algorithm

Let us first take a look at the naïve algorithm for APNP, a simple Dijkstra-type search [4]. Initially set $d(i,j) = w(i,j)$ for all edges $(i,j) \in E$, and $d(i,j) = +\infty$ otherwise. Each time we visit the minimum unvisited $d(i,j)$ and enumerate every out-going edge (j,k) of vertex j. If $d(i,j)$ and (j,k) form a nondecreasing path, then we update $d(i,k) \leftarrow \min(d(i,k), w(j,k))$. (See Algorithm 1.) We refer to this update step as *relaxing* edge (j,k) w.r.t. $d(i,j)$. By the greedy nature of Dijkstra search, when we visit $d(i,j)$, $d(i,j) = OPT(i,j)$. Namely, it is optimal.

Algorithm 1 Relaxing edges naïvely.

```
while there exists unvisited d(i,j) do
    visit the minimum unvisited d(i,j)
    for every (j,k) such that w(j,k) > d(i,j) do
        perform update d(i,k) ← min(d(i,k), w(j,k))
```

For clarity, our usages of symbols i, j, k stick to the following convention: $d(i,k)$ refers to the path being updated by the concatenation of path $d(i,j)$ and edge (j,k).

The naïve algorithm takes $O(n^3)$ time when edge weights are integers from 0 to $2^b - 1$, since a bucket heap is enough to maintain unvisited $d(i,j)$.

2.3 Classifying edges according to degrees

In order to avoid relaxing all edges when visiting $d(i,j)$, our algorithm will classify the edges based on the degrees of their two endpoints, and partition the edges whose both endpoints have high degrees into two sets by the order of their weights, then recursively deal with these two sets. We relax different types of edges by different approaches. First, we define “high-degree” and “low-degree” vertices in a subgraph.

High degree and low degree

In this paper, sometimes we use a subset of edges $E' \subseteq E$ to denote the subgraph $G' = (V', E')$ where V' is the set of vertices associated with E', namely, vertices in E' refers to vertices in V'. In a subgraph $G' = (V', E')$ of G, a vertex has high outdegree if its outdegree is larger than n^{1-t}, and otherwise, it has low outdegree. Here t is a parameter to be determined later (we will choose $t = \frac{3-\omega}{2}$). Similarly, a vertex has high indegree if its indegree is larger than n^{1-t}, and otherwise, it has low indegree. In our algorithm, edges (j,k) in a subgraph are divided into three types based on the outdegree of j and indegree of k:

- Low edge: if j has low outdegree
- High-high edge: if j has high outdegree and k has high indegree
- High-low edge: if j has high outdegree and k has low indegree

Divide the edges

In this binary partition procedure, starting from the entire edge set $E'_{[\,]} = E$, each time we divide the set of high-high edges $H_{[x]}$ in $E'_{[x]}$ into two parts: $H'_{[x][0]}$ and $H'_{[x][1]}$, based on its next bit after prefix $[x]$, then recursively partition the edge sets $H'_{[x][0]}$ and $H'_{[x][1]}$. Here we use $L_{[x]}$ and $\Gamma_{[x]}$ to denote the low edges and high-low edges with prefix $[x]$ obtained in the algorithm.

Algorithm 2 Divide the edges.

procedure $\textsc{Divide}(E'_{[x]})$
  Consider the indegrees and outdegrees of vertices in the graph $G'_{[x]} = (V, E'_{[x]})$;
  Let $L_{[x]}$ be the set of edges from low outdegree vertices;
  Let $H_{[x]}$ be the set of edges from high outdegree vertices to high indegree vertices;
  Let $\Gamma_{[x]}$ be the set of edges from high outdegree vertices to low indegree vertices;
  Let $H'_{[x][0]} = \{(j,k) \in H_{[x]}$ s.t. $[w(j,k)]$ has prefix $[x][0]\}$ and $H'_{[x][1]} = \{(j,k) \in H_{[x]}$ s.t. $[w(j,k)]$ has prefix $[x][1]\}$.
  $\textsc{Divide}(H'_{[x][0]})$
  $\textsc{Divide}(H'_{[x][1]})$
end procedure

At the beginning we call $\textsc{Divide}(E)$. Since only the edges in $H_{[x]}$ go into the next recursion, every edge can only appear in one of $L_{[x]}$ or $\Gamma_{[x]}$ but can appear in many $H_{[x]}$.

2.4 Outline of our algorithm

In our algorithm, we run a Dijkstra-type procedure. When visiting $d(i,j)$, which is guaranteed to be optimal, we relax all the edges (j,k) in $L_{[x]}$ and $\Gamma_{[x]}$ w.r.t. $d(i,j)$ for all $[x]$ which is a prefix of $[d(i,j)]$. The outdegree of j is small in $L_{[x]}$ but not in $\Gamma_{[x]}$, so to relax $L_{[x]}$, we relax all edges from j as the naïve algorithm. But to relax $\Gamma_{[x]}$, a preprocessing procedure of the edges in $\Gamma_{[x]}$ is needed in order to save time. For edges in $H_{[x]}$, instead of immediately relaxing them, we wait until all optimal paths with prefix $[x][0]$ are visited, then perform a $(\min, \leq)$-matrix product of these paths with the adjacency matrix of $H'_{[x][1]}$ to relax all edges in it. Details of these methods will be given in Section 3.

3 Basic techniques

In this section, we explain the method we use for relaxing edges. Suppose we are trying to relax edges (j,k) w.r.t. $d(i,j)$, it will be based on whether (j,k) is in $L_{[x]}$, $H_{[x]}$, or $\Gamma_{[x]}$.

As explained before, $L_{[x]}$ is easiest. Since j has low outdegree, like the naïve algorithm, we simply relax all of its outgoing edges, which takes $O(n^{1-t})$ time. $H_{[x]}$ and $\Gamma_{[x]}$ are handled by different methods, though they both use the "row/column balancing" technique of matrix proposed in [6]. Instead of considering it as splitting rows/columns on matrix, we describe the balancing technique as splitting vertices so that every vertex has low outdegree/indegree. In the following subsections we describe how to handle $H_{[x]}$ and $\Gamma_{[x]}$.

Table 1 Balancing the indegrees of vertices k.

Balancing the indegrees of vertices
(S1) For every k, sort all the in-coming edges of k in increasing order and obtain the sorted list L_k.
(S2) Split k into vertices $k'_1, k'_2, \cdots k'_p$, and divide L_k into segments each of size n^{1-t} (while the last segment is possibly incomplete). The edges in the r-th segment is assigned to vertex k'_r. Namely, edge (j,k) in the r-th segment now becomes edge (j,k'_r).
(S3) Let $[L(k'_r), R(k'_r)]$ denote the weight range of in-coming edges of k'_r. Namely $L(k'_r)$ is the minimum weight of in-coming edges of k'_r, and $R(k'_r)$ is the maximum. Obviously, $L(k'_1) \le R(k'_1) < L(k'_2) \le R(k'_2) < \cdots < L(k'_p) \le R(k'_p)$.

3.1 Balancing

A graph $G=(V,E)$ contains at most $\frac{|E|}{n^{1-t}}$ vertices with high indegrees or outdegrees. If there are $\Theta(\frac{|E|}{n^{1-t}})$ number of vertices with high degrees, we would expect an average degree of $O(n^{1-t})$. However, the degrees of some vertices may be far greater than n^{1-t}. To balance the indegree (outdegree) of each vertex, we split every vertex into several vertices each of indegree (outdegree) n^{1-t} and one vertex of indegree (outdegree) $\le n^{1-t}$. The number of new vertices with indegree (outdegree) exactly n^{1-t} is bounded by $O\left(\frac{|E|}{n^{1-t}}\right)$, and every original high indegree (outdegree) vertex corresponds to at most one new vertex with indegree (outdegree) $< n^{1-t}$, thus we have at most $O\left(\frac{|E|}{n^{1-t}}\right)$ many new vertices each with indegree (outdegree) $\le n^{1-t}$.

In our algorithm, for edge set $\{(j,k)\}$, we use this technique to either balance the outdegrees of vertices j or the indegrees of vertices k. Here we demonstrate this technique for balancing indegrees of k as an example in Table 1. Denote the set of edges after balancing to be $\bar{E}$, then the graph corresponding to $\bar{E}$ is actually a bipartite graph with edges between vertices j and k'_r. The procedure for balancing outdegrees of j is symmetric.

3.2 High-high edges

The technique for high-high edges solves the following problem: Given a set P of optimal paths of prefix $[x][0]$ and a set $H'_{[x][1]}$ of high-high edges, the problem asks to relax all edges in $H'_{[x][1]}$ w.r.t. paths in P, namely, extend paths in P by a single edge in $H'_{[x][1]}$.

This problem is equivalent with a length-two nondecreasing path problem. As discussed in Section 1, this can be solved by fast $(\min, \le)$-product implied in [6]. But the rectangular version is not covered by [6], so we fully describe the algorithm and its analysis (in graphs).

We have two extra guarantees when we use this procedure in our main algorithm:

- P is the set of optimal paths $OPT(i,j)$ such that $[OPT(i,j)]$ has prefix $[x][0]$, so any path $d(i,j)$ in P can form a nondecreasing path with any edge (j,k) in $H'_{[x][1]}$.
- All $d(i,j)$ satisfying $[d(i,j)] < [x][1]$ are already visited by Dijkstra search. So we can tell whether $[OPT(i,j)] < [x][1]$ or not.

Suppose there are $n_{[x][1]}$ many $[OPT(i,j)]$ which have $[x][1]$ as a prefix. In our algorithm the time complexity will depend on $n_{[x][1]}$.

The first step is to apply the balancing technique in Section 3.1 to indegrees of vertices k in $H'_{[x][1]}$, and let the edge set after balancing be $\bar{H}'_{[x][1]}$. Here each high indegree vertex

k is split into vertices $\{k'_r\}$. We define the following two matrices: (j and k'_r only include vertices having out-going edges and in-coming edges in $\bar{H}'_{[x][1]}$, respectively.)

$$A_{i,j} = \begin{cases} 1 & d(i,j) \in P \\ 0 & \text{otherwise} \end{cases} \qquad B_{j,k'_r} = \begin{cases} 1 & (j,k'_r) \in \bar{H}'_{[x][1]} \\ 0 & \text{otherwise} \end{cases}$$

Then we multiply them with rectangular matrix multiplication. Let $C = AB$, then,

$$C_{i,k'_r} \begin{cases} > 0 & \text{if there is a nondecreasing path from } i \text{ to } k'_r \\ = 0 & \text{otherwise} \end{cases}$$

For every pair (i,k) such that $[OPT(i,k)] \geq [x][1]$ $(d(i,j) \notin P)$ and k is an in-coming vertex in $H'_[x][1]$, we find the minimum r with $C_{i,k'_r} > 0$ and relax all n^{1-t} incoming edges (j,k'_r) of k'_r w.r.t. $d(i,j)$ if $d(i,j) \in P$. We can skip other $r' > r$ because $OPT(i,k) \leq R(k'_r) < L(k'_{r'})$. Namely, we only have to relax $O(n^{1-t})$ many edges for each (i,k), which is the benefit of balancing. We attribute this $O(n^{1-t})$ cost of relaxations to the $OPT(i,k)$ with prefix $[x][1]$ found in these relaxations, which must exist if $C_{i,k'_r} > 0$.

The procedure above consists of three parts: finding minimum r for each (i,k), relaxation and matrix multiplication. The time cost for the first part is at most enumerating all nonzero elements of C, so it is dominated by the matrix multiplication part.

▶ **Lemma 4.** *The relaxation part takes $O\left(n_{[x][1]}n^{1-t}\right)$ time in total.*

Proof. The cost of relaxation is attributed to each $OPT(i,k)$ with prefix $[x][1]$. Since there are $n_{[x][1]}$ many $OPT(i,k)$ with prefix $[x][1]$, and each corresponds to $O(n^{1-t})$ relaxations, it costs $O(n_{[x][1]}n^{1-t})$ time in total. ◀

▶ **Lemma 5.** *In the matrix multiplication part, A is an $n \times O\left(\min\left(n, \frac{2^{b-|[x]|}}{n^{1-t}}\right)\right)$ matrix, and B is a $O\left(\min\left(n, \frac{2^{b-|[x]|}}{n^{1-t}}\right)\right) \times O\left(\frac{2^{b-|[x]|}}{n^{1-t}}\right)$ matrix, so the time complexity for matrix multiplication is $M\left(n, \min\left(n, \frac{2^{b-|[x]|}}{n^{1-t}}\right), \frac{2^{b-|[x]|}}{n^{1-t}}\right)$.*

Proof. There are at most $\min\left(n, \frac{|H'_{[x][1]}|}{n^{1-t}}\right)$ many j because vertices j have high outdegrees. After balancing, there are at most $O\left(\frac{|H'_{[x][1]}|}{n^{1-t}}\right)$ many k'_r by discussion in Section 3.1. Since each edge in $H'_{[x][1]}$ has prefix $[x][1]$, $|H'_{[x][1]}| \leq 2^{b-|[x]|-1} = O\left(2^{b-|[x]|}\right)$. Plug in the size of $H'_{[x][1]}$ gives the desired bound. ◀

3.3 High-low edges

Now we consider the relaxation of the high-low edges in $\Gamma_{[x]}$ when visiting paths with the same prefix $[x]$. To preprocess high-low edges, we run an initialization step when all optimal paths less than $[x]$ have been visited. As before, we denote the number of $[OPT(i,j)]$ with prefix $[x]$ by $n_{[x]}$.

Since the outdegrees of vertices j are high, we cannot relax edges one by one. But we still want to utilize the property that the indegrees of k are low. As in the last subsection, we denote the set of optimal paths we have found to be P, namely when we visit $d(i,j)$, we add $d(i,j)$ to P. By the nature of Dijkstra search, such $d(i,j)$ is always visited in increasing order. At the initialization step, P is the set of all optimal paths less than $[x]$. During the procedure, optimal paths with prefix $[x]$ are added to P.

We also maintain a dynamic set Q which is initially empty. When we relax an edge (j,k) w.r.t. $d(i,j)$, we put $d(i,k)$ into Q only if $[d(i,k)] \geq [x]$, that is, $d(i,k)$ was not visited at initialization. So Q contains new nondecreasing path but not guaranteed to be optimal.

Initialization by Matrix Multiplication

The first step is to apply the balancing technique to the outdegrees of vertices j in $\Gamma_{[x]}$ such that every vertex j in $\Gamma_{[x]}$ is split into a sequence of vertices $\{j'_r\}$. Suppose the edge set after balancing is $\bar{\Gamma}_{[x]}$. Then, we define the following two matrices: (j'_r and k only include vertices having out-going edges and in-coming edges in $\bar{\Gamma}_{[x]}$, respectively.)

$$A_{i,k} = \begin{cases} 1 & d(i,k) \notin P \cup Q \\ 0 & \text{otherwise} \end{cases} \qquad B_{k,j'_r} = \begin{cases} 1 & (j'_r, k) \in \bar{\Gamma}_{[x]} \\ 0 & \text{otherwise} \end{cases}$$

We compute $C = AB$. Basically speaking, the matrix C can indicate whether we need to relax edges (j'_r, k) from j'_r when visiting $d(i,j)$. Note that when we run initialization, Q is empty, so $d(i,k) \notin Q$ is trivially true. But we will add paths to Q and dynamically update the matrix multiplication later.

We make the following observation about C_{i,j'_r}.

▶ **Observation 1.** *If $d(i,j) < L(j'_r)$ and $C_{i,j'_r} > 0$, there is at least one edge (j'_r, k) such that $d(i,k) \notin P \cup Q$ and $[x]$ is a prefix of $[OPT(i,k)]$.*
Conversely, if $C_{i,j'_r} = 0$, there is no such an edge (j'_r, k).

Proof. Since $C_{i,j'_r} > 0$, at least one vertex k satisfy the following:
- $A_{i,k} > 0 : d(i,k) \notin P$, so $[OPT(i,k)] \geq [x][0 \cdots 0]$. Also $d(i,k) \notin Q$.
- $B_{k,j'_r} > 0 : (j'_r, k) \in \bar{\Gamma}_{[x]}$.

Since $d(i,j) < L(j'_r)$, $d(i,j)$ and the original edge of (j'_r, k) form a non-decreasing path. So $[OPT(i,k)] \leq [x][1 \cdots 1]$.

Conversely, if $C_{i,j'_r} = 0$, for every k, at least one of these happens:
- $A_{i,k} = 0 : d(i,k) \in P \cup Q$
- $B_{k,j'_r} = 0 :$ edge (j'_r, k) does not exist in $\bar{\Gamma}_{[x]}$. ◀

Update matrix multiplication

When a new path $d(i,k)$ is added to P or Q, the matrix A and product C need to be updated. Adding a new path to P or Q only changes one entry of $A_{i,k}$, so we utilize the low indegree of k. There are at most $O(n^{1-t})$ many j'_r such that $B_{k,j'_r} \neq 0$, since $\Gamma_{[x]}$ contains high-low edges only. So the update of C when changing one element $A_{i,k}$ takes only $O(n^{1-t})$ time by enumerating all nonzero B_{k,j'_r}. This cost can be attributed to each $[OPT(i,k)]$ with prefix $[x]$, since every such path can only be added to $P \cup Q$ once. However, adding $d(i,k)$ to Q does not mean we have found the optimal path $OPT(i,k)$, as it can still be updated. How to deal with this will be discussed later.

Relaxation when visiting $d(i,j)$

When we visit $d(i,j)$, for each split vertex j'_r of j, there are three cases:
1. $d(i,j) > R(j'_r) : d(i,j)$ cannot form a non-decreasing path with any out-going edge of j'_r, so we skip j'_r.
2. $d(i,j) \in [L(j'_r), R(j'_r)] :$ We relax all out-going edges of j'_r larger than $d(i,j)$.
3. $d(i,j) < L(j'_r) :$ Only when $C_{i,j'_r} > 0$, we relax all out-going edges of j'_r one by one.

By Observation 1, when $C_{i,j'_r} = 0$, for each edge (j'_r, k), either $d(i,k) \in P$ or $d(i,k) \in Q$. If $d(i,k) \in P$, it needs no more update. If $d(i,k) \in Q$, roughly speaking, since k has indegree less than n^{1-t}, the updates for $d(i,k)$ can be done "in advance" when it is added to Q. The details will be clear later. This is why we can skip j'_r when $C_{i,j'_r} = 0$.

Since the degree of j'_r is bounded by n^{1-t}, the relaxation takes $O(n^{1-t})$ time for each j'_r. For the second case, it only happens once for each $d(i,j)$ because $[L(j'_r), R(j'_r)]$ are disjoint for different j'_r, so the $O(n^{1-t})$ cost is attributed to $d(i,j)$. For the third case, by Observation 1, there is at least one edge (j'_r, k) such that $OPT(i,k) \notin P \cup Q$ with prefix $[x]$, so the $O(n^{1-t})$ time is attributed to $d(i,k)$. (If there is more than one such k, choose an arbitrary one.) Then $d(i,k)$ is added to Q, and the cost of updating A and C is also bounded by n^{1-t}, dominated by the cost of relaxation.

Because the first path $d(i,k)$ we found for each (i,k) is not necessarily the optimal one, we discuss how to handle all future updates of $d(i,k)$ "in advance" after adding it to Q. We enumerate every in-coming edge $(j'', k) \in \Gamma_{[x]}$ of k. If $d(i,j'')$ is not in P, we add (i,k) to a *waiting list* for (i,j''), denoted by $W(i,j'')$. When $d(i,j)$ is visited in the future, we can go through its waiting list $W(i,j)$ and update $d(i,k)$ for all pair (i,k) in the list. There are only n^{1-t} in-coming edges for k, so the waiting list construction cost is also $O(n^{1-t})$ for every $d(i,k)$.

In conclusion, we follow the procedure in Algorithm 3 when visiting $d(i,j)$ with prefix $[x]$.

Algorithm 3 High-low relaxation when visiting $d(i,j)$.

1: Add $d(i,j)$ to P and update A and $C = AB$.
2: **for** (i,k) in the *waiting list* $W(i,j)$ **do**
3:     Relax (j,k) w.r.t. $d(i,j)$ if $w(j,k) > d(i,j)$
4: **for** every j'_r satisfying $d(i,j) \in [L(j'_r), R(k'_r)]$ or $(d(i,j) < L(j'_r)$ and $C_{i,j'_r} > 0)$ **do**
5:     **for** every outgoing edge (j'_r, k) of j'_r larger than $d(i,j)$ **do**
6:         **if** $d(i,k) \notin P \cup Q$ **then**
7:             Relax (the original edge of) (j'_r, k) w.r.t. $d(i,j)$
8:             Add $d(i,k)$ to Q and update A and $C = AB$
9:             **for** incoming edge (j'', k) of k **do**
10:                 Add (i,k) to the *waiting list* $W(i,j'')$ if $d(i,j'') \notin P$

Complexity

This procedure is divided into matrix multiplication part (initialization) and relaxation part (Algorithm 3) as well.

▶ **Lemma 6.** *The relaxation part takes* $O\left(n_{[x]} n^{1-t}\right)$ *time.*

Proof. From discussion above, the $O(n^{1-t})$ cost of each relaxation is either attributed to optimal $d(i,j)$ with prefix $[x]$ or $d(i,k) \in Q$. For each $d(i,k) \in Q$, $OPT(i,k)$ is of course larger than $[x][0 \cdots 0]$, and then relaxed by an edge $\leq [x][1 \cdots 1]$, so the size of Q is also bounded by $n_{[x]}$. Since each $d(i,j)$ can only be added to P and Q once, respectively, the total time is $O\left(n_{[x]} n^{1-t}\right)$.

The total size of all waiting lists $W(i,j)$ is bounded by $O\left(n_{[x]} n^{1-t}\right)$ as well, because each time when an $d(i,k)$ is added to Q, we enumerate $\leq n^{1-t}$ many incoming edges of k, and add (i,k) to waiting list at Line 10. Every waiting list can only be relaxed once, thus, the relaxation of waiting list edges in Line 3 needs $O\left(n_{[x]} n^{1-t}\right)$ in total. ◀

The following lemma is crucial. Although edges (j,k) in $\Gamma_{[x]}$ are high-low edges, the number of k is not directly bounded, but remind that in our binary partition of edges, only high-high edges of previous level can be in this set, thus in fact the number of k *cannot* be asymptotically larger than the number of j'_r.

▶ **Lemma 7.** *In the matrix multiplication part, A is an $n \times O\left(\min\left(n, \frac{2^{b-|[x]|}}{n^{1-t}}\right)\right)$ matrix, and B is an $O\left(\min\left(n, \frac{2^{b-|[x]|}}{n^{1-t}}\right)\right) \times O\left(\frac{2^{b-|[x]|}}{n^{1-t}}\right)$ matrix, so the time complexity of matrix multiplication is $M\left(n, \min\left(n, \frac{2^{b-|[x]|}}{n^{1-t}}\right), \frac{2^{b-|[x]|}}{n^{1-t}}\right)$.*

Proof. Since $|\Gamma_{[x]}| \le 2^{b-|[x]|}$ (because each edge in it has prefix $[x]$), after balancing, there are at most $O\left(\frac{2^{b-|[x]|}}{n^{1-t}}\right)$ many j'_r.

If $[x] \neq []$, suppose $[x] = [x'][0/1]$, namely $[x']$ is the prefix of $[x]$ which is one bit shorter. If $(j,k) \in \Gamma_{[x]}$, by Algorithm 2, $(j,k) \in H_{[x']}$. Since k has high indegree in $H_{[x']}$, the number of such k is bounded by $O\left(\min\left(n, \frac{|H_{[x']}|}{n^{1-t}}\right)\right)$. Also $|H_{[x']}| = O\left(2^{b-|[x]|}\right)$. Plug it in gives the $O\left(\min\left(n, \frac{2^{b-|[x]|}}{n^{1-t}}\right)\right)$ bound for the number of k. If $[x] = []$, of course the number of k is bounded by n. ◀

4 Main algorithm for directed graphs and analysis

4.1 Main algorithm

Just like in the naïve algorithm, we use a bucket to maintain all $d(i,j)$ we have found, and the minimal *unvisited* $d(i,j)$ is guaranteed to be optimal. Our algorithm enumerates the value x from 0 to $2^b - 1$ and visit $d(i,j)$ if $d(i,j) = x$. We carefully combine techniques introduced in the previous section into this framework.

Recall that in Algorithm 2 of Section 2.3 we define $L_{[x]}$ to be the set of low edges (from low outdegree vertices) and $\Gamma_{[x]}$ to be the set of high-low edges in $H'_{[x]}$, which are high-high edges in higher level, then divide the edge set $H_{[x]}$ of high-high edges in $H'_{[x]}$ to $H'_{[x][0]}$ and $H'_{[x][1]}$ and recursively deal with them. Our main algorithm is presented in Algorithm 4.

Algorithm 4 Main algorithm.

$d(i,j) = w(i,j)$ for all edges $(i,j) \in E$, and $d(i,j) = +\infty$ otherwise
for x from 0 to $2^b - 1$ **do**
  for all prefix $[y]$ of $[x]$ **do**
    if $[x] = [y][000\cdots0]$ **then**
      Run high-low edge initialization for edges in $\Gamma_{[y]}$
    if $[x] = [y][100\cdots0]$ **then**
      $P_{[y][0]} = \{d(i,j) \mid [y][0]$ is a prefix of $[d(i,j)]\}$
      Append high-high edges in $H'_{[y][1]}$ to paths in $P_{[y][0]}$
  for $d(i,j) = x$ **do**
    Mark $d(i,j)$ as visited, add $d(i,j)$ to P.
    for all prefix $[y]$ of $[d(i,j)]$ **do**
      Relax edges $(j,k) \in L_{[y]}$
      Relax edges $(j,k) \in \Gamma_{[y]}$ by Algorithm 3

When visiting an optimal path $d(i,j)$, we need to relax all edges (j,k) which are larger than $d(i,j)$.

▶ **Observation 2.** *For $[d(i,j)] = [x]$, every edge (j,k) larger than $d(i,j)$ must be one of the following three cases: (so $[y]$ is a prefix of both $[x]$ and $[w(j,k)]$.)*
- *$(j,k) \in L_{[y]}$ for some prefix $[y]$ of $[x]$*
- *$(j,k) \in \Gamma_{[y]}$ for some prefix $[y]$ of $[x]$*
- *$(j,k) \in H'_{[y][1]}$ where $[y][0]$ is a prefix of $[x]$*

Proof. Consider the longest common prefix $[y] = LCP([x], [w(j,k)])$. If (j,k) is not in the $L_{[y']}$ or $\Gamma_{[y']}$ for any prefix $[y']$ of $[y]$, it must be in $H_{[y]}$. Since $[y]$ is the longest common prefix and $w(j,k)$ is larger than $d(i,j)$, $[d(i,j)]$ has prefix $[y][0]$ and $[w(j,k)]$ has prefix $[y][1]$, thus (j,k) is in $H'_{[y][1]}$. ◀

Thus, we can simply relax the edges of the first type, and use the method in Section 3.2 to relax the edges in $H'_{[y][1]}$ when all of the optimal paths with prefix $[y][0]$ have been visited. The method for high-low edges $\Gamma_{[y]}$ is like a dynamic data structure: we initialize it when $[x] = [y][0 \cdots 0]$, and update it when relaxing an edge in $\Gamma_{[y]}$.

High-high edges

For high-high edges, when $[x] = [y][100 \cdots 0]$, before those $d(i,j) = x$ are visited, we append edges in $H'_{[y][1]}$ to paths in $P_{[y][0]} = \{d(i,j) \mid [y][0]$ is a prefix of $[d(i,j)]\}$ using the technique introduced in Section 3.2. See Line 8, Algorithm 4.

In Section 3.2, we have two guarantees. Now we check them one by one:

- Because each edge in $H'_{[y][1]}$ has prefix $[y][1]$, and each path in $P_{[y][0]}$ has prefix $[y][0]$, the maximum weight in $P_{[y][0]}$ is smaller than the minimum weight of $H'_{[y][1]}$. At the time of $[x] = [y][100 \cdots 0]$, all paths in $P_{[y][0]}$ are optimal.
- Since $[x] = [y][100 \cdots 0]$, all $[d(i,j)] < [y][1]$ are visited, and none of $[d(i,j)] \geq [y][1]$ are visited yet.

High-low edges

We initialize for $\Gamma_{[y]}$ when $[x] = [y][000 \cdots 0]$ before we visit those $d(i,j) = x$. See Line 5 of Algorithm 4. All $[d(i,j)] < [y]$ are visited, and none of $[d(i,j)] \geq [y]$ are visited. Once a $d(i,j)$ within the range $[y][000 \cdots 0] \sim [y][111 \cdots 1]$ is visited, we use the approach in Algorithm 3.

4.2 Correctness

We now prove the correctness of our algorithm. Suppose the last edge of $OPT(i,k)$ is (j,k). Then $d(i,k)$ is correctly computed before it is visited if and only if the following two conditions holds:

- If $i \neq j$, $d(i,j)$ is correctly computed before it is visited.
- After $d(i,j)$ is visited, before we visit $d(i,k)$, $d(i,k)$ is updated by relaxing the edge (j,k) w.r.t. $d(i,j)$.

We prove the second condition holds for every $d(i,j)$, and the first one simply follows from induction.

Suppose $[z] = LCP([OPT(i,j)], [OPT(i,k)]) = LCP([OPT(i,j)], [w(j,k)])$. By Observation 2 the last edge (j,k) must be in one of the three cases, so we check them one by one.

▶ **Lemma 8.** *If (j,k) is in $L_{[y]}$ for some prefix $[y]$ of $[z]$, and $d(i,j)$ is correctly computed before visited, then $d(i,k)$ is also correctly computed before visited.*

Proof. Because $[y]$ is also a prefix of $[OPT(i,j)]$, at Line 12, when we visit $d(i,j)$, $d(i,k)$ is updated by relaxing (j,k). Because $OPT(i,j) < OPT(i,k)$, $d(i,k)$ is not visited yet. ◀

▶ **Lemma 9.** *Suppose (j,k) is in $H'_{[y][1]}$ where $[y][0]$ is a prefix of $[OPT(i,j)]$. If $d(i,j)$ is correctly computed before visited, $d(i,k)$ is also correctly computed before visited.*

Proof. We can see $[y] = [z]$ from the proof of Observation 2. At Line 8, when $[x] = [y][100\cdots 0]$, $d(i,k)$ is updated by $d(i,j)$ and (j,k). $d(i,j)$ is already visited before because it has prefix $[y][0]$. $d(i,k)$ will be visited later because it has prefix $[y][1]$. ◀

▶ **Lemma 10.** *If (j,k) is in $\Gamma_{[y]}$ for some prefix $[y]$ of $[z]$, and $d(i,j)$ is correctly computed before visited, then $d(i,k)$ is also correctly computed before visited.*

Proof. Since $[y]$ is a prefix of both $[OPT(i,j)]$ and $[w(j,k)]$, the initialization for $\Gamma_{[y]}$ is done at Line 5 when $[x] = [y][000\cdots 0]$. After that, $d(i,k)$ is updated when we visit $d(i,j)$ at Line 13. $d(i,k)$ is not visited yet because $OPT(i,k) > OPT(i,j)$. ◀

▶ **Lemma 11.** *All $d(i,j)$ are correctly computed before visited.*

Proof. This follows from a simple induction. In the base case, for all length 1 optimal paths $OPT(i,j)$, they are obviously correctly computed in Line 1. Then if all length $l-1$ paths $OPT(i,j)$ are correctly computed before visited, by Lemma 8, 9, 10, all length l paths $OPT(i,j)$ are also correctly computed before visited. ◀

4.3 Running time

▶ **Lemma 12.** *The relaxation for low edges ($L_{[x]}$) takes $\tilde{O}\left(n^{3-t}\right)$ time in total. (Line 12)*

Proof. At Line 12, we only enumerate $O(n^{1-t})$ many edges because j has low outdegree in $L_{[y]}$. Since there are only $b = O(\log n)$ many prefix $[y]$ for each $[d(i,j)]$, each $d(i,j)$ takes $\tilde{O}(n^{1-t})$ time. So in total, these updates take $\tilde{O}(n^{3-t})$ time for all $O(n^2)$ many $d(i,j)$. ◀

▶ **Lemma 13.** *The relaxation for high-high edges and high-low edges besides matrix multiplication takes $\tilde{O}\left(n^{3-t}\right)$ time in total.*

Proof. By Lemma 4 and Lemma 6, for each $[y]$, the complexity for relaxation is bounded by $(n_{[y]} + n_{[y][1]})n^{1-t} = O\left(n_{[y]}n^{1-t}\right)$, where $n_{[y]}$ stands for the number of optimal paths with prefix $[y]$. Since an optimal path can be counted in $O(\log n)$ many $n_{[y]}$, the total time is therefore $\tilde{O}(n^{3-t})$. ◀

▶ **Lemma 14.** *The matrix multiplication parts for high-high edge updates and the initialization of high-low edge updates take $\tilde{O}\left(n^{t+\omega}\right)$ time in total.*

Proof. By Lemma 5 and Lemma 7, the complexity for matrix multiplication is at most $M\left(n, \min\left(\frac{2^{b-|[y]|}}{n^{1-t}}, n\right), \frac{2^{b-|[y]|}}{n^{1-t}}\right)$ for each $[y]$. We fix the length of $[y]$, denoted by $l = |[y]|$, then consider the two cases:

- $2^l < n^t$: There are at most 2^l many such $[y]$, and each takes

$$M\left(n, n, \frac{2^{b-l}}{n^{1-t}}\right) = O\left(\frac{n^2 \cdot \frac{2^{b-l}}{n^{1-t}}}{n^{3-\omega}}\right) = O\left(n^{t+\omega} \cdot 2^{-l}\right)$$

  This follows from both Lemma 3 and the fact that $2^b = |E| = O(n^2)$. For each l, the time complexity is exactly $O(n^{t+\omega})$. So the total complexity is $\tilde{O}(n^{t+\omega})$ since $l = O(\log_2(n))$.
- $2^l \geq n^t$: There are at most 2^l many such $[y]$. Each takes

$$\begin{aligned} M\left(n, \frac{2^{b-l}}{n^{1-t}}, \frac{2^{b-l}}{n^{1-t}}\right) &= O\left(n \cdot \left(\frac{2^{b-l}}{n^{1-t}}\right)^{2-(3-\omega)}\right) \\ &= O\left(n^{(t+1)(\omega-1)+1} \cdot 2^{-l(\omega-1)}\right) \end{aligned}$$

So for all $[y]$ of length l, it takes $O\left(n^{(t+1)(\omega-1)+1} \cdot 2^{-l(\omega-2)}\right)$ time. The term $2^{-l(\omega-2)}$ is maximized when l is minimized, so $2^{-l(\omega-2)} \leq n^{-t(\omega-2)}$, and the total time for all lengths of $[y]$ is

$$\tilde{O}\left(n^{(t+1)(\omega-1)+1-t(\omega-2)}\right) = \tilde{O}\left(n^{t+\omega}\right)$$ ◀

▶ **Theorem 15.** *The All Pair Non-decreasing Paths (APNP) problem on directed simple graphs can be solved in $\tilde{O}\left(n^{\frac{3+\omega}{2}}\right)$ time. The optimal path of length l between any two vertices can also be explicitly found in $O(l)$ time if we slightly modify the algorithm.*

Proof. We choose $t = \frac{3-\omega}{2}$. The running time of this algorithm follows from previous lemmas. Since all optimal paths $OPT(i,k)$ are obtained by relaxation of edges (j,k), we can store the last edge (j,k) for each $OPT(i,k)$, so retrieving the optimal path can be done in $O(l)$ time. ◀

References

1 Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

2 D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM. doi:10.1145/28395.28396.

3 Artur Czumaj, Mirosław Kowaluk, and Andrzej Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, 2007.

4 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

5 Ran Duan, Yong Gu, and Le Zhang. Improved Time Bounds for All Pairs Non-decreasing Paths in General Digraphs. In *45th International Colloquium on Automata, Languages, and Programming*, pages 44:1–44:14, 2018.

6 Ran Duan and Seth Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *Proceedings of the 20th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–391. SIAM, 2009.

7 François Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the Coppersmith-Winograd tensor. In *Proceedings of the 29th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1046. SIAM, 2018.

8 François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, pages 296–303. ACM, 2014.

9 François Le Gall and Harumichi Nishimura. Quantum algorithms for matrix products over semirings. In *Scandinavian Workshop on Algorithm Theory*, pages 331–343. Springer, 2014.

10 Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

11 George J. Minty. A Variant on the Shortest-Route Problem. *Operations Research*, 6(6):882–883, 1958.

12 Virginia Vassilevska, Ryan Williams, and Raphael Yuster. All Pairs Bottleneck Paths and Max-min Matrix Products in Truly Subcubic Time. *Theory of Computing*, 5(9):173–189, 2009.

13 Virginia Vassilevska Williams. Nondecreasing Paths in a Weighted Graph or: How to Optimally Read a Train Schedule. *ACM Trans. Algorithms*, 6(4):70:1–70:24, September 2010.

14 Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith–Winograd. In *Proceedings of the 44th annual ACM Symposium on Theory of Computing*, pages 887–898. ACM, 2012.

15 Uri Zwick. All Pairs Shortest Paths Using Bridging Sets and Rectangular Matrix Multiplication. *J. ACM*, 49(3):289–317, May 2002.