Improved Distance Sensitivity Oracles with Subcubic Preprocessing Time

Hanlin Ren
Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
rhl16@mails.tsinghua.edu.cn

Abstract

We consider the problem of building *Distance Sensitivity Oracles* (DSOs). Given a directed graph $G = (V, E)$ with edge weights in $\{1, 2, \ldots, M\}$, we need to preprocess it into a data structure, and answer the following queries: given vertices $u, v, x \in V$, output the length of the shortest path from u to v that does not go through x. Our main result is a simple DSO with $\tilde{O}(n^{2.7233}M^2)$ preprocessing time and $O(1)$ query time. Moreover, if the input graph is undirected, the preprocessing time can be improved to $\tilde{O}(n^{2.6865}M^2)$. Our algorithms are randomized with correct probability $\geq 1 - 1/n^c$, for a constant c that can be made arbitrarily large. Previously, there is a DSO with $\tilde{O}(n^{2.8729}M)$ preprocessing time and polylog(n) query time [Chechik and Cohen, STOC'20].

At the core of our DSO is the following observation from [Bernstein and Karger, STOC'09]: if there is a DSO with preprocessing time P and query time Q, then we can construct a DSO with preprocessing time $P + \tilde{O}(Mn^2) \cdot Q$ and query time $O(1)$. (Here $\tilde{O}(\cdot)$ hides polylog(n) factors.)

2012 ACM Subject Classification Theory of computation → Shortest paths; Theory of computation → Data structures design and analysis

Keywords and phrases Graph theory, Failure-prone structures

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.79

Related Version A full version of this paper is available at `https://arxiv.org/abs/2007.11495`.

Acknowledgements I would like to thank Zihao Li for introducing this problem to me, and Ran Duan and Yong Gu for helpful discussions. I would also like to thank Hongxun Wu for reading and commenting an early draft version of this paper, and pointing out the issue with non-unique shortest paths. I am also grateful to the anonymous referees of ESA for their helpful comments that improves the presentation of this work.

1 Introduction

Suppose we are given a directed graph $G = (V, E)$, and we want to build a data structure that, given any three vertices $u, v, x \in V$, outputs the length of the shortest path from u to v that does not go through x. Such a data structure is called a *Distance Sensitivity Oracle* (or DSO for short).

The problem of constructing DSOs is motivated by the fact that real-life networks often suffer from failures. Suppose we have a network with n nodes and m (directed) links, and we want to route a package from a node u to another node v. Normally, it suffices to compute the shortest path from u to v. However, if some node x in this network fails, then our route cannot use x, and our task becomes to find the shortest path from u to v that does not go through x. Usually, there is only a very small number of failures. In this paper, we consider the simplest case, in which there is only one failed node.

The problem of constructing a DSO is well-studied: Demetrescu et al. [6] showed that given a directed graph $G = (V, E)$, there is a DSO which occupies $O(n^2 \log n)$ space, and can answer a query in $O(1)$ time. Duan and Zhang [8] improved the space complexity to $O(n^2)$, which is optimal for dense graphs (i.e. $m = \Theta(n^2)$).

Unfortunately, the oracle in [6] requires a large preprocessing time ($O(mn^2 + n^3 \log n)$). In real-life applications, the preprocessing time of the DSO is also very important. Bernstein and Karger [2, 3] improved this time bound to $\tilde{O}(mn)$. Note that the All-Pairs Shortest Paths (APSP) problem, which only asks the distances between each pair of vertices u, v, is conjectured to require $mn^{1-o(1)}$ time to solve [13]. Since we can solve the APSP problem by using a DSO, the preprocessing time $\tilde{O}(mn)$ is optimal in this sense.

However, if the edge weights are small positive integers (that do not exceed M), then the APSP problem can be solved in $\tilde{O}(n^{2.58}M^{0.7})$ time [20], which is significantly faster than $O(mn)$ for dense graphs with small weights (e.g. $M = O(1)$). Thus it might be possible to obtain better results than [3] in the regime of small integer edge weights. Weimann and Yuster [19] showed that for any constant $\alpha \in (0, 1)$, we can construct a DSO in $\tilde{O}(n^{1-\alpha+\omega}M)$ time. Here $\omega < 2.3728639$ is the exponent of matrix multiplication [10]. However, the query time for this oracle is $\tilde{O}(n^{1+\alpha})$, which is *superlinear*. Later, Grandoni and Williams [12] showed that for every constant $\alpha \in [0, 1]$, we can construct a DSO in $\tilde{O}(n^{\omega+1/2}M + n^{\omega+\alpha(4-\omega)}M)$ time, which answers each query in $\tilde{O}(n^{1-\alpha})$ time.

Recently, in an independent work, Chechik and Cohen [4] showed that a DSO with polylog(n) query time can be constructed in $\tilde{O}(Mn^{2.873})$ time, achieving *both* subcubic preprocessing time and polylogarithmic query time. The space complexity for their DSO is $\tilde{O}(n^{2.5})$.

1.1 Our Results

In this work, we show improved and simplified constructions of DSOs. We start with an observation.

▶ **Observation 1** (informal version). *If we have a DSO with preprocessing time P and query time Q, then we can build a DSO with preprocessing time $P + \tilde{O}(M \cdot n^2) \cdot Q$ and query time $O(1)$.*

For $\alpha = 0.2$, the oracle in [12] already achieves $\tilde{O}(n^{2.8729}M)$ preprocessing time and $O(n^{0.8})$ query time. Observation 1 implies that this query time can be brought down to $O(1)$.

Observation 1 can be proven by a close inspection of [3]: The algorithm in [3] for constructing a DSO picks $\tilde{O}(n^2)$ carefully chosen queries (u, v, x), such that the answers of all these queries can be computed in $\tilde{O}(mn)$ time. Then from these answers, we can easily compute a DSO with $O(1)$ query time. If, instead of computing these answers in $\tilde{O}(mn)$ time, we use the given DSO to answer these queries, the preprocessing time becomes $P + \tilde{O}(n^2) \cdot Q$. As the proof essentially repeats the arguments in [3], we leave it in Appendix A.

Our main result is a simple construction of DSOs with preprocessing time $\tilde{O}(n^{2.7233}M^2)$ and query time $O(1)$. If the input graph is undirected, we can achieve a better preprocessing time of $\tilde{O}(n^{2.6865}M^2)$.

▶ **Theorem 2.** *We can construct a DSO with $\tilde{O}(n^{2.7233}M^2)$ preprocessing time and $O(1)$ query time. Moreover, if the input graph is* undirected*, then we can construct a DSO with $\tilde{O}(n^{(3+\omega)/2}M^2) = \tilde{O}(n^{2.6865}M^2)$ preprocessing time and $O(1)$ query time. The construction algorithms are randomized and yield valid DSOs w.h.p.*[1]

[1] We say that an event happens *with high probability* (w.h.p.), if it happens with probability $1 - 1/n^C$, for some constant C that can be made arbitrarily large.

We remark that two drawbacks of our results compared to [4, 12, 19] are that it does not handle negative edge weights, and it has a worse (quadratic) dependence on M. However, our results have the currently best dependence on n, and is conceptually simple. Also, the space complexity of our DSO is $\tilde{O}(n^2)$, which is better than [4].

Non-unique shortest paths. A subtle technical issue is that the shortest paths in G may not be unique. Normally, if we perturb every edge weight by a small random value, then we can ensure that shortest paths are unique w.h.p. by the isolation lemma [15, 18]. However, the subcubic-time algorithms for shortest paths [16, 17, 20] depend crucially on the assumption that edge weights are small integers, so we cannot perform the random perturbation.

Therefore, we have to work without assuming the uniqueness of shortest paths. It turns out that Observation 1 is affected. Actually, if we assume the shortest paths are unique, we can build a DSO with preprocessing time $P + \tilde{O}(n^2) \cdot Q$ in the conclusion of Observation 1, regardless of the edge weights. However, without this assumption, Observation 1 somehow needs $\tilde{O}(Mn^2)$ queries to the slower DSO, instead of $\tilde{O}(n^2)$. See Remark 5 and Remark 6 for more details.

1.2 Notation

We mainly stick to the notation used in [7], namely:

- If p is a path, then $|p|$ denotes the number of edges in it, and $\|p\|$ denotes its length (i.e. the total weight of its edges).
- We use uv to denote the shortest path from u to v in the original graph, and $uv \diamond x$ the shortest path from u to v that does not go through x. In the case that there are multiple shortest paths, we will use e.g. "a path of the form $uv \diamond x$" to denote an arbitrary shortest path from uv in $G - x$ (i.e. the graph G with vertex x deleted). Note that if x is not in the original path uv, then $\|uv \diamond x\| = \|uv\|$.
- Let p be a path from u to v. For two vertices $a, b \in p$ such that a appears earlier than b, we say the interval $p[a, b]$ is the subpath from a to b, and $p(a, b)$ is the path $p[a, b]$ without its endpoints (a and b). If the path p is known in the context, then we may omit p and simply write $[a, b]$ or (a, b).

We define $\mathsf{MM}(n_1, n_2, n_3)$ as the complexity of multiplying an $n_1 \times n_2$ matrix and an $n_2 \times n_3$ matrix. Let a, b, c be real numbers, we define $\omega(a, b, c)$ be the infimum over all real numbers α such that $\mathsf{MM}(n^a, n^b, n^c) = O(n^\alpha)$. For any real number r, we have $\omega(1, 1, r) = \omega(1, r, 1) = \omega(r, 1, 1)$ [14], and we denote $\omega(r) = \omega(1, 1, r)$.

We also need the following adaptation of Zwick's APSP algorithm [20] (see also [12, Corollary 1]):

▶ **Theorem 3.** *Given a directed graph $G = (V, E)$ with edge weights in $\{1, 2, \ldots, M\}$, and an integer r, we can compute the distances between every pair of nodes whose shortest path uses at most r edges, in $\tilde{O}(rM \cdot \mathsf{MM}(n, n/r, n))$ time.*

Proof Sketch. We simply run the first $\lceil \log_{3/2} r \rceil$ iterations of the algorithm **rand-short-path** in [20]. The correctness of this algorithm is guaranteed by [20, Lemma 4.2]. ◀

2 Constructing a DSO in $\tilde{O}(n^{2.7233}M^2)$ Time

In this section, we prove Theorem 2.

▶ **Theorem 2.** *We can construct a DSO with $\tilde{O}(n^{2.7233}M^2)$ preprocessing time and $O(1)$ query time. Moreover, if the input graph is* undirected*, then we can construct a DSO with $\tilde{O}(n^{(3+\omega)/2}M^2) = \tilde{O}(n^{2.6865}M^2)$ preprocessing time and $O(1)$ query time. The construction algorithms are randomized and yield valid DSOs w.h.p.*

Given an integer r and a graph $G = (V, E)$, we define an *r-truncated DSO* to be a data structure that, when given a query (u, v, x), outputs the value $\min\{\|uv \diamond x\|, r\}$. In other words, an r-truncated DSO is a DSO which only needs to be correct when the path $uv \diamond x$ has length at most r. If this length is greater than r, it outputs r instead.

Inspired by Zwick's APSP algorithm [20], our main idea is to compute an r-truncated DSO for every $r = (3/2)^i$. Our strategies for small r and large r are completely different.

When r is small, the sampling approach in [12, 19] already suffices. Fix a particular query (u, v, x), we assume that $\|uv \diamond x\| \le r$. In particular, if we fix any path of the form $uv \diamond x$, then this path contains at most $r + 1$ vertices. Suppose we sample a graph by discarding each vertex w.p. $1/r$. With probability $\Omega(1/r)$, the resulting graph would "capture" this query in the sense that x is not in it but $uv \diamond x$ is completely included in it. Therefore, if we take $\tilde{O}(r)$ independent samples, and compute APSP for each sampled subgraph, we can deal with all queries w.h.p.

For large r, our idea is to compute a $(3/2)r$-truncated DSO from an r-truncated DSO. More precisely, given an r-truncated DSO with $O(1)$ query time, we can compute a $(3/2)r$-truncated DSO with $\tilde{O}(Mn/r)$ query time. First we sample a bridging set (cf. [20]) H of size $\tilde{O}(Mn/r)$. Let (u, v, x) be a query such that $r \le \|uv \diamond x\| \le (3/2)r$, then w.h.p. there is a "bridging vertex" $h \in H$ such that h is on some path of the form $uv \diamond x$, and both of the queries (u, h, x) and (h, v, x) are captured by the r-truncated DSO. If we iterate through H, we can answer the query (u, v, x) in $\tilde{O}(Mn/r)$ time. Then we use an "r-truncated" version of Observation 1 to transform this $(3/2)r$-truncated DSO into a new one with $O(1)$ query time.

2.1 Case I: r is Small

We make $\tilde{r} = \lceil 4Cr \ln n \rceil$ independent samples of graphs $G_1, G_2, \ldots, G_{\tilde{r}}$, where C is a large enough constant. The vertex set $V(G_i)$ of each graph is sampled by including each vertex independently w.p. $1 - 1/r$. The graph G_i is simply the induced subgraph of G on vertices $V(G_i)$. Then, for each $1 \le i \le \tilde{r}$, we compute all-pairs shortest paths of the graph G_i, but we only compute the shortest paths that use at most r edges. By Theorem 3, this step can be done in $\tilde{O}(rM \cdot \mathsf{MM}(n, n/r, n))$ time for each graph G_i. Alternatively, if the input graph is undirected, then this step can be done in $\tilde{O}(Mn^\omega)$ time [16, 17] for each G_i.

Consider a query (u, v, x), assume that $\|uv \diamond x\| \le r$, and fix any path of the form $uv \diamond x$. Let $1 \le i \le \tilde{r}$, we say i is *good* for the query (u, v, x), if both of the following hold.

- The graph G_i does not contain the failed vertex x.
- The graph G_i contains the path $uv \diamond x$ entirely.

For every i $(1 \le i \le \tilde{r})$, the probability that i is good for the particular query (u, v, x) is at least

$$(1/r) \cdot (1 - 1/r)^r \ge 1/4r \text{ (if } r \ge 2).$$

Given a query (u, v, x), we iterate through every i's such that $x \notin V(G_i)$, and take the smallest value among the distances from u to v in these graphs G_i. With high probability, there are only $\tilde{O}(1)$ valid i's such that $x \notin V(G_i)$, and we can preprocess this set of i's for every $x \in V$. Therefore the query time is $\tilde{O}(1)$.

The algorithm succeeds at a query (u, v, x) if there is an i that is good for (u, v, x). Since the G_i's are independent, the probability that there is an i good for (u, v, x) is at least

$$1 - (1 - 1/4r)^{\tilde{r}} \geq 1 - 1/n^C.$$

By a union bound over all n^3 triples of possible queries (u, v, x), it follows that our data structure is correct w.p. at least $1 - 1/n^{C-3}$, which is high probability.

In conclusion, there is an r-truncated DSO with $\tilde{O}(1)$ query time, whose preprocessing time is $\tilde{O}(\tilde{r} \cdot rM \cdot \mathsf{MM}(n, n/r, n))$ for directed graphs, and $\tilde{O}(\tilde{r} \cdot Mn^\omega)$ for undirected graphs.

2.2 An Observation

We need the following observation ("r-truncated" version of Observation 1), which roughly states that given an r-truncated DSO with preprocessing time P and query time Q, we can build an r-truncated DSO with preprocessing time $P + \tilde{O}(Mn^2) \cdot Q$ and query time $O(1)$. More formally, we have:

▶ **Observation 4.** *Let r be an integer, $G = (V, E)$ be an input graph whose edge weights are in $\{1, 2, \ldots, M\}$, and $\mathcal{D}$ be an arbitrary r-truncated DSO. We can construct* Fast($\mathcal{D}$)*, which is an r-truncated DSO with $O(1)$ query time and a preprocessing algorithm as follows.*

- *It first computes the distance matrix of G, and the outgoing shortest path trees rooted at each vertex.*
- *Then it invokes the preprocessing algorithm of $\mathcal{D}$ on the input graph G.*
- *At last, it makes $\tilde{O}(Mn^2)$ queries to $\mathcal{D}$, and spends $\tilde{O}(Mn^2)$ additional time to finish the preprocessing algorithm.*

We emphasize the following technical details that are not reflected in the informal statement of Observation 1. First, we need to compute the distance matrix and outgoing shortest path trees of G (henceforth the "APSP data" of G) before using $\mathcal{D}$. The APSP data can be computed in $\tilde{O}(Mn^{2.58})$ time [20], and in particular, the **wit-to-suc** algorithm in [20] describes how to compute the shortest path trees efficiently. Second, the preprocessing algorithm of $\mathcal{D}$ is called *only once*, and on the same graph G (on which we have already computed the APSP data). The reason that the second detail is important is: Suppose we have another routine that given an r-truncated DSO $\mathcal{D}$, constructs Extend($\mathcal{D}$) which is a $(3/2)r$-truncated DSO with a possibly large query time. Then given an 1-truncated DSO $\mathcal{D}^{\mathsf{start}}$, we can construct a (normal) DSO as follows:

$$\mathcal{D}^{\mathsf{final}} = \underbrace{\mathrm{Fast}(\mathrm{Extend}(\mathrm{Fast}(\mathrm{Extend}(\ldots \mathrm{Extend}(\mathcal{D}^{\mathsf{start}})))))}_{O(\log n) \text{ times}}.$$

However, even if the preprocessing algorithm of Fast($\mathcal{D}$) invokes the preprocessing algorithm of $\mathcal{D}$ *twice*, the preprocessing algorithm of $\mathcal{D}^{\mathsf{final}}$ would invoke *a polynomial times* the preprocessing algorithm of $\mathcal{D}^{\mathsf{start}}$, which is too many. In contrast, if the preprocessing algorithm of both Fast($\mathcal{D}$) and Extend($\mathcal{D}$) only invokes the preprocessing algorithm of $\mathcal{D}$ once, then the preprocessing algorithm of $\mathcal{D}^{\mathsf{final}}$ would also invoke the preprocessing algorithm of $\mathcal{D}^{\mathsf{start}}$ only once, which is okay.

2.3 Case II: r is Large

Suppose we have an r-truncated DSO $\mathcal{D}$, which has preprocessing time P and query time $O(1)$. We show how to construct a $(3/2)r$-truncated DSO, which we name as Extend($\mathcal{D}$), with preprocessing time $P + O(n^2)$ and query time $\tilde{O}(nM/r)$. This is done by the following bridging set argument.

Let $\mathcal{P}$ be a set of paths that contains exactly one path of the form $uv \diamond x$, for every u, v, x such that $r \le \|uv \diamond x\| < (3/2)r$. This corresponds to the paths that $\mathcal{D}$ cannot deal with, but Extend($\mathcal{D}$) has to output the correct answer. Let $p = uv \diamond x \in \mathcal{P}$, $\mathsf{mid}(p)$ be the set of vertices $y \in p$ such that $\|p[u, y]\| < r$ and $\|p[y, v]\| < r$. (See Figure 1.) For every $y \in \mathsf{mid}(p)$, as $p[u, y]$, $p[y, v]$ are of the form $uy \diamond x$ and $yv \diamond x$ respectively, it follows that $\mathcal{D}$ can find $\|uy \diamond x\|$ and $\|yv \diamond x\|$ correctly. Moreover, $|\mathsf{mid}(p)| \ge r/3M$.

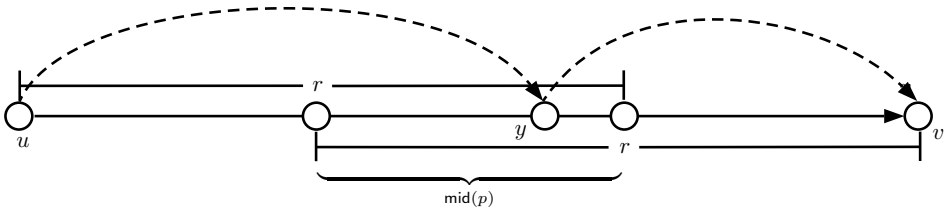

Figure 1 Example of a path $p = uv \diamond x$. If we can find a vertex $y \in \mathsf{mid}(p)$, then we can use $\mathcal{D}$ to compute $\|uy \diamond x\|$ and $\|yv \diamond x\|$, thus to compute the length of p.

Fix a large enough constant C, the preprocessing algorithm of Extend($\mathcal{D}$) is as follows: We preprocess $\mathcal{D}$, and then randomly sample a set H of vertices, where every vertex $v \in V$ is in H with probability $\min\{1, 3CM \ln n/r\}$ independently. We have $|H| = \tilde{O}(nM/r)$ w.h.p.

Fix $u, v, x \in V$, suppose $p = uv \diamond x$ and $r \le \|p\| < (3/2)r$. Then the probability that H hits $\mathsf{mid}(p)$ (i.e. $H \cap \mathsf{mid}(p) \ne \varnothing$) is at least

$$1 - (1 - 3CM \ln n/r)^{r/3M} \ge 1 - 1/n^C.$$

By a union bound over $O(n^3)$ paths in $\mathcal{P}$, it follows that w.h.p. H hits $\mathsf{mid}(p)$ for every path $p \in \mathcal{P}$.

The query algorithm for Extend($\mathcal{D}$) is as follows: Given a query (u, v, x), if $\mathcal{D}(u, v, x) < r$, then we output $\mathcal{D}(u, v, x)$; otherwise we output

$$\min\left\{(3/2)r, \min_{h \in H}\{\mathcal{D}(u, h, x) + \mathcal{D}(h, v, x)\}\right\}.$$

It is easy to see that Extend($\mathcal{D}$) is a correct $(3/2)r$-truncated DSO, has preprocessing time $P + O(n^2)$ and query time $\tilde{O}(nM/r)$.

2.4 Putting it Together

Let $a \in [0, 1]$ be a constant that we pick later, and $r = n^a$. To start, we invoke Section 2.1 to build an r-truncated DSO for $r = n^a$, which costs $\tilde{O}(r^2 M \cdot \mathsf{MM}(n, n/r, n))$ time for directed graphs or $\tilde{O}(r \cdot Mn^\omega)$ for undirected graphs. Then for every $1 \le i \le \lceil \log_{3/2}(Mn/r) \rceil$, suppose we have an $r(3/2)^{i-1}$-truncated DSO $\mathcal{D}^{i-1}$, we can construct $\mathcal{D}^i = \text{Fast}(\text{Extend}(\mathcal{D}^{i-1}))$ which is an $r(3/2)^i$-truncated DSO. This step costs $\tilde{O}(n^3 M^2/(r(3/2)^i))$ time. The preprocessing algorithm terminates when $i = i_\star = \lceil \log_{3/2}(Mn/r) \rceil = O(\log n)$, and we obtain an $r(3/2)^{i_\star}$-truncated DSO which is a (normal) DSO.

Case 1: the input graph is undirected. The total preprocessing time is

$$\tilde{O}\big(r \cdot Mn^{\omega} + n^3M^2/r\big) = \tilde{O}\Big(n^{\max\{\omega+a,3-a\}}M^2\Big).$$

When $a = (3-\omega)/2$, this time complexity is $\tilde{O}(n^{(3+\omega)/2}M^2) = \tilde{O}(n^{2.6865}M^2)$.

Therefore, given an undirected graph $G = (V, E)$, there is a DSO with $\tilde{O}(n^{2.6865}M^2)$ preprocessing time and $O(1)$ query time.

Case 2: the input graph is directed. The total preprocessing time is

$$\tilde{O}\big(r^2M \cdot \mathsf{MM}(n, n/r, n) + n^3M^2/r\big) = \tilde{O}\Big(n^{2a+\omega(1-a)}M + n^{3-a}M^2\Big).$$

(Recall that $\omega(1-a)$ is the exponent of multiplying an $n \times n^{1-a}$ matrix and an $n^{1-a} \times n$ matrix.)

Let $a = 0.276724$, then $1 - a = 0.723276$. By convexity of the function $\omega(\cdot)$ [14], we have

$$\omega(1-a) \leq \frac{(a-0.25)\omega(0.7) + (0.3-a)\omega(0.75)}{0.75-0.7}.$$

We substitute $\omega(0.7) \leq 2.154399$ and $\omega(0.75) \leq 2.187543$ [11], and obtain:

$$\omega(1-a) \leq 20 \cdot ((a-0.25) \cdot 2.154399 + (0.3-a) \cdot 2.187543) \leq 2.169829.$$

Therefore, given a directed graph $G = (V, E)$, there is a DSO with

$$\tilde{O}\Big(n^{\max\{2a+\omega(1-a),3-a\}}M^2\Big) = \tilde{O}(n^{2.723277}M^2).$$

preprocessing time and $O(1)$ query time.

3 Open Questions

The main open problem after this work is to improve the preprocessing time for DSOs. We believe it should be possible to overcome the issue that shortest paths are not unique, and improve the preprocessing time of the oracle specified in Observation 1 and Observation 4 to $P + \tilde{O}(n^2) \cdot Q$ (dropping the M factor). Then, we can build DSOs with $O(1)$ query time, and $\tilde{O}(n^{2.7233}M)$ preprocessing time (for directed graphs), or $\tilde{O}(n^{(3+\omega)/2}M)$ preprocessing time (for undirected graphs).

Can we improve the preprocessing time for directed graphs to $\tilde{O}(n^{2.58}M)$, matching the current best algorithm for APSP in directed graphs [20]? Can we improve the preprocessing time for undirected graphs to $\tilde{O}(n^{\omega}M)$, matching the nearly-optimal algorithm for APSP in undirected graphs [16, 17]?

References

1 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

2 Aaron Bernstein and David R. Karger. Improved distance sensitivity oracles via random sampling. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 34–43. SIAM, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347087.

3 Aaron Bernstein and David R. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 – June 2, 2009*, pages 101–110. ACM, 2009. doi:10.1145/1536414.1536431.

4 Shiri Chechik and Sarel Cohen. Distance sensitivity oracles with subcubic preprocessing time and fast query time. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 1375–1388. ACM, 2020. doi:10.1145/3357713.3384253.

5 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On Cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014. doi:10.1007/s00453-012-9683-x.

6 Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008. doi:10.1137/S0097539705429847.

7 Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 506–515. SIAM, 2009. URL: http://dl.acm.org/citation.cfm?id=1496770.1496826.

8 Ran Duan and Tianyi Zhang. Improved distance sensitivity oracles via tree partitioning. In Faith Ellen, Antonina Kolokolova, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures – 15th International Symposium, WADS 2017, St. John's, NL, Canada, July 31 – August 2, 2017, Proceedings*, volume 10389 of *Lecture Notes in Computer Science*, pages 349–360. Springer, 2017. doi:10.1007/978-3-319-62127-2_30.

9 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 165–169. IEEE Computer Society, 1982. doi:10.1109/SFCS.1982.39.

10 François Le Gall. Powers of tensors and fast matrix multiplication. In Katsusuke Nabeshima, Kosaku Nagasaka, Franz Winkler, and Ágnes Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. doi:10.1145/2608628.2608664.

11 Francois Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the Coppersmith-Winograd tensor. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1029–1046. SIAM, 2018. doi:10.1137/1.9781611975031.67.

12 Fabrizio Grandoni and Virginia Vassilevska Williams. Improved distance sensitivity oracles via fast single-source replacement paths. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 748–757. IEEE Computer Society, 2012. doi:10.1109/FOCS.2012.17.

13 Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1236–1252. SIAM, 2018. doi:10.1137/1.9781611975031.80.

14 Grazia Lotti and Francesco Romani. On the asymptotic complexity of rectangular matrix multiplication. *Theor. Comput. Sci.*, 23:171–185, 1983. doi:10.1016/0304-3975(83)90054-3.

15 Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987. doi:10.1007/BF02579206.

16 Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995. doi:10.1006/jcss.1995.1078.

17 Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 605–615. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814635.

18 Noam Ta-Shma. A simple proof of the isolation lemma. *Electronic Colloquium on Computational Complexity (ECCC)*, 22:80, 2015. URL: https://eccc.weizmann.ac.il/report/2015/080.

19 Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Trans. Algorithms*, 9(2):14:1–14:13, 2013. doi:10.1145/2438645.2438646.

20 Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002. doi:10.1145/567112.567114.

A Proof of Observation 4 and Observation 1

In this section, we prove Observation 4. Note that Observation 1 follows from Observation 4 by setting $r = +\infty$.

▶ **Observation 4.** *Let r be an integer, $G = (V, E)$ be an input graph whose edge weights are in $\{1, 2, \ldots, M\}$, and $\mathcal{D}$ be an arbitrary r-truncated DSO. We can construct* Fast($\mathcal{D}$)*, which is an r-truncated DSO with $O(1)$ query time and a preprocessing algorithm as follows.*

- *It first computes the distance matrix of G, and the outgoing shortest path trees rooted at each vertex.*
- *Then it invokes the preprocessing algorithm of $\mathcal{D}$ on the input graph G.*
- *At last, it makes $\tilde{O}(Mn^2)$ queries to $\mathcal{D}$, and spends $\tilde{O}(Mn^2)$ additional time to finish the preprocessing algorithm.*

Let $T_{\mathsf{out}}(u)$ be the outgoing shortest path tree rooted at u. In this section, the notation uv denotes the shortest path from u to v *in the tree* $T_{out}(u)$. In particular, for any $x \in uv$, the path ux is guaranteed to be a subpath of uv.

A.1 The Preprocessing Algorithm

We review and slightly modify the preprocessing algorithm of [3]. For convenience, we denote $\|p\|_r = \min\{\|p\|, r\}$ for any path p and number r.

We define a path to be *good* if it is a subpath of some shortest path in the shortest path trees. In other words, for any vertices $u, v, x \in V$ such that $x \in uv$, we say the subpath $(uv)[x, v]$ is *good*. Note that if the shortest paths in G are unique, then the set of good paths coincides with the set of shortest paths in G. Also note that there are only $O(n^3)$ good paths even if the shortest paths are not unique.

Assigning priorities. We assign each vertex a *priority*, which is independently sampled from the following distribution: for any positive integer c, each vertex has priority c w.p. $1/2^c$. Denote $c(v)$ the priority of the vertex v. With high probability, all of the following are true:

- The maximum priority is $O(\log n)$.
- For every $c \le O(\log n)$, there are $\tilde{O}(n/2^c)$ vertices with priority c.
- Let C be a large enough constant. For every good path p with at least $C \cdot 2^c \log n$ edges, there is a vertex on p whose priority is greater than c.

In the following discussions, we will assume that all of the above assumptions hold.

Fix a pair $u, v \in V$, let s_i be the first vertex in uv with priority $\geq i$, and t_i be the last such vertex. Then we can write the path uv as

$$u \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \ldots \rightsquigarrow s_{O(\log n)} \rightsquigarrow t_{O(\log n)} \rightsquigarrow \ldots \rightsquigarrow t_1 \rightsquigarrow v.$$

We say that the vertices u, v, s_i, t_i are *key vertices*, and the i-th key vertex is denoted as k_i. Then the path uv can also be written as

$$u = k_0 \rightsquigarrow k_1 \rightsquigarrow \ldots \rightsquigarrow k_{O(\log n)} = v.$$

It is important to see that

$$|(uv)[k_i, k_{i+1}]| \leq C \cdot 2^{\min\{c(k_i), c(k_{i+1})\}} \log n \tag{1}$$

for every valid i, as otherwise there will be another key vertex between k_i and k_{i+1}.

Data structure for quick location. Suppose we are given a query (u, v, x), the first thing we should do is to "locate" x, i.e. find the key vertices $k_i, k_{i+1} \in uv$ such that $x \in (uv)[k_i, k_{i+1}]$. We will utilize the following data (cf. [2]).

For every $u, v \in V$, we compute

- $\mathsf{CL}[u, v, c]$ (for "center left"): the first vertex in uv with priority at least c; and
- $\mathsf{CR}[u, v, c]$ (for "center right"): the last vertex in uv with priority at least c.

It is easy to compute these numbers in $\tilde{O}(n^2)$ time: for each priority c and each vertex $u \in V$, we simply perform a depth-first search on the outgoing shortest path tree $T_{\mathsf{out}}(u)$ rooted at u to compute all $\mathsf{CL}[u, \cdot, c]$ and $\mathsf{CR}[u, \cdot, c]$.

We also compute a structure called BCP (for "biggest center priority"). In this structure, for each vertex $u \in V$, we preprocess the outgoing shortest path tree $T_{\mathsf{out}}(u)$ rooted at u in $\tilde{O}(n)$ time such that given any vertices $x, y \in V$, we can find the biggest priority of any vertex on the path from x to y in $T_{\mathsf{out}}(u)$ in $O(1)$ time [5]. Then, given any good path p (p is specified by vertices u, v, x and $p = (uv)[x, v]$), we can find the biggest priority of any vertex on p in $O(1)$ time.

In addition, for every $u, v \in V$, we store the key vertices on uv into a hash table of size $O(\log n)$. Given a vertex x, we can output whether x is among these key vertices on uv in $O(1)$ worst-case time [9].

Data structure for avoiding a failure. We use $\mathcal{D}$ to preprocess the input graph. Then we compute the following data:

(Data a) For every $u, v \in V$, and every $1 \leq i \leq \min\{M \cdot C \cdot 2^{c(u)} \log n, |uv|\}$, let x_i be the i-th vertex in the path uv. (Here u is the 0-th vertex.) We compute and store the value $\|uv \diamond x_i\|_r$. Symmetrically, let x_{-i} be the *last* i-th vertex in the path vu (not uv!), for every $1 \leq i \leq \min\{M \cdot C \cdot 2^{c(u)} \log n, |vu|\}$, we compute and store $\|vu \diamond x_{-i}\|_r$.

(Data b) For every $u, v \in V$ and consecutive key vertices $k_i, k_{i+1} \in uv$, let y be the vertex in the portion $k_i \rightsquigarrow k_{i+1}$ that maximizes $\|uv \diamond y\|_r$. We compute and store $\|uv \diamond y\|_r$.

(Data c) For every $u, v \in V$ and key vertex $k_i \in uv$, we compute and store $\|uv \diamond k_i\|_r$.

For each priority $c \leq \tilde{O}(1)$, there are $\tilde{O}(n/2^c)$ vertices u whose priority is exactly c. In (Data a), we make $\tilde{O}(nM2^c)$ queries for each such u ($\tilde{O}(M2^c)$ queries for each $v \in V$). Therefore in total, we make $\tilde{O}(Mn^2)$ queries in (Data a). We will show in Appendix A.3 that we can compute (Data b) using $\tilde{O}(n^2)$ queries to $\mathcal{D}$ and $\tilde{O}(n^2)$ additional time. (Data c) can be computed in $\tilde{O}(n^2)$ queries easily.

▶ Remark 5. If shortest paths in G are unique, then in (Data a) we only need to store $\|uv \diamond x_i\|_r$ and $\|vu \diamond x_{-i}\|_r$ for $i \le C \cdot 2^{c(u)} \log n$ (shaving off a factor of M). The reason will be evident when we see the query algorithm.

A.2 The Query Algorithm

Let (u, v, x) be a query. We check whether $x \in uv$ in $O(1)$ time on the shortest path tree $T_{\mathsf{out}}(u)$. If $x \notin uv$, then it is easy to see that $\|uv \diamond x\|_r = \|uv\|_r$.

We check whether x is a key vertex on uv (that is, $x = k_i$ for some i), using the hash tables. If this is the case, we return $\|uv \diamond x\|_r$ stored in (Data c) immediately.

Otherwise, we start by finding two consecutive key vertices $k_i, k_{i+1} \in uv$ such that $x \in [k_i, k_{i+1}]$. Recall that, if ℓ is the biggest priority of any vertex on uv, then the key vertices on uv are

$$(u =)\mathsf{CL}[u,v,1] \rightsquigarrow \mathsf{CL}[u,v,2] \rightsquigarrow \ldots \rightsquigarrow \mathsf{CL}[u,v,\ell] \rightsquigarrow \mathsf{CR}[u,v,\ell] \rightsquigarrow \ldots \rightsquigarrow \mathsf{CR}[u,v,2] \rightsquigarrow \mathsf{CR}[u,v,1](= v).$$

Therefore, we can find k_i in $O(1)$ time using the following procedure. Let c_1 be the biggest priority of any vertex in ux (which coincides the $[u, x]$ portion of uv), and c_2 be the biggest priority of any vertex in the $[x, v]$ portion of uv. We can compute c_1, c_2 from the structure BCP in $O(1)$ time. If $c_2 = \ell$, then x is in the range $(u, \mathsf{CR}[u, v, \ell])$, so $k_i = \mathsf{CL}[u, v, c_1]$. Otherwise, x is in the range $(\mathsf{CR}[u, v, \ell], v)$, so $k_i = \mathsf{CR}[u, v, c_2 + 1]$. We can find k_{i+1} similarly.

By (1), if $k_i = u$, then $|(uv)[u, x]| \le C \cdot 2^{c(u)} \log n$, and we can look up the value $\|uv \diamond x\|_r$ from (Data a) directly. Similarly, if $k_{i+1} = v$ then we can also look up $\|uv \diamond x\|_r$ from (Data a).

Now we assume that $k_i \ne u$ and $k_{i+1} \ne v$. A crucial observation is that

$$\|uv \diamond x\| = \min\{\|uk_{i+1} \diamond x\| + \|k_{i+1}v\|, \|uk_i\| + \|k_iv \diamond x\|, \|uv \diamond y\|\}, \tag{2}$$

where y is the vertex in $[k_i, k_{i+1}]$ that maximizes $\|uv \diamond y\|$. The proof of (2) is as follows:

- **(i)** If there is a path of the form $uv \diamond x$ that goes through k_i, then $\|uv \diamond x\| = \|uk_i\| + \|k_iv \diamond x\|$.
- **(ii)** If there is a path of the form $uv \diamond x$ that goes through k_{i+1}, then $\|uv \diamond x\| = \|uk_{i+1} \diamond x\| + \|k_{i+1}v\|$.
- **(iii)** If every path of the form $uv \diamond x$ does not through either k_i or k_{i+1}, then every path of the form $uv \diamond x$ avoids the entire portion of $k_i \rightsquigarrow k_{i+1}$, thus also avoids y. We have $\|uv \diamond x\| \ge \|uv \diamond y\|$. But $\|uv \diamond y\| \ge \|uv \diamond x\|$ by definition of y, thus $\|uv \diamond x\| = \|uv \diamond y\|$.

It is easy to see that a similar equation holds for r-truncated DSOs:

$$\|uv \diamond x\|_r = \min\{\|uk_{i+1} \diamond x\|_r + \|k_{i+1}v\|, \|uk_i\| + \|k_iv \diamond x\|_r, \|uv \diamond y\|_r, r\}, \tag{3}$$

where y is any vertex in $[k_i, k_{i+1}]$ that maximizes $\|uv \diamond y\|_r$.

Recall that we already know the values $\|uk_i\|$ and $\|k_{i+1}v\|$. To compute $\|uk_{i+1} \diamond x\|_r$, we note that if x is the last j-th vertex in uk_{i+1}, then $j \le C \cdot 2^{c(k_{i+1})} \log n$. Therefore we can look up the value of $\|uk_{i+1} \diamond x\|_r$ from (Data a). Similarly, to compute $\|k_iv \diamond x\|_r$, we note that if x is the j-th vertex in the $k_i \rightsquigarrow v$ portion of the path uv, then $j \le C \cdot 2^{c(k_i)} \log n$. However, k_iv may be different from the $k_i \rightsquigarrow v$ portion of the path uv. Nevertheless, since $\|k_ix\| = \|(uv)[k_i, x]\| \le M \cdot C \cdot 2^{c(k_i)} \log n$, we also have $|k_ix| \le M \cdot C \cdot 2^{c(k_i)} \log n$, so we can still look up the value $\|k_iv \diamond x\|_r$ in (Data a). Finally, we can look up $\|uv \diamond y\|_r$ from (Data b).

We can see that the query time is $O(1)$.

▶ Remark 6. If shortest paths in G are unique, then the path $k_i v$ actually coincides with the $k_i \rightsquigarrow v$ portion of the path uv. Therefore, $|k_i x| = |(uv)[k_i, x]| \le C \cdot 2^{c(k_i)} \log n$. In this case we do not need to "sacrifice" a factor of M in (Data a): We can look up $\|k_i v \diamond x\|_r$ even if we only stored the values $\|k_i v \diamond x_{i'}\|_r$ for $i' \le C \cdot 2^{c(k_i)} \log n$, as in Remark 5.

A.3 Computing (Data b)

We will use the following notation. Let p be a path from u to v which is fixed in context, and a, b be two vertices in p. We will say that $a < b$ if $|p[u, a]| < |p[u, b]|$, i.e. a appears strictly before b on the path p. Similarly, $a > b$, $a \le b$, $a \ge b$ mean $|p[u, a]| > |p[u, b]|$, $|p[u, a]| \le |p[u, b]|$, $|p[u, a]| \ge |p[u, b]|$ respectively.

Let $u, v \in V$ and $s < t$ be two vertices on the path uv. Let $y \in (uv)[s, t]$ be the vertex in $(uv)[s, t]$ which maximizes $\|uv \diamond y\|_r$. We first show that *assuming we have built some oracles*, we can find this vertex y in $O(\log n)$ oracle calls and $O(\log n)$ additional time. The idea is to use a binary search described in [3, Section 6].

▶ **Lemma 7.** *Let r be an integer, $u, v \in V$, p be the path uv, and s, t be two vertices on p such that $u < s < t < v$. Suppose we have the following oracles, each with $O(1)$ query time:*

- *an oracle that given a vertex $x \in p[s, t]$, outputs $\|ut \diamond x\|_r$;*
- *an oracle that given a vertex $x \in p[s, t]$, outputs $\|sv \diamond x\|_r$;*
- *an oracle that given an interval $p[s', t']$ such that $s \le s' \le t' \le t$, outputs a vertex $x \in p[s', t']$ that maximizes the value $\|ut \diamond x\|_r$.*

Then we can find a vertex $y \in p[s, t]$ which maximizes $\|uv \diamond y\|_r$ in $O(\log n)$ time.

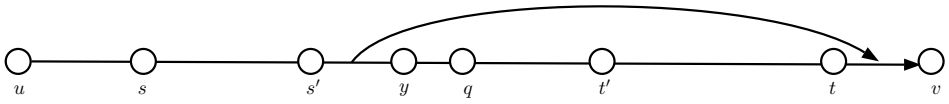

Figure 2 If every path of the form $sv \diamond y$ does not go through t, then every path of the form $sv \diamond y$ does not go through the whole interval $p[q, t']$.

Proof. For any $y \in p[s, t]$, we denote

$$h(y) = \min\{\|ut \diamond y\|_r + \|tv\|, \|us\| + \|sv \diamond y\|_r, r\}.$$

By (3), we have $\|uv \diamond y\|_r = \min\{h(y), \|uv \diamond y^\star\|_r\}$ where $y^\star$ is some vertex independent of y. Thus it suffices to find some $y \in p[s, t]$ that maximizes $h(y)$.

We use a binary search. Assume that we know the optimal y is in some interval $p[s', t']$, where $s \le s' < t' \le t$. (Initially we set $s' = s$ and $t' = t$.) If $|p[s', t']| = O(1)$ then we can use brute force to find a vertex $y \in p[s', t']$ that maximizes $h(y)$. Otherwise let q be the middle point of $p[s', t']$, and we use the third oracle to find a vertex $y \in p[s', q]$ that maximizes $\|ut \diamond y\|_r$. There are two cases:

- If $\min\{\|ut \diamond y\|_r + \|tv\|, r\} = h(y)$, then we can restrict our attention to the interval $p[q, t']$. This is because for every vertex $x \in p[s', q]$,

$$h(x) \le \min\{\|ut \diamond x\|_r + \|tv\|, r\} \le \min\{\|ut \diamond y\|_r + \|tv\|, r\} \le h(y).$$

- Otherwise, $h(y) = \|us\| + \|sv \diamond y\|_r$, and every path of the form $sv \diamond y$ does not go through t. Therefore every path of the form $sv \diamond y$ avoids every vertex in $p[q, t']$. (See Figure 2.) For every vertex $x \in p[q, t']$,

$$h(x) \le \|us\| + \|sv \diamond y\| \le h(y).$$

It follows that we can restrict our attention to the interval $[s', q]$ now.
Therefore, we can always shrink the length of our candidate interval $p[s', t']$ by a half. It follows that we can find the desired vertex y in $O(\log n)$ time. ◀

Now we show how to compute (Data b) in $\tilde{O}(n^2)$ time (assuming that (Data a) is ready). The most crucial ingredient is the following Range Maximum Query (RMQ) structures (used in the third item of Lemma 7).

For every $u, v \in V$, consider the following sequence (of length $\ell = \min\{|uv| - 1, C \cdot 2^{c(v)} \log n\}$):

$$(\|uv \diamond x_{-1}\|_r, \|uv \diamond x_{-2}\|_r, \ldots, \|uv \diamond x_{-\ell}\|_r),$$

where x_{-i} denotes the last i-th vertex in the path uv (v is the last 0-th). We build an RMQ structure of this sequence, which given a query (s, t) $(1 \le s \le t \le \ell)$, outputs a number $i \in [s, t]$ that maximizes $\|uv \diamond x_{-i}\|_r$. After we compute the above sequence, this data structure can be preprocessed in $O(\ell)$ time, and each query costs $O(1)$ time [1].

For every priority $c \le O(\log n)$, there are $\tilde{O}(n/2^c)$ vertices v of this priority, and for each vertex v we construct n RMQ structures (one for each $u \in V$) on length-$\tilde{O}(2^c)$ sequences. The total size of these RMQ structures is

$$\sum_{c=1}^{O(\log n)} \tilde{O}(n/2^c) \cdot n \cdot \tilde{O}(2^c) = \tilde{O}(n^2).$$

Therefore, these RMQ structures can be preprocessed in $\tilde{O}(n^2)$ time. (Note that every element $\|uv \diamond x_{-i}\|_r$ is already computed in (Data a).)

To compute (Data b), we enumerate u, v, k_i, k_{i+1} where k_i, k_{i+1} are consecutive key vertices in uv. There are $\tilde{O}(n^2)$ possible combinations of (u, v, k_i, k_{i+1}). As argued in Appendix A.2, we know that the following data are already computed in (Data a):

- $\|uk_{i+1} \diamond x\|_r$, for any $x \in (uv)[k_i, k_{i+1}]$;
- $\|k_i v \diamond x\|_r$, for any $x \in (uv)[k_i, k_{i+1}]$.

Since uk_{i+1} is a prefix of uv (as defined in the outgoing shortest path trees), we also have the following RMQ oracles constructed above:

- An oracle that given any interval $[s', t']$ on the path uv such that $k_i \le s' \le t' \le k_{i+1}$, finds the vertex $y \in [s', t']$ that maximizes $\|uk_{i+1} \diamond y\|_r$ in $O(1)$ time.

It follows from Lemma 7 that we can find a vertex $y \in (uv)[k_i, k_{i+1}]$ that maximizes $\|uv \diamond y\|_r$ in $O(\log n)$ time. The total time for computing (Data b) is thus $\tilde{O}(n^2)$.