Brief Announcement: Algorithms for Distance Sensitivity Oracles on the PRAM

Vignesh Manoharan
The University of Texas at Austin
Austin, TX, USA
vigneshm@cs.utexas.edu

Vijaya Ramachandran
The University of Texas at Austin
Austin, TX, USA
vlr@cs.utexas.edu

Abstract

The distance sensitivity oracle (DSO) problem asks us to preprocess a given graph $G = (V, E)$ in order to answer queries of the form $d(x, y, e)$, which denotes the shortest path distance in G from vertex x to vertex y when edge e is removed. This is an important problem for network communication, and it has been extensively studied in the sequential setting [2, 4, 8, 9] and recently in the distributed CONGEST model [7]. However, no prior DSO results tailored to the parallel setting were known. We present the first PRAM algorithms to construct DSOs in directed weighted graphs, that can answer a query in $O(1)$ time with a single processor after preprocessing.

CCS Concepts

• **Theory of computation** → **Shortest paths**; **Shared memory algorithms**; • **Mathematics of computing** → **Graph algorithms**.

Keywords

Parallel Algorithms, Distance Sensitivity Oracles

ACM Reference Format:
Vignesh Manoharan and Vijaya Ramachandran. 2025. Brief Announcement: Algorithms for Distance Sensitivity Oracles on the PRAM. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28–August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3694906.3743333

1 Introduction

In a network modeled by a graph $G = (V, E)$, we investigate the problem of computing shortest path distances when an edge $e \in E$ fails. Specifically, we need to answer queries $d(s, t, e)$ for $s, t \in V, e \in E$, where $d(s, t, e)$ is the shortest path distance from s to t in G when edge e is removed. This is a fundamental problem in network communication for routing under an arbitrary edge failure.

Let $|V| = n, |E| = m$. Since explicitly computing all distances would be prohibitively expensive due to output size, we instead consider Distance Sensitivity Oracles (DSOs). In the DSO problem, we first preprocess the graph G to construct an oracle that stores certain information in order to answer queries quickly. Note that without any preprocessing, we can answer a query using a single source shortest path (SSSP) computation. The goal in DSO construction is to obtain a faster query time, ideally constant query time, while keeping the preprocessing cost reasonable.

The DSO problem has received considerable attention in the sequential setting. Starting from [4], algorithms with different tradeoffs between query time and preprocessing time were obtained, culminating in an efficient algorithm with preprocessing time $\tilde{O}(mn)$, which constructs a distance sensitivity oracle of size $\tilde{O}(n^2)$ that can answer any query in $O(1)$ time [2]. This construction nearly matches the best runtime of APSP. Other algorithms addressed DSO for dense graphs [8], and higher query time [9]. Recently, the DSO problem was studied in the distributed CONGEST model [7] where the input graph models the communication network itself.

Despite the importance of the DSO problem, no non-trivial DSO constructions are known for the parallel setting. In this paper, we present PRAM algorithms for computing DSOs in directed weighted graphs. All our algorithms involve a preprocessing step, after which queries can be answered in $O(1)$ time using a single processor. Our algorithms have a range of tradeoffs between work and parallel time used for preprocessing, described in Section 1.2.

1.1 Preliminaries

Let $G = (V, E)$ be a directed weighted graph, let $|V| = n, |E| = m$. Each edge $(s, t) \in E$ (for $s, t \in V$) has a non-negative integer weight $w(s, t)$. We denote the shortest path distance from s to t by $d(s, t)$. We use $d(s, t, e)$, for $s, t \in V, e \in E$, to denote the shortest path distance from s to t in the graph $G - \{e\}$, i.e., the graph G with edge e removed, also known as the replacement path distance.

We use the work-depth model to present our algorithms, similar to other PRAM results [3, 5]. Our work and time bounds are presented in $\tilde{O}$ format which hides polylog factors and are valid for EREW, CREW, CRCW models of PRAM due to known standard reductions between these models.

1.2 Results

We present algorithms for DSO in directed weighted graphs with integer weights. Integer weights are needed in our algorithms only for computing SSSP where we utilize the PRAM algorithms of [3, 6]. Otherwise, our results hold for real weighted graphs. All of our preprocessing algorithms are randomized, and are correct w.h.p. in n. In all our algorithms, we construct an $\tilde{O}(n^2)$-sized oracle that can answer any query in $O(1)$ time on a single processor.

Our first algorithm DSO-A with bounds shown in Theorem 1.1 implements the sequential DSO of [2] on the PRAM. This algorithm is work-efficient for constant query cost, matching the preprocessing, space and query time of the DSO in [2]. This is described in Section 3.

SPAA '25, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/25/07
https://doi.org/10.1145/3694906.3743333

Theorem 1.1. *We can construct a DSO for directed weighted graphs on the PRAM with preprocessing cost of $\tilde{O}(mn)$ work and $\tilde{O}(n^{1/2+o(1)})$ parallel time. The constructed oracle has size $\tilde{O}(n^2)$, and can answer any query in $O(1)$ work.*

We present a fast algorithm DSO-B with bounds shown in Theorem 1.2 that is work efficient for dense graphs ($m = \Theta(n^2)$). The work also matches the current best work of PRAM APSP in $\tilde{O}(1)$ parallel time. This result is described in Section 4.2.1.

Theorem 1.2. *We can construct a DSO for directed weighted graphs on the PRAM with preprocessing cost of $\tilde{O}(n^3)$ work and $\tilde{O}(1)$ parallel time. The constructed oracle has size $\tilde{O}(n^2)$, and can answer any query in $O(1)$ work.*

We use the notion of hop-limited DSOs in DSO-B, and use a sequential technique from [8] to construct a $\frac{3}{2}h$-hop limited DSO given a h-hop limited DSO ($1 \leq h \leq n$) at the cost of increased preprocessing. This extension, which is described in Section 4.1 consists of two steps: constructing a $\frac{3}{2}h$-hop limited DSO with high query time using vertex sampling (in Section 4.1.1), and then reducing the query time to $O(1)$ using the framework of [2] and the PRAM APSP algorithm of [5] (in Section 4.1.2). Using a simple 2-hop DSO construction as a base case together with repeated application of the parallel DSO extension procedure gives us the result.

Theorem 1.3 gives a tradeoff between work and parallel time in algorithm DSO-C (Section 4.2.2).

Theorem 1.3. *We can construct a DSO for directed weighted graphs on the PRAM with preprocessing cost of $\tilde{O}(mnh + (n^3/h))$ work and $\tilde{O}(h)$ parallel time, for any $1 \leq h \leq n$. The constructed oracle has size $\tilde{O}(n^2)$, and can answer any query in $O(1)$ work.*

In Theorem 1.3, we can choose $h = \tilde{o}\left(\min\left(\frac{n^2}{m}, \sqrt{n}\right)\right)$ to obtain an algorithm that achieves sub-$\sqrt{n}$ parallel time with sub-n^3 work as long as $m = \tilde{o}(n^2)$, i.e., the graph is not fully dense. Theorem 1.3 is only meaningful for $h \leq \tilde{O}(n^{1/2+o(1)})$ due to Theorem 1.1. We use a hop limited approach similar to Theorem 1.2, but we use a more sophisticated base case construction that directly constructs a h-hop limited DSO with a graph sampling technique, using ideas from a sequential DSO algorithm [9]. After the base case construction, we repeatedly apply the DSO extension procedure (in Section 4.1).

2 Framework for Distance Sensitivity Oracle

Our PRAM algorithms for DSO use techniques from sequential DSO algorithms [2, 4, 8, 9]. In this section, we describe a framework for DSO construction due to Bernstein and Karger [2]. In Sections 3 and 4, we will use different implementations of this framework along with other techniques in our PRAM algorithms. We present our results for edge removal (this can be modified for vertex removal as in [2]). Recall that the sequential DSO construction of [2] has $\tilde{O}(mn)$ preprocessing time, $\tilde{O}(n^2)$ space requirement, and $O(1)$ query time.

The DSO framework of [2] is described in Algorithm 1. Each vertex is assigned a priority $k \in \{1, 2, \ldots \log n\}$, and a vertex with priority k is called a *k-center*. In each x-y shortest path, for $x, y \in V$, the algorithm identifies a sequence of centers of strictly increasing priority up to the maximum, followed by strictly decreasing priority to y. A center x *covers* an edge e, if e is on its outgoing (or incoming) shortest path tree, and there is no center of higher priority on the shortest path from x to e. Thus, the sequence of centers partitions the path into intervals that are covered by their endpoint centers. By randomly sampling k-centers with probability $\Theta(1/2^k)$, a center of priority k only has to cover edges within $\tilde{O}(2^k)$ hops from it.

The algorithm precomputes distances $d(x, y, e)$ for all $y \in V$ and edges $e \in E$ that are covered by center x. Furthermore, the algorithm uses the notion of a *bottleneck edge* for an interval, which is the edge in the interval that maximizes the replacement path distance when it is removed. The following data is computed during preprocessing, where $x, y \in V, e \in E$ and $i \in \{1, 2, \ldots 2 \log n\}$:

- $CR[x, y, i]$: first center of priority-$\geq i$ on x-y shortest path.
- $CL[x, y, i]$: first center of priority-$\geq i$ on reversed y-x shortest path (edge directions flipped).
- $BCP[x, y]$: biggest center priority on x-y shortest path.
- $D_i[x, y, e]$: distance $d(x, y, e)$, where x is a center of priority i that covers e.
- $BV[x, y, i]$: bottleneck edge of interval i on x-y shortest path.
- $DBV[x, y, i]$: distance $d(x, y, BV[x, y, i])$, for each interval i.

This preprocessing data takes up a total of $\tilde{O}(n^2)$ space. After preprocessing, a query can be answered by looking up $O(1)$ values, as shown in Algorithm 1.

Algorithm 1 Framework from [2] for DSO with $O(1)$ query.

Input: Directed weighted graph $G = (V, E)$.

1: **Preprocessing Algorithm** :
2: Every vertex is randomly assigned a priority k with probability $\Theta(\frac{1}{2^k})$, for $k = 1, 2, \ldots \log n$.
3: For each $x, y \in V$, $i \leq \log n$, compute $CR[x, y, i], CL[x, y, i], BCP[x, y]$, .
4: For each $x \in V$ and edge e covered by x, for each $y \in V$, compute $D_i[x, y, e]$ (i = priority of x).
5: For each $x, y \in V$, $i \leq \log n$, compute bottleneck vertex $BV[x, y, i]$ using binary search and an RMQ data structure [1] on the computed $D_i[x, y, e]$ values.
6: For each $x, y \in V$, $i \leq \log n$, compute distance $DBV[x, y, i]$ when vertex $BV[x, y, i]$ is removed.
7: **Query Algorithm**, given query $d(x, y, e)$:
8: Let edge $e = (u, v)$. Let $i = BCP[x, u]$, $j = BCP[v, y]$.
9: Compute $c_x = CL[x, y, i]$ and $c_y = CR[x, y, j]$. ▷ *Edge e is covered by c_x and c_y*
10: Output $d(x, y, e) = \min\{d(x, c_x) + D_i[c_x, y, e], D_i[x, c_y, v] + d(c_y, y), DBV[x, y, i]\}$

In the next two sections, we present PRAM algorithms that use this framework, implementing the steps using different methods. All our algorithms finally compute the same preprocessing data as [2], so the oracle can be stored in $\tilde{O}(n^2)$ space and can answer a query in $O(1)$ work on a single processor. In Section 3 we show how to directly implement Algorithm 1 efficiently on the PRAM, to obtain a work-efficient preprocessing algorithm that runs is $\tilde{O}(n^{1/2+o(1)})$ parallel time. In Section 4, we use this framework in addition to hop-limited DSO construction techniques to obtain faster preprocessing algorithms.

3 Work-Efficient PRAM DSO

In this section, we prove Theorem 1.1 by presenting PRAM algorithm DSO-A which implements Algorithm 1. DSO-A preprocesses the input graph in $\tilde{O}(mn)$ work and $\tilde{O}(n^{1/2+o(1)})$ parallel time. The constructed oracle has $\tilde{O}(n^2)$ size and can answer any query in $O(1)$ work. This algorithm uses excluded shortest paths computations, where given a graph $G = (V, E)$, set of edges $P \subseteq E$ and a source $x \in V$, we need to compute distances $d(x, y, e)$ for all $y \in V, e \in P$. If the set P is *independent*, i.e., the edges are in the outgoing shortest path tree of x and the subtrees rooted at any pair of edges in P are disjoint, then we can compute all excluded shortest paths distances using one SSSP computation [4].

Proof of Theorem 1.1. We show how we implement Algorithm 1 on the PRAM. The computation in line 3 is implemented using $O(n \log n)$ SSSPs as follows. For each vertex $x \in V$ and priority $1 \leq i \leq \log n$, we can readily modify an SSSP computation from x to compute the relevant center information $CR[x, y, i], BCP[x, y]$. We repeat this on the reversed graph for $CL[x, y, i]$.

To implement line 4, note that each center x of priority k covers edges within $\tilde{O}(2^k)$ depth in its outgoing shortest path tree w.h.p. in n. The set of edges at a given depth j in the outgoing shortest path tree of x is independent, so we use one excluded shortest paths computation for these edges. For depths $1 \leq j \leq \tilde{O}(2^k)$, we use $\tilde{O}(2^k)$ exclude computations, and each exclude involves one SSSP computation. Since we have $\tilde{O}(n/2^k)$ vertices of priority k, and $O(\log n)$ priorities, we get a total of $\tilde{O}(n)$ SSSP computations over all $x \in V$ w.h.p. in n. Finding bottleneck vertices in line 5 uses a range minimum query (RMQ) data structure, which is implemented on the PRAM with linear work and $O(\log^* n)$ parallel time for preprocessing and $O(1)$ work per query [1].

To implement line 6, we perform $\tilde{O}(n)$ SSSP computations as follows. The algorithm of [2] constructs a graph with vertices (x, y, i) for each interval i on x-y shortest path for $x, y \in V, i \leq \log n$, and an additional vertex s. The edges are between (x, y, i) to (x, y', j) with weight $w(y', y)$ for each edge $(y', y) \in E$, and additional edges from s to (x, y, i). Then, they directly compute SSSP on this graph. We instead construct n graphs G_x, one for each $x \in V$, with vertex s and vertices $(x, y, i), \forall y \in V, i \leq \log n$. Since there are no edges between different G_x, G_y, we can compute SSSP separately in each G_x, which has $\tilde{O}(n)$ vertices and $\tilde{O}(m + n)$ edges. This computes $DBV[x, y, i]$ as the shortest distance from s to vertex (x, y, i) in G_x.

In total, we perform $\tilde{O}(n)$ SSSP computations, which takes a total of $\tilde{O}(mn)$ work and $\tilde{O}(n^{1/2+o(1)})$ time using the PRAM algorithm of [3]. Line 5 takes $\tilde{O}(n^2)$ work and $\tilde{O}(1)$ time, and other lines take $O(n)$ work. The correctness of our algorithm readily follows from the correctness of [2], and the oracle has the same $\tilde{O}(n^2)$ size as the construction of [2]. The query algorithm is the same as in Algorithm 1, which takes $O(1)$ work per query. □

4 Faster PRAM DSO for Dense Graphs

In this section, we present PRAM algorithms that can beat the $\tilde{O}(n^{1/2+o(1)})$ parallel time of the preprocessing algorithm presented in Section 3. Our method constructs a series of hop limited DSOs in order to ultimately obtain a complete DSO. We first describe the hop limited DSO constructions, and then show how we use it in our algorithms.

4.1 Hop-limited DSO

Let $d_h(x, y, e)$, for any $1 \leq h \leq n$, denote the minimum weight path among all x-y paths of at most h edges not containing edge e. In an h-hop limited DSO we needed to preprocess an input graph $G = (V, E)$ to answer queries $d_h(x, y, e)$ for any $x, y \in V, e \in E$. Note that an n-DSO is simply a complete DSO. In this section, we describe a two-step DSO extension procedure to construct $\frac{3}{2}h$-hop limited DSO from an h-hop limited DSO, adapting a sequential technique of Ren [8].

Lemma 4.1. *Let $1 \leq h \leq n$. Given an h-hop DSO with preprocessing cost P_w work and P_t parallel time ($P_t \leq n$) which answers a query in $\tilde{O}(1)$ work, we can construct a $\frac{3}{2}h$-hop DSO with preprocessing cost $P_w + \tilde{O}(mn + (n^3/h) + (n/P_t)^3)$ work and $P_t + \tilde{O}(1)$ parallel time, which can answer any query in $O(1)$ work.*

Proof. The two-step procedure is as follows: The first step, proven in Lemma 4.2 of Section 4.1.1, constructs a $\frac{3}{2}h$-hop DSO with preprocessing cost of $P_w + O(n)$ and $P_t + O(1)$ parallel time and cost per query of $\tilde{O}(n/h)$ work and $\tilde{O}(1)$ parallel time.

The second step, proven in Lemma 4.3 of Section 4.1.2, reduces the query cost to $O(1)$ work. We construct a $\frac{3}{2}h$-hop DSO with $O(1)$ work per query and preprocessing cost of $P_w + \tilde{O}(mn + (n^3/h) + (n/P_t)^3)$ work and $P_t + \tilde{O}(1)$ time. □

4.1.1 Extended hop DSO with high query time. Assume that we are given a h-hop limited DSO with $O(1)$ query time. The following observation is from [8]: Let s be a vertex on a $\frac{3}{2}h$-hop replacement path from u to v such that s is at most h-hops from both u and v along this path. Then, $d_{\frac{3}{2}h}(u, v, e) = d_h(u, s, e) + d_h(s, v, e)$, as the two subpaths u-s and s-v have distances $d_h(u, s, e)$ and $d_h(s, v, e)$ respectively. We use this result in the following lemma.

Lemma 4.2. *Given an h-hop DSO with P_w work and P_t parallel time for preprocessing and $\tilde{O}(1)$ work for query, we can construct a $\frac{3}{2}h$-hop DSO with preprocessing cost $P_w + O(n)$ work and $P_t + O(1)$ time, which answers a query in $\tilde{O}(n/h)$ work and $\tilde{O}(1)$ time.*

Proof. Sample each vertex into a set S with probability $\Theta(\log n/h)$ so that any path of length $\frac{h}{2}$ is hit by a sampled vertex w.h.p in n. Note that $|S| = \tilde{O}(n/h)$ w.h.p. in n. To answer a query, we compute $d_{\frac{3}{2}h}(u, v, e) = \min_{s \in S} d_h(u, s, e) + d_h(s, v, e)$. This is correct w.h.p. in n: The path with distance $d_h(u, s, e) + d_h(s, v, e)$ is a valid u-v path not containing e for any $s \in V$. Consider the subpath of the $\frac{3}{2}h$-hop replacement path containing vertices within h hops of both u and v. This segment has length of at least $\frac{h}{2}$ and therefore has a sampled vertex $s \in S$ w.h.p. in n. This s gives us the correct distance $d_{\frac{3}{2}h}(u, v, e)$ by the above observation.

The preprocessing algorithm of the $\frac{3}{2}h$-hop DSO is to first sample and store S, which takes $O(n)$ work and $O(1)$ time. Then, we run the preprocessing algorithm of the input h-hop DSO. A query computes the minimum over $O(|S|) = \tilde{O}(n/h)$ h-hop distances, which are computed using queries to the h-hop DSO. This takes $\tilde{O}(n/h)$ work and $\tilde{O}(1)$ time per $\frac{3}{2}h$-hop query. □

4.1.2 Reducing to $O(1)$ query time. Now, we convert a given h-hop DSO with high query time into an h-hop DSO with $O(1)$ query time using additional preprocessing by implementing Algorithm 1.

Lemma 4.3. *Given an h-hop DSO with P_w work and P_t parallel time for preprocessing and q work and $\tilde{O}(1)$ time for query, we can construct a $\frac{3}{2}h$-hop DSO with $P_w + \tilde{O}(mn + n^2 \cdot q + (n/P_t)^3)$ work and $P_t + \tilde{O}(1)$ time for preprocessing and $O(1)$ work for query.*

Proof. We show how we implement each preprocessing step of Algorithm 1, after which we can use the query algorithm described there. Line 5 is implemented in $\tilde{O}(n^2)$ work and $\tilde{O}(1)$ time as in Section 3. Lines 3,4,6 are implemented differently, as follows.

Line 3: To compute CR, CL and BCP values for each interval on the shortest path between each vertex pair $x, y \in V$, for a fixed i, we do a modified APSP. To compute $CR[x, y, i]$, we track the first center of priority $\geq i$ in addition to $d(x, y)$. When combining distances during APSP, we also update the CR value. This modification is readily done to the PRAM APSP algorithm in [5]. The values CL, BCP are computed in a similar way. For $1 \leq i \leq \log n$, we perform $O(\log n)$ APSP computations which takes $\tilde{O}(mn + (n/P_t)^3)$ work and parallel time P_t using the algorithm of [5].

Lines 4,6: These lines require computing h-hop replacement distances, for which we use queries to the input h-hop DSO with q work per query. We run the preprocessing algorithm of the given DSO, in parallel to line 3. Then, in line 4, we compute distance $d(x, y, e)$ for each edge e on the x-y shortest path that is covered by center x. The number of such distances for a fixed vertex y is $\leq 2n$ as each edge is covered by the endpoint centers of the interval containing it. This is a total of $O(n^2)$ distances for all $y \in V$. Line 6 thus computes $\tilde{O}(n^2)$ distances. So, we have an additional preprocessing cost of $\tilde{O}(n^2 q)$ work and $\tilde{O}(1)$ time. □

4.2 Faster DSO constructions for Dense Graphs

4.2.1 Preprocessing algorithm for DSO in $\tilde{O}(n^3)$ work and $\tilde{O}(1)$ time. In this section, we present algorithm DSO-B that takes $\tilde{O}(n^3)$ work and $\tilde{O}(1)$ parallel time to construct an oracle of size $\tilde{O}(n^2)$, that can answer a query in $O(1)$ time. We first construct a 2-hop limited DSO with high query time and apply Lemma 4.3 to obtain a 2-hop DSO with $O(1)$ query time. Then, we repeatedly apply Lemma 4.1 for $O(\log n)$ steps until we obtain a n-hop DSO – which is just a complete DSO.

Base Case: Our base DSO is a 2-hop limited DSO, with a query cost of $O(n)$ work, and $\tilde{O}(1)$ parallel time. This needs no preprocessing, as $d_2(x, y, e) = \min_{s \in V} w(x, s) + w(s, y)$, where the minimum is over $s \in V$ such that $(x, s), (s, y) \in E$ and $e \neq (x, s), (s, y)$. We now apply Lemma 4.3 to reduce this query time, with parameters $P_t = \tilde{O}(1), q = n$, to obtain a 2-hop DSO with preprocessing cost of $\tilde{O}(n^3)$ work and $\tilde{O}(1)$ parallel time and $O(1)$ work query.

Obtaining complete DSO: We apply Lemma 4.1 with $P_w = \tilde{O}(n^3)$ and $P_t = \tilde{O}(1)$. We repeat this procedure $O(\log n)$ times until we obtain a n-hop DSO with $O(1)$ query. Over all $O(\log n)$ steps, we get a total of $\tilde{O}(n^3)$ work and $\tilde{O}(1)$ parallel time.

Note that the final query is just looking up $O(1)$ values precomputed by the n-hop DSO. Thus, we only maintain preprocessed data for the n-hop DSO, which has size $\tilde{O}(n^2)$, and the intermediate DSOs can be discarded.

4.2.2 DSO with improved work-time tradeoff. In this section, we present algorithm DSO-C that performs $\tilde{O}(mnh + (n^3/h))$ work and $\tilde{O}(h)$ parallel time for preprocessing, for any $2 \leq h \leq n$, to construct an oracle of size $\tilde{O}(n^2)$ that answers a query in $O(1)$ work. Our preprocessing algorithm follows a scheme similar to DSO-B in the previous section, but uses a different base case.

Base Case: We present a method to construct an h-hop DSO using a sequential graph sampling method of [9]. We sample $\tilde{O}(h)$ graphs G_i by removing every edge with probability $(1/h)$, independently at random. In [9], it is proven that for any path $\mathcal{P}$ of h hops not containing edge e, w.h.p. in n, there is at least one graph G_i that includes all edges of $\mathcal{P}$ but not e. Additionally for a specific edge e, w.h.p. in n, there are only $O(\log n)$ graphs G_i that do not contain e. If $d_i(x, y)$ is the shortest path distance in G_i, then $d(x, y, e) = \min_{i: e \notin G_i} d_i(x, y)$, w.h.p. in n.

To construct an h-hop DSO, we sample $\tilde{O}(h)$ graphs as described above and then compute h-hop limited APSP distances in each sampled graph G_i. We compute h-hop SSSP from each vertex of G_i in $\tilde{O}(mn)$ work and $\tilde{O}(h)$ parallel time using an $\tilde{O}(m)$-work $\tilde{O}(h)$-time hop-limited shortest path algorithm (Lemma 5.3 of [3]). We also store the identities of the set of graphs G_i that do not contain e, for each edge $e \in E$. Thus, after preprocessing, we can answer a query in $O(\log n)$ time using the precomputed $d_i(x, y)$ distances. Since we have $\tilde{O}(h)$ graphs, we obtain a DSO with preprocessing cost of $\tilde{O}(mnh)$ work and $\tilde{O}(h)$ parallel time, with $\tilde{O}(1)$ work query.

Obtaining complete DSO: Similar to DSO-B, we repeatedly apply Lemma 4.1 $O(\log n)$ times to construct an n-hop DSO from an h-hop DSO. We use parameters $P_w = \tilde{O}(mnh)$ and $P_t = \tilde{O}(h)$. In an extension step where we construct a $\frac{3}{2}k$-hop DSO from a k-hop DSO for some $k \geq h$, we incur an additional preprocessing cost of $\tilde{O}(\frac{n^3}{k} + \frac{n^3}{k^3}) = \tilde{O}(\frac{n^3}{h})$ work and $\tilde{O}(h)$ parallel time. Adding up the costs for $O(\log n)$ steps, preprocessing cost is $\tilde{O}(mnh + \frac{n^3}{h})$ work and $\tilde{O}(h)$ parallel time. We only store preprocessing data for the final n-DSO, so the oracle has size $\tilde{O}(n^2)$.

5 Conclusion and Open Problems

We have presented the first non-trivial PRAM algorithms for DSO under single edge failure: We constructed a work-efficient algorithm for dense graphs with $\tilde{O}(n^3)$ work and $\tilde{O}(1)$ parallel time, matching APSP. We also presented a work-efficient algorithm for sparse graphs with preprocessing work $\tilde{O}(mn)$ matching sequential DSO [2] but with parallel time $\tilde{O}(n^{1/2+o(1)})$. While we have presented a work-time tradeoff algorithm that can achieve sub-$\sqrt{n}$ parallel time, it is not work-efficient. An open problem is whether there is a work-efficient DSO preprocessing algorithm with $\tilde{O}(mn)$ work and $\tilde{O}(n^{1/3})$ parallel time, matching the current best work-efficient PRAM algorithm for APSP [5]. A larger open problem is whether we can reduce the parallel time beyond $n^{1/3}$ for both APSP and DSO, while maintaining $\tilde{O}(mn)$ work.

Acknowledgments

This work was supported in part by NSF grant CCF-2008241.

References

[1] Omer Berkman and Uzi Vishkin. 1993. Recursive Star-Tree Parallel Data Structure. *SIAM J. Comput.* 22, 2 (1993), 221–242.

[2] Aaron Bernstein and David R. Karger. 2009. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009*. ACM, Bethesda, MD, USA, 101–110.

[3] Nairen Cao and Jeremy T. Fineman. 2023. Parallel Exact Shortest Paths in Almost Linear Work and Square Root Depth. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*. SIAM, Florence, Italy, 4354–4372.

[4] Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. 2008. Oracles for Distances Avoiding a Failed Node or Link. *SIAM J. Comput.* 37, 5 (2008), 1299–1318.

[5] Adam Karczmarz and Piotr Sankowski. 2021. A Deterministic Parallel APSP Algorithm and its Applications. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*. SIAM, Virtual Conference, 255–272.

[6] Philip N. Klein and Sairam Subramanian. 1997. A Randomized Parallel Algorithm for Single-Source Shortest Paths. *J. Algorithms* 25, 2 (1997), 205–220.

[7] Vignesh Manoharan and Vijaya Ramachandran. 2025. Distributed Distance Sensitivity Oracles. In *Structural Information and Communication Complexity - 32nd International Colloquium, SIROCCO 2025, Proceedings (LNCS, Vol. 15671)*. Springer, Delphi, Greece, 366–383.

[8] Hanlin Ren. 2022. Improved distance sensitivity oracles with subcubic preprocessing time. *J. Comput. Syst. Sci.* 123 (2022), 159–170.

[9] Oren Weimann and Raphael Yuster. 2013. Replacement Paths and Distance Sensitivity Oracles via Fast Matrix Multiplication. *ACM Trans. Algorithms* 9, 2 (2013), 14:1–14:13.