Connectivity Oracles for Predictable Vertex Failures

Bingbing Hu
Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
University of California San Diego, CA, USA

Evangelos Kosinas
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Adam Polak

Bocconi University, Milan, Italy

Abstract

The problem of designing connectivity oracles supporting vertex failures is one of the basic data structures problems for undirected graphs. It is already well understood: previous works [Duan–Pettie STOC'10; Long–Saranurak FOCS'22] achieve query time linear in the number of failed vertices, and it is conditionally optimal as long as we require preprocessing time polynomial in the size of the graph and update time polynomial in the number of failed vertices.

We revisit this problem in the paradigm of algorithms with predictions: we ask if the query time can be improved if the set of failed vertices can be predicted beforehand up to a small number of errors. More specifically, we design a data structure that, given a graph $G = (V, E)$ and a set of vertices predicted to fail $\widehat{D} \subseteq V$ of size $d = |\widehat{D}|$, preprocesses it in time $\tilde{O}(d|E|)$ and then can receive an update given as the symmetric difference between the predicted and the actual set of failed vertices $\widehat{D} \triangle D = (\widehat{D} \setminus D) \cup (D \setminus \widehat{D})$ of size $\eta = |\widehat{D} \triangle D|$, process it in time $\tilde{O}(\eta^4)$, and after that answer connectivity queries in $G \setminus D$ in time $O(\eta)$. Viewed from another perspective, our data structure provides an improvement over the state of the art for the *fully dynamic subgraph connectivity problem* in the *sensitivity setting* [Henzinger–Neumann ESA'16].

We argue that the preprocessing time and query time of our data structure are conditionally optimal under standard fine-grained complexity assumptions.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Data structures, graph connectivity, algorithms with predictions

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.72

Related Version *Full Version*: `https://arxiv.org/abs/2312.08489`

Acknowledgements Part of this work was done when Evangelos Kosinas was at University of Ioannina and Adam Polak was at Max Planck Institute for Informatics.

1 Introduction

Connectivity is a fundamental problem in undirected graphs. For a static input graph, one can compute its connected components in linear time; after such preprocessing it is easy to answer, in constant time, queries asking whether two given vertices u and v are connected.

The problem becomes much more challenging when the graph may change over time – for instance, when certain vertices may fail and have to be removed from the graph. Duan and Pettie [8] were the first to propose a data structure that, after constructing it for graph $G = (V, E)$, can receive an *update* consisting of a set of *failed* vertices $D \subseteq V$, and then can answer connectivity *queries* in $G \setminus D$ (i.e., in the subgraph of G induced on $V \setminus D$). For a

© Bingbing Hu, Evangelos Kosinas, and Adam Polak;
licensed under Creative Commons License CC-BY 4.0

graph with n vertices and m edges, the data structure can be constructed in $\tilde{O}(nm)$ time[1], the update takes time $\tilde{O}(d^6)$, where $d = |D|$, and each query runs in $O(d)$ time.[2] Under the Online Matrix-Vector Multiplication (OMv) Hypothesis, this linear query time is optimal (up to subpolynomial factors) if we require that preprocessing runs in time polynomial in the graph size and the update time is polynomial in the number of failed vertices [14].

We study the problem of designing such connectivity oracles through the new lens of *algorithms with predictions*. More specifically, we ask whether the query time can be improved, beyond the fine-grained lower bound, if some quite-accurate-but-not-perfect prediction of the set of failed vertices is available already during the preprocessing.

1.1 Our results

We define the problem of designing a connectivity oracle for predictable vertex failures as follows. We ask for a data structure that:

- in the *preprocessing phase*, receives an undirected graph $G = (V, E)$, an upper bound on the number of failed vertices $d \in \mathbb{Z}_+$, and a set $\widehat{D} \subseteq V$ of vertices predicted to fail ($|\widehat{D}| \leqslant d$);
- in the *update phase*, receives a set $D \subset V$ of vertices that actually fail ($|D| \leqslant d$);
- in the *query phase*, receives two vertices $u, v \in V \setminus D$, and answers whether u and v are connected in $G \setminus D \coloneqq G[V \setminus D]$.

We measure the prediction error as the size of the difference between the predicted and the actual set of failed vertices $\eta \coloneqq |\widehat{D} \triangle D| = |\widehat{D} \setminus D| + |D \setminus \widehat{D}|$. Our goal is to design a data structure that improves upon the running time of classic (prediction-less) data structures when $\eta \ll d$. To allow for update time sublinear in d, we let the set D be given not explicitly but as the symmetric difference $\widehat{D} \triangle D \coloneqq (\widehat{D} \setminus D) \cup (D \setminus \widehat{D})$ instead.

Our main result is such data structure with the following guarantees:

▶ **Theorem 1.** *There is a (deterministic) connectivity oracle for predictable vertex failures with $\tilde{O}(dm)$ preprocessing time and space, $\tilde{O}(\eta^4)$ update time, and $O(\eta)$ query time.*

Our result can also be interpreted outside of the realm of algorithms with predictions. One can think of $\widehat{D}$ as of the set of vertices that are initially *inactive*. Then, in the update phase, a small number η of vertices change their state – some inactive vertices become active and some active come to be inactive. With this perspective, our result can be interpreted as extending the type of the data structure of Duan and Pettie [8] with vertex *insertions*. Since we want to keep the update and query times depending only on the number of affected vertices we cannot even afford to read all the edges of an inserted vertex, which can potentially have a high degree; hence, we need to know candidates for insertion already in the preprocessing phase, and the initially inactive vertices are precisely these candidates.

The above perspective – of adding insertions to a Duan–Pettie-style oracle – actually gives a good intuition of how our data structure works internally: we create a (prediction-less) vertex-failure connectivity oracle for $G \setminus \widehat{D}$, we use it to further remove $D \setminus \widehat{D}$, and we extend it so that it is capable of reinserting $\widehat{D} \setminus D$. We found that the original connectivity oracle of Duan and Pettie [8], with its complex *high-degree hierarchy tree*, is not best suited for such an extension. Instead, we base our data structure on a recent DFS-tree decomposition scheme proposed by Kosinas [21].

---

[1] We use the $\tilde{O}$ notation to hide polylogarithmic $(\log n)^{O(1)}$ factors.
[2] Their data structure allows also for a certain trade-off between the preprocessing and update times.

Comparison to prior work. Henzinger and Neumann [15] studied already the problem of designing a data structure like ours, under the name of *fully dynamic subgraph connectivity* in the *sensitivity setting*. Their proposed solution is a reduction to a Duan–Pettie-style (i.e., any deletion-only) connectivity oracle. Plugging in the current best oracle of Long and Saranurak [26], the reduction gives $\hat{O}(d^3 m)$ preprocessing time[3], $\hat{O}(\eta^4)$ update time, and $O(\eta^2)$ query time. Because the oracle of [26] is (near-)optimal under plausible fine-grained complexity assumptions, these running times turn out to be (nearly) the best we can hope to get from that reduction.

Using algebraic techniques, van den Brand and Saranurak [36] tackle a more general problem of answering *reachability* queries in *directed* graphs in the presence of edge insertions and deletions. Note that in directed graphs edge updates can be used to simulate both vertex failures and activating (initially inactive) vertices. This is done by splitting each vertex into two – one only for incoming edges, one only for outgoing – and having a special edge from the former to the later whenever the original vertex is meant to be active. Therefore, the data structure that they provide also solves the problem that we study; it achieves $\hat{O}(n^\omega)$ preprocessing time, $\hat{O}(\eta^\omega)$ update time, and $O(\eta^2)$ query time, with high probability, where $\omega \leqslant 2.372$ denotes the matrix multiplication constant.

Our result offers a considerable improvement over the preprocessing and query times of both [15] and [36].

Lower bounds. We argue that the preprocessing and update times of our data structure are optimal under standard fine-grained complexity assumptions. For the query time, notice that a classic (prediction-less) vertex-failure connectivity oracle can be simulated by setting $\widehat{D} = \emptyset$ in the preprocessing; then $\eta = d$, and hence the lower bound of Henzinger et al. [14] implies that the query time $O(\eta^{1-\varepsilon})$ is impossible, for any $\varepsilon > 0$ (assuming the OMv Hypothesis and requiring that the preprocessing time is polynomial in the graph size and the update time is polynomial in d).

Regarding the preprocessing time, in Section 4 we give a lower bound based on the Exact Triangle Hypothesis[4], which is implied by both the 3SUM Hypothesis and the APSP Hypothesis, so it is a weaker assumption than either of those popular fine-grained complexity hypotheses. Our lower bound shows that the $\tilde{O}(dm)$ complexity is conditionally optimal, up to subpolynomial factors, as long as the update and query times depend polynomially only on η, and not on the graph size nor on d.

▶ **Theorem 2.** *Unless the Exact Triangle Hypothesis fails, there is no connectivity oracle for predictable vertex failures with preprocessing time $O(d^{1-\varepsilon} m)$ or $O(dm^{1-\varepsilon})$ and update and query times of the form $f(\eta) \cdot n^{o(1)}$, for any $\varepsilon > 0$.*

We note that Long and Saranurak [26] give a similar lower bound for connectivity oracles for vertex failures (without predictions), which is however based on the Boolean Matrix Multiplication Hypothesis, and hence it holds only against "combinatorial" algorithms.

1.2 Related work

Vertex-failure connectivity oracles. Duan and Pettie [8] were the first to study connectivity oracles supporting vertex failures, and they already achieved query time that turns out to be (conditionally) optimal [14]. Their preprocessing time and update time were subsequently

[3] The $\hat{O}$ notation hides subpolynomial $n^{o(1)}$ factors.
[4] The Exact Triangle Hypothesis says that there is $O(n^{3-\varepsilon})$ time algorithm for finding in an n-node edge-weighted graph a triangle whose edge weights sum up to 0, for any $\varepsilon > 0$.

improved [9, 10, 26] to $\tilde{O}(dm)$ and $\tilde{O}(d^2)$, respectively, which are optimal under standard fine-grained complexity assumptions [26]. Recently, Kosinas [21] gave a data structure with a slightly worse $\tilde{O}(d^4)$ update time, but much simpler than all the previous constructions.

On the other hand, Pilipczuk et al. [34] proposed a data structure that handles update and query together, in time doubly exponential in d but with no dependence (even logarithmic) on the graph size. Their approach is thus beneficial, for instance, when d is a constant.

Edge-failure connectivity oracles. Edge failures are much easier to handle than vertex failures. In particular, there exist edge-failure connectivity oracles using $\tilde{O}(m)$ space with $\tilde{O}(d)$ update time and $\tilde{O}(1)$ query time [32, 19, 10], which is *unconditionally* optimal (up to polylogarithmic factors). This difference can be explained by the fact that a single edge deletion can only split the graph into two parts while deleting a high-degree vertex can create multiple new connected components.

Fully dynamic subgraph connectivity. In the fully dynamic subgraph connectivity problem every update consists in the activation or deactivation of a vertex, and there is no bound on the number of vertices that may be active or inactive at any given point. In the meantime, there are queries that ask whether two active vertices are connected through a path that visits only active vertices. This model was introduced by Frigioni and Italiano [13] in the context of planar graphs, where they show that one can achieve polylogarithmic time per operation. For general undirected graphs, there are several solutions, deterministic and randomized, that provide a trade-off between the update and query time [5, 7, 11, 2]. In all those cases, both the update and the query time are $\Omega(m^\delta)$, for some $\delta > 0$, which is in accordance with known conditional lower bounds [1, 14, 18]. Thus, there is a clear distinction between the complexity in this unrestricted model, against that in the fault-tolerant (sensitivity) setting. In other words, we can achieve better performance if there is a bound on the number of vertices at any given time whose state may be different from their original one.

Fault-tolerant labelling schemes. There is a recent line of work that provides efficient labelling schemes for connectivity queries in the fault-tolerant setting [6, 30, 17, 31]. The problem here is to attach labels to the vertices, so that one can answer connectivity queries in the presence of failures given the labels of the query vertices and those of the failed vertices. The most significant complexity measures are the size of the labels (ideally, the number of bits per label should be polylogarithmic on the number of vertices), and the time to retrieve the answer from the labels. There are many problems that admit efficient labelling schemes (see, e.g., [33]). Such labellings have applications in the distributed setting, because they allow to compute the answer without having to access a centralized data structure. All else being equal, it is obviously a much harder problem to provide labels whose total size matches that of the best centralized data structures.

Algorithms with predictions. The field of algorithms with predictions, also known as learning-augmented algorithms, blossoms in the recent years, see surveys [28, 29] and an updated list of papers [24]. While the majority of those works concern online algorithms, data structures are also being studied in this paradigm, see, e.g., [22, 12, 23]. These are however mostly index data structures, BSTs, etc., and not graph data structures.

Probably the closest to our work are three recent papers about dynamic graph algorithms with predictions [25, 35, 16]. We note that [35] also manage to use predictions to go beyond OMv-based conditional lower bounds, but they achieve it using algebraic algorithms for fast matrix multiplication, while we only use simple combinatorial methods and benefit from the fact that the predicted information is available already during the preprocessing time.

Recent follow-up work. Following the initial submission of our manuscript and posting a preprint on arXiv, Long and Wang [27] improved over our update time, from $\tilde{O}(\eta^4)$ down to $\tilde{O}(\eta^2)$, which is conditionally optimal. Their data structure builds on the work of Long and Saranurak [26], and is arguably much more complex than ours. In particular, our bounds involve less logarithmic factors hidden in the $\tilde{O}$ notation.

2 Preliminaries

Let G denote the input graph with n vertices and m edges. Let $\widehat{D}$ be the set of vertices predicted to fail, and let D be the set of vertices that actually fail. Moreover, we let the size of both D and $\widehat{D}$ to be at most d, and we let the prediction error $\eta := |D \setminus \widehat{D}| + |\widehat{D} \setminus D|$.

Without loss of generality, we may assume that $G \setminus \widehat{D}$ is connected. Indeed, we may add a new auxiliary vertex z to G, that is connected with an edge with every vertex of the graph. Then, $G \setminus \widehat{D}$ is definitely connected, and in order to perform the updates, we always include z in the set D of failed vertices. Notice that this does not affect the asymptotic complexity of any of our measures of efficiency.

We build on the DFS-based tree decomposition scheme from [21]. Given the inputs G and $\widehat{D}$, we grow an arbitrary DFS tree T of $G \setminus \widehat{D}$ rooted at some vertex r. Let $T(v)$ denote the subtree of T rooted at v. Let $ND(v)$ denote the number of descendants of v, which is equal to the size of $T(v)$. Let $p_T(v)$ denote the parent of v in T.

Suppose we further remove some set of failed vertices $D \setminus \widehat{D}$ from T. Then T gets decomposed into multiple connected components, which can be however connected to each other through *back-edges* in $(G \setminus \widehat{D}) \setminus (D \setminus \widehat{D})$. We use r_C to denote the root of each connected component C. The presence of back-edges is important in that they establish connectivity between the otherwise loose connected components of $T \setminus (D \setminus \widehat{D})$, so connectivity queries between two vertices can be reduced to queries between the connected component(s) that they reside in. Let $low(v)$ denote the lowest proper ancestor of v that is connected to $T(v)$ through a back-edge. It is useful to extend the definition of $low(v)$ to $low_k(v)$, as is introduced in [21, Section 2.1]. We identify $low_1(v)$ with $low(v)$. Then for every $k > 1$, we define $low_k(v)$ to be the next lowest ancestor of v connected to $T(v)$ through a back-edge that is higher than $low_{k-1}(v)$.

It is important to distinguish between two types of connected components: *hanging subtree* and *internal component*, which are introduced in [21]. A connected component C of $T \setminus (D \setminus \widehat{D})$ is called a *hanging subtree* if none of the failed vertices from $D \setminus \widehat{D}$ is a descendant of it. Otherwise, C is called an *internal component*. (See [21, Section 3.3] for more properties of these two types of components.) For a visual illustration of the decomposition, see the left hand side of Figure 1. After the black vertices are removed, the white arrows represent internal components, and the grey triangles represent hanging subtrees. The reason for this distinction is that while the number of hanging subtrees can be close to n, the number of internal components is bounded by the number of failed vertices, which is a convenient fact that we leverage in the update phase. If C is an internal component and $f \in D \setminus \widehat{D}$ is a failed vertex whose parent lies in C, then we call f a *boundary vertex of* C. The set of all boundary vertices of C is denoted $\partial(C)$. For instance, in Figure 1, C_1 has two boundary vertices and C_3 has only one boundary vertex.

We also add structure to the vertices in $D \setminus \widehat{D}$ by putting them in a *failed vertex forest* F (see [21, Section 3.3]). Basically each failed vertex $f \in D \setminus \widehat{D}$ has a parent pointer $parent_F(f)$ to the largest (w.r.t. the DFS numbering) failed vertex that is an ancestor of f. Each f also has a pointer to the list of its children in the forest. This forest can be built in $O(\eta^2)$ time. To check whether a failed vertex f is a boundary vertex of some internal component C, we can check whether $parent_F(f) \neq p_T(f)$ [21, Lemma 5].

3 Our data structure

Our oracle operates in three phases: preprocessing, update, and query.

Preprocessing. We begin with a DFS tree T for the graph $G \setminus \widehat{D}$, and prepare relevant data structures, which are detailed in Section 3.1.

Update. In this phase, we are concerned with further removing the set of vertices $D \setminus \widehat{D}$ from $G \setminus \widehat{D}$, and adding back the vertices $\widehat{D} \setminus D$ that were deleted in the previous phase because of prediction errors. We build an auxiliary *connectivity graph* $\mathcal{M}$, which captures connectivity relations among so called *internal components* of $T \setminus (D \setminus \widehat{D})$ and the reinserted vertices $\widehat{D} \setminus D$.

Query. Given two vertices $s, t \in G \setminus D$, we find a representative connected component for each of s and t, and answer the connectivity query based on $\mathcal{M}$.

3.1 The preprocessing phase

In the preprocessing phase, we build a number of data structures around $G \setminus \widehat{D}$. It takes $\tilde{O}(dm)$ time. Our preprocessing data structures expand on the data structures introduced in [21] (the first six in the list below). See Section 3.1 in [21] for more explanation and ways to construct them.

1. A DFS-tree T for $G \setminus \widehat{D}$ rooted at some vertex r. For each vertex v in $G \setminus \widehat{D}$, we compute its depth in T and number of descendants $ND(v)$. We store the DFS numbering and identify the vertices with the order in which they are visited.
2. A level-ancestor data structure on T [3]. For any vertex $v \in T$ and any depth $\ell \in \mathbb{Z}_+$, this data structure can return in constant time the ancestor of v at depth ℓ in the tree.
3. A 2D-range-emptiness data structure [4] that can answer, in $\tilde{O}(1)$ time, whether there is a back-edge with one end in some segment[5] $[X_1, X_2]$ of T and the other end in some segment $[Y_1, Y_2]$ of T.
4. The low_i points of all vertices in T, for $i \in \{1, \ldots, d\}$.
5. For every $i \in \{1, \ldots, d\}$, we first sort the children of each vertex v in T in increasing order with respect to their low_i points. Then we compute a new DFS numbering, which we denote T_i, corresponding to the DFS traversal in which children of every vertex are visited in that sorted order. Note that the ancestor-descendant relation in T_i is the same as that in T.
6. For every T_i, a separate 2D-range-emptiness data structure described in item 3.
7. For every $u \in \widehat{D}$, we traverse the vertices in T in a bottom-up fashion and mark vertices that have in their subtrees a neighbor of u. Then we create a tree T_u as follows: for every vertex v in T, rearrange the children of v by putting unmarked vertices before marked vertices, and do a DFS traversal under this rearrangement to get T_u. Note that the ancestor–descendant relation in T_u is the same as that in T.
8. For every $u \in \widehat{D}$, a 2D-range-emptiness data structure on T_u as described in item 3.

[5] When we talk about *segments* of a DFS tree T, we mean that we identify vertices w.r.t. their DFS numbering, and a segment $[X, Y]$ of T is the set of vertices $\{X, X + 1, \ldots, Y\}$.

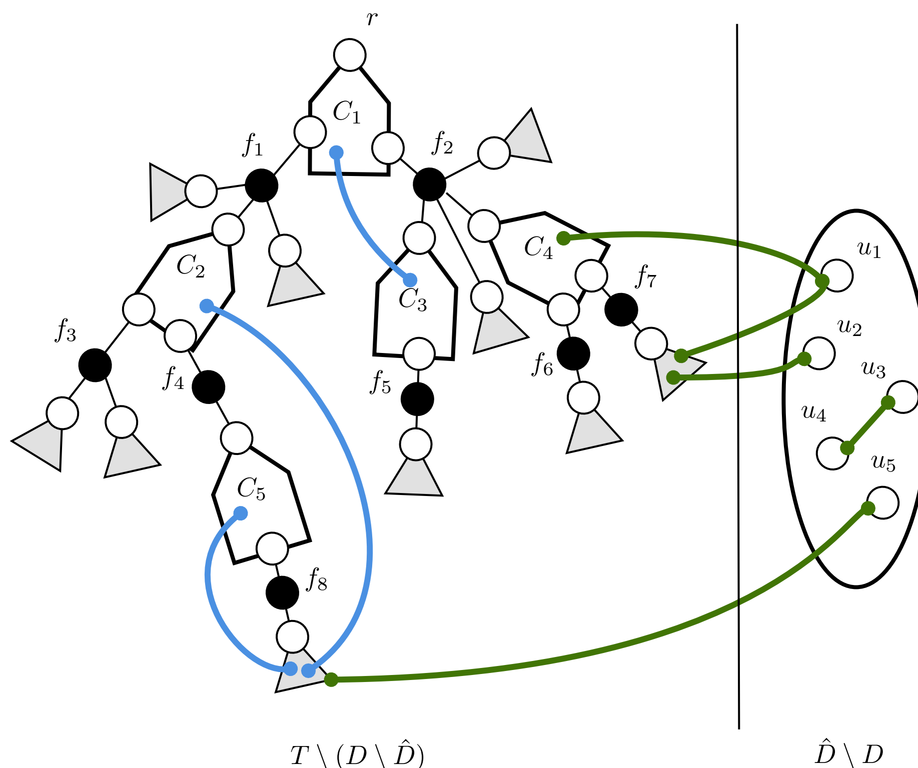

Figure 1 The decomposition graph after we further remove $D \setminus \widehat{D}$ from T and add back $\widehat{D} \setminus D$. On the left hand side we have $T \setminus (D \setminus \widehat{D})$. The black vertices $f_1, \ldots, f_8$ are failed vertices in $D \setminus \widehat{D}$. The white pentagonal arrows represent internal components. The light grey triangles represent hanging subtrees. The blue edges represent back-edges between connected components of $T \setminus (D \setminus \widehat{D})$. On the right hand side we have $\widehat{D} \setminus D$. The edges with an end in $\widehat{D} \setminus D$ are colored green.

9. For every $u, v \in \widehat{D}, u \neq v$, we sort the adjacent vertices of v in $G \setminus \widehat{D}$ in increasing order with respect to their numbering in T_u and store them as $neighbors_u(v)$. Moreover, in $neighbors(v)$ we store the adjacent vertices of v in $G \setminus \widehat{D}$ in increasing order w.r.t. their original DFS numbering in T.

▶ Remark 3. The purpose of items 1–9 will become clear when we describe how to add edges to the auxiliary connectivity graph $\mathcal{M}$ in the update phase. Items 1–6 allow us to efficiently establish connectivity within $(G \setminus \widehat{D}) \setminus (D \setminus \widehat{D})$. Items 7–9 allow us to efficiently establish connectivity between $\widehat{D} \setminus D$ and $(G \setminus \widehat{D}) \setminus (D \setminus \widehat{D})$.

The total time for initializing items 1–6 is $\tilde{O}(dm)$. See [21, Section 3.1] for more explanation. Items 7, 8, and 9 take $\tilde{O}(dm)$ time because for every $u \in \widehat{D}$, we create a new tree by taking the old tree T and rearranging its children lists in $\tilde{O}(m)$ time. Hence, the total preprocessing time is $\tilde{O}(dm)$, and the created data structures take $\tilde{O}(dm)$ space.

3.2 The update phase

In the update phase, we receive the real set of failed vertices D. Since $\widehat{D}$ has already been removed from G before computing our DFS tree T, we just need to further remove $D \setminus \widehat{D}$ from T, and then add back $\widehat{D} \setminus D$, which we needlessly removed (because of prediction errors) in the preprocessing phase. See Figure 1 for the decomposition of G after these operations have been performed.

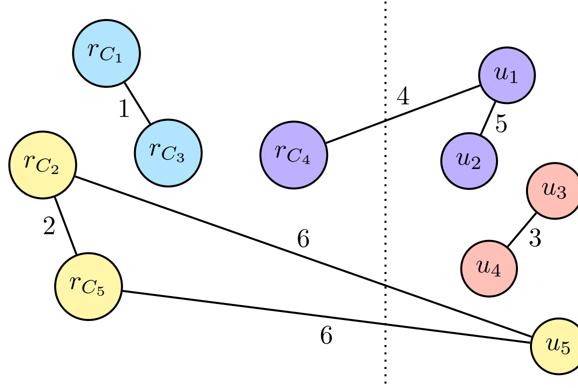

Figure 2 The connectivity graph $\mathcal{M}$ for the example shown in Figure 2. Edges are labeled with their types. The left hand side of the dotted line are vertices representing the internal components of $T \setminus (D \setminus \widehat{D})$; the right hand side of the dotted line are vertices in $\widehat{D} \setminus D$. Edge (r_{C_1}, r_{C_3}) is due to a back-edge between C_1 and C_3. Edges $(r_{C_2}, r_{C_5}), (r_{C_2}, u_5), (r_{C_5}, u_5), (u_1, u_2)$ are all due to mutual connections to a hanging subtree. Edges $(r_{C_4}, u_1), (u_3, u_4)$ are both due to direct edges. Vertices in the same connected component are filled with the same color.

An issue with the full decomposition graph is that the number of hanging subtrees can be as large as order of n. We want to "shrink" the decomposition graph by getting rid of the hanging subtrees while still preserving connectivity. For this purpose, we define an auxiliary *connectivity graph*, which builds on and extends a similar construction already introduced in [21].

▶ **Definition 4.** *Let $\mathcal{M}$ be a graph with the vertex set*

$$V(\mathcal{M}) := (\widehat{D} \setminus D) \cup \{r_C : C \text{ is an internal component of } T \setminus (D \setminus \widehat{D})\}.$$

The edge set $E(\mathcal{M})$ consists of edges of the following six types (the first two types are the same as in [21]).

1. *If there is a back-edge connecting two internal components C_1, C_2 of $T \setminus (D \setminus \widehat{D})$, then we add an edge between r_{C_1} and r_{C_2} in $\mathcal{M}$.*
2. *If there is a hanging subtree H of $T \setminus (D \setminus \widehat{D})$ which is connected to internal components $C_1, \ldots, C_k$ through back-edges, with C_k being an ancestor of all $C_1, \ldots, C_{k-1}$, then we add edges $(r_{C_k}, r_{C_1}), \ldots, (r_{C_k}, r_{C_{k-1}})$ in $\mathcal{M}$.*
3. *If there is an edge connecting two vertices $u_1, u_2 \in \widehat{D} \setminus D$ in G, then we also add an edge between u_1 and u_2 in $\mathcal{M}$.*
4. *If there is an edge connecting an internal component C of $T \setminus (D \setminus \widehat{D})$ and a vertex $u \in \widehat{D} \setminus D$, then we add an edge between r_C and u in $\mathcal{M}$.*
5. *If there are two vertices $u_1, u_2 \in \widehat{D} \setminus D$ and a hanging subtree H of $T \setminus (D \setminus \widehat{D})$ such that both u_1 and u_2 are connected to H, then we add an edge between u_1 and u_2 in $\mathcal{M}$.*
6. *If there are an internal component C of $T \setminus (D \setminus \widehat{D})$, a vertex $u \in \widehat{D} \setminus D$, and a hanging subtree H of $T \setminus (D \setminus \widehat{D})$ such that there is a back-edge connecting C and H, and there is an edge connecting u and H, then we add an edge between r_C and u in $\mathcal{M}$.*

As an illustration, in Figure 2 we show the connectivity graph for the graph in Figure 1.

The validity of type-2 edges comes from the following property of back-edges: If e is a back-edge of $T \setminus (D \setminus \widehat{D})$, then we have that either (i) the two ends of e are in two internal components that are related as ancestor and descendant, or (ii) one end of e is in a hanging

subtree H and the other end lies in an internal component that is an ancestor of H (see Lemma 6 and Corollary 7 in [21] for a full proof). Hence, the internal components $C_1, \ldots, C_k$ in the characterization of type-2 edges are all ancestors of H, so it suffices to pick the most ancestral internal component C_k and connect it by type-2 edges with the remaining components $C_1, \ldots, C_{k-1}$.

In the next lemma we show that $\mathcal{M}$ correctly captures the connectivity between internal components of $T \setminus (D \setminus \widehat{D})$ and $\widehat{D} \setminus D$ in $G \setminus D$.

▶ **Lemma 5.** *Let S and S' each be an internal component of $T \setminus (D \setminus \widehat{D})$ or a vertex in $\widehat{D} \setminus D$. Then S and S' are connected in $G \setminus D$ if and only if the vertices representing them in $\mathcal{M}$, $S_{\mathcal{M}}$ and $S'_{\mathcal{M}}$, are connected in $\mathcal{M}$.*

Proof. First, let us show the "only if" direction. Let $S = P_0 - P_1 - P_2 - \cdots - P_{k-1} - P_k = S'$ denote a path from S to S' in $G \setminus D$, where each P_i is either an internal component of $T \setminus (D \setminus \widehat{D})$, a hanging subtree of $T \setminus (D \setminus \widehat{D})$, or a vertex in $\widehat{D} \setminus D$. Note that by the property of back-edges above, we cannot have two consecutive hanging subtrees in the path.

Let us consider a segment $(P_i - P_{i+1})$ of this path. If neither P_i nor P_{i+1} is a hanging subtree, then we automatically have that their representatives in $\mathcal{M}$ are connected, by one of the constructions of type-1, type-3, or type-4 edges.

However, if one of P_i and P_{i+1} is a hanging subtree, and without loss of generality we assume it is P_i, then it follows that P_{i-1} exists and it is not a hanging subtree. If P_{i-1} and P_{i+1} are both internal components, then their representatives in $\mathcal{M}$ are (perhaps indirectly) connected by type-2 edges. If P_{i-1} and P_{i+1} are both vertices in $\widehat{D} \setminus D$, then their representatives are connected in $\mathcal{M}$ by a type-5 edge. If one of them is an internal component and the other is a vertex in $\widehat{D} \setminus D$, then their representatives are connected in $\mathcal{M}$ by a type-6 edge. Therefore, by applying the connectivity to all segments of the path, we get that the representatives of S and S' in $\mathcal{M}$ are connected, as desired.

Now, let us show the more straightforward "if" direction. Suppose $S_{\mathcal{M}}$ and $S'_{\mathcal{M}}$ are connected in $\mathcal{M}$. Then S and S' are either directly connected with an edge in $G \setminus D$, or indirectly connected through a hanging subtree in $G \setminus D$. In either case, we have that S and S' are connected. ◀

Note that only vertices from $\widehat{D} \setminus D$ and the roots of internal components of $T \setminus (D \setminus \widehat{D})$ are present in $V(\mathcal{M})$. Hence, the number of vertices in $\mathcal{M}$ is $O(\eta)$. We argue that we can add in type-1 to type-6 edges in $\tilde{O}(\eta^4)$ time. Proposition 11 and Proposition 12 in [21] describe how we can compute type-1 edges in $\tilde{O}(\eta^2)$ time and type-2 edges in $\tilde{O}(\eta^4)$ time. Computing type-3 edges is also straightforward. Hence, we focus on computing type-4, type-5, and type-6 edges in the following sections. We finish the update phase by computing connected components of $\mathcal{M}$, in time $O(|V(\mathcal{M})| + |E(\mathcal{M})|) = O(\eta^2)$, so that in the query phase we are able to decide in constant time whether two vertices in $V(\mathcal{M})$ are connected.

3.2.1 Computing type-4 edges

In this section we describe our algorithm for adding type-4 edges to $\mathcal{M}$, see Algorithm 1. Recall that type-4 edges connect vertices from $\widehat{D} \setminus D$ with internal components in which they have neighbors.

For each vertex $u \in \widehat{D} \setminus D$ and each internal component C of $T \setminus (D \setminus \widehat{D})$ we want to efficiently establish if there is an edge between them. Let $f_1, f_2, \ldots, f_k$ be the boundary vertices of C. By Lemma 4(3) of [21], C can be written as the union of the following segments: $[r_C, f_1 - 1], [f_1 + ND(f_1), f_2 - 1], \ldots, [f_{k-1} + ND(f_{k-1}), f_k - 1], [f_k + ND(f_k), r_C + ND(r_C) - 1]$.

For each of these segments we check whether $neighbors(u)$ contains a vertex with the DFS number in that segment and if it does we add an edge between u and r_C in $\mathcal{M}$. Each such check can be performed efficiently, in time $O(\log(|neighbors(u)|)) = O(\log n)$, by using binary search. Indeed, for a segment $[L, R]$, we first binary search in $neighbors(u)$ the smallest number greater than or equal to L, and then just check if it is less than or equal to R.

Algorithm 1 For each internal component C of $T \setminus (D \setminus \widehat{D})$ and $u \in \widehat{D} \setminus D$, adds a type-4 edge between r_C and u in $\mathcal{M}$ if there is an edge in G connecting u to C.

```
1 foreach u ∈ D̂ \ D do
2     foreach internal component C of T \ (D \ D̂) do
3         f_1, f_2, ..., f_k ← boundary vertices of C sorted in increasing order;
4         for i = 0, ..., k do
5             if i > 0 then L ← f_i + ND(f_i) else L ← r_C;
6             if i < k then R ← f_{i+1} − 1 else R ← r_C + ND(r_C) − 1;
              // Check the condition below in O(log n) time using binary
                 search.
7             if neighbors(u) ∩ [L, R] ≠ ∅ then
8                 add an edge between r_C and u in M;
```

For each internal component C, the number of binary searches we make is $O(|\partial(C)|)$. Since the total number of boundary vertices is $O(\eta)$, we have that the total number of binary searches in Algorithm 1 is $O(\eta^2)$, and hence its running time is $O(\eta^2 \log n)$.

3.2.2 Computing type-5 edges

In this section we describe our algorithm for adding type-5 edges to $\mathcal{M}$, see Algorithm 2. Recall that type-5 edges join pairs of vertices in $\widehat{D} \setminus D$ that are connected in $G \setminus D$ via a hanging subtree.

For every $u, v \in \widehat{D} \setminus D$, we want to know whether they are connected to the same hanging subtree H. If we checked this separately for every tuple (u, v, H), then it would be inefficient because the number of hanging subtrees can be as large as order of n. However, if we group together vertices that have edges to u in their subtrees, which is exactly what we do in T_u, then we can process the hanging subtrees in batches. Let f be a failed vertex in $D \setminus \widehat{D}$, and let $L_u(f)$ be the sequence of children of f that are marked by u, sorted by their DFS numbering in T_u. Since we only care about hanging subtrees, we disregard vertices in $L_u(f)$ which are roots of internal components. This breaks $L_u(f)$ into $O(\eta)$ *slices* consisting of roots of hanging subtrees. Each such slice $s = (L, \dots, R) \in S_u(f)$ corresponds to (potentially) multiple hanging subtrees – each subtree containing a neighbor of u – which form a contiguous set of vertices $[L, R + ND(R) - 1]$ in the DFS numbering for T_u. We can efficiently check if v has a neighbor with the DFS number in that range. Indeed, we binary search in $neighbors_u(v)$ the smallest number greater than or equal to L, and check if it is less than or equal to $R + ND(R) - 1$. If this is the case, then u and v both have edges to a common hanging subtree, so we add a type-5 edge between them.

We claim that the number of binary searches we make in line 8 of Algorithm 2 is $O(\eta^3)$. Each of them corresponds to a tuple (u, v, f, s). We leverage the fact that the number of internal components in $T \setminus (D \setminus \widehat{D})$ is $O(\eta)$. Suppose $f_1, \dots, f_k$ are all the failed vertices in $D \setminus \widehat{D}$. Then $|S_u(f_1)| + |S_u(f_2)| + \cdots + |S_u(f_k)| = O(\eta)$ for every $u \in \widehat{D} \setminus D$. It follows that Algorithm 2 runs in time $O(\eta^3 \log n)$.

Algorithm 2 Adds a type-5 edge in $\mathcal{M}$ between every $u, v \in \widehat{D} \setminus D$ that are connected to the same hanging subtree.

```
1 foreach u ∈ D̂ \ D do
      // Consider the DFS tree T_u and the respective DFS numbering.
2     foreach v ∈ D̂ \ D do
3         foreach failed vertex f ∈ D \ D̂ do
4             S_u(f) ← the collection of contiguous slices of the sorted list of children of
                f that are marked by u consisting only of roots of hanging subtrees;
5             foreach slice s ∈ S_u(f) do
6                 L ← min(s);
7                 R ← max(s);
                  // Check the condition below in O(log n) time using binary
                     search.
8                 if neighbors_u(v) ∩ [L, R + ND(R) − 1] ≠ ∅ then
9                     add an edge between u and v in M;
```

3.2.3 Computing type-6 edges

In this section we describe our algorithm for adding type-6 edges to $\mathcal{M}$, see Algorithm 3. Recall that type-6 edges join vertices from $\widehat{D} \setminus D$ with internal components that are connected to them indirectly via a hanging subtree.

Conceptually, we want to query each (C, u, H)-tuple (where C is an internal component, u is a vertex from $\widehat{D} \setminus D$, and H is a hanging subtree) whether H contains a neighbor of u and a back-edge to C. Similarly to how we compute type-5 edges, for each $u \in \widehat{D} \setminus D$ we group together relevant hanging subtrees in order to bound the number of queries we make on them (line 3). Note that if C is an internal component connected to H through a back-edge, then C is an ancestor of H (see Lemma 6 and Corollary 7 in [21]). Therefore, once we have fixed a vertex $u \in \widehat{D} \setminus D$ and a failed vertex $f \in D \setminus \widehat{D}$, we only need to traverse T_u up from f and perform queries on each encountered internal component (the while-loop in line 5). During that traversal, whenever we encounter an internal component C, we want to check whether there exists a back-edge between C and a hanging subtree H whose root is a child of f and which contains a neighbor of u. We do it in line 11 by sending a query to the 2D-range-emptiness data structure (created in item 8 of the preprocessing phase) indexed by the DFS numbering of tree T_u. The validity of line 11 comes from two lemmas in [21], which we reprint here:

▶ **Lemma 6** ([21] Lemma 4(2)). *Let C be an internal component of $T \setminus X$, where X is any set of failed vertices. For every vertex v that is a descendant of C, there is a unique boundary vertex of C that is an ancestor of v.*

▶ **Lemma 7** ([21] Lemma 8). *Let C, C' be two connected components of $T \setminus X$, where X is any set of failed vertices, such that C' is an internal component that is an ancestor of C. Let b be the boundary vertex of C' that is an ancestor of C. Then there is a back-edge from C to C' if and only if there is a back-edge from C whose lower end lies in $[r_{C'}, p_T(b)]$.*

Now let us show the correctness of Algorithm 3. Suppose that a vertex u from $\widehat{D} \setminus D$ and an internal component C are both connected in $G \setminus D$ to a hanging subtree H, and denote by $f \in D \setminus \widehat{D}$ the parent of the root of H. We claim that Algorithm 3 eventually finds C in

Algorithm 3 Adds a type-6 edge in $\mathcal{M}$ between every $u \in \widehat{D} \setminus D$ and every internal component C of $T \setminus (D \setminus \widehat{D})$ that are connected to the same hanging subtree H.

```
1  foreach u ∈ D̂ \ D do
       // Consider the DFS tree T_u and the respective DFS numbering.
2      foreach failed vertex f ∈ D \ D̂ do
3          S_u(f) ← the collection of contiguous slices of the sorted list of children of f
             that are marked by u consisting only of roots of hanging subtrees;
4          f' ← f;
5          while f' ≠ null do
6              if p_T(f') ≠ parent_F(f') then
                   // Vertex f' is a boundary vertex of some internal
                      component.
7                  C ← the internal component of T \ (D \ D̂) such that f' ∈ ∂(C);
8                  foreach slice s ∈ S_u(f) do
9                      L ← min(s);
10                     R ← max(s);
11                     if 2D_range_query_u([L, R + ND(R) − 1] × [r_C, p_T(f')]) ≠ ∅ then
12                         add an edge between u and r_C in M;
13             f' ← parent_F(f');
```

T_u by traversing up from f. Indeed, by Lemma 6, let f' be the boundary vertex of C that is an ancestor of f. Then, by Lemma 7, we know that the 2D range query in line 11 returns true. Conversely, if the 2D range query in line 11 returns true for some C and f', then we can also conclude that u is connected with C through the mediation of a hanging subtree whose root is a child of f.

By the same reasoning as for Algorithm 2, each query corresponds to a tuple (u, f, f', s). For fixed u and f' the total number of segments s in $T_u \setminus (D \setminus \widehat{D})$, over all choices of f, is $O(\eta)$. Hence the number of 2D range queries Algorithm 3 makes is $O(\eta^3)$. Since each such query takes $O(\log n)$ time [4], the total running time is $\tilde{O}(\eta^3)$.

3.3 The query phase

Finally, we describe our algorithm for the query phase, see Algorithm 4. Given s and t in $G \setminus D$, we first determine where they lie in. If they lie in some internal component(s) of $T \setminus (D \setminus \widehat{D})$ or in the extra set of vertices $\widehat{D} \setminus D$, we can check whether these components/vertices are connected in $\mathcal{M}$. However, if s (or t) lies in a hanging subtree, we try to find a surrogate for s (respectively t), namely a vertex in $\widehat{D} \setminus D$ or an internal component of $T \setminus (D \setminus \widehat{D})$ that is connected by an edge to the hanging subtree of s (respectively t).[6] If we can find such a surrogate, we can replace s (respectively t) with it, and refer to $\mathcal{M}$ as in the previous case. However, if no such surrogate exists, either for s or for t, then s and t are connected if and only if they reside in the same hanging subtree.

[6] Note that in order to find a surrogate internal component it is sufficient to check the $low_1, \ldots, low_\eta$ points of the root of the hanging subtree.

Algorithm 4 Given two vertices $s, t \in V \setminus D$, answers if they are connected in $G \setminus D$.

```
1  function find_representative(u)
2      if u ∈ D̂ \ D then return u;
3      if u lies in an internal component C of T \ (D \ D̂) then return r_C;
4      H ← the hanging subtree of T \ (D \ D̂) where u lies in;
       // Try to find an internal component as a surrogate.
5      for i ∈ {1, ..., η} do
6          if low_i(r_H) ≠ null and low_i(r_H) ∉ D then
7              C ← the internal component of T \ (D \ D̂) where low_i(r_H) lies in;
8              return r_C;
       // Try to find a vertex in D̂ \ D as a surrogate.
9      foreach v ∈ D̂ \ D do
10         if r_H is marked in T_v then return v;
       // No surrogate found, return the hanging tree's root as the
          representative.
11     return r_H;
12 s ← find_representative(s);
13 t ← find_representative(t);
14 if s ∈ V(M) and t ∈ V(M) then
15     return whether s and t are connected in M;
16 else /* At least one vertex lies in an isolated hanging subtree,
      represented by its root.                                          */
17     return whether s = t;
```

Now let us justify the running time for Algorithm 4. Given that we have computed the connected components of $\mathcal{M}$ at the end of the update phase, we can decide in $O(1)$ time the connectivity among internal components and vertices from $\widehat{D} \setminus D$. Locating the correct connected component in $T \setminus (D \setminus \widehat{D})$ and deciding whether it is an internal component or a hanging subtree takes $O(\eta)$ time (see Proposition 13 in [21] for more detail). The for-loops on lines 5 and 9, which are looking for a surrogate, both take $O(\eta)$ time. Summing up, Algorithm 4 answers a query in $O(\eta)$ time.

4 Lower bound for preprocessing time

In this section, we show that the $\tilde{O}(dm)$ preprocessing time of our oracle cannot be improved by a polynomial factor assuming the Exact Triangle Hypothesis, which is implied by both the 3SUM Hypothesis and the APSP Hypothesis.

To show this, we give a fine-grained reduction from the offline SetDisjointness problem. This problem was introduced by Kopelowitz, Pettie, and Porat [20], who showed it is hard under the 3SUM Hypothesis. Vassilevska Williams and Xu extended their hardness result to hold under a weaker assumption, namely the Exact Triangle Hypothesis; as a consequence the problem is also hard under the APSP Hypothesis. The input to the SetDisjointness problem is a universe of elements U, a family of subsets $F \subseteq 2^U$, and a collection of query pairs $(S_i, S_j) \in F \times F$, whose number is denoted by q. For each query, one has to answer whether $S_i \cap S_j$ is empty or not. We use the following lower bound for SetDisjointness from [37, Corollary 3.11].

▶ **Theorem 8** (Vassilevska Williams and Xu [37, Corollary 3.11]). *For any constant $\gamma \in (0, 1)$, let $\mathcal{A}$ be an algorithm for offline SetDisjointness where $|U| = \Theta(n^{2-2\gamma})$, $|F| = \Theta(n)$, each set in F has at most $O(n^{1-\gamma})$ elements from U, and $q = \Theta(n^{1+\gamma})$. Assuming the Exact Triangle Hypothesis, $\mathcal{A}$ cannot run in $O(n^{2-\varepsilon})$ time, for any $\varepsilon > 0$.*

Proof of Theorem 2. We reduce solving the offline SetDisjointness problem, in the parameter regime specified in Theorem 8 (later, we set $\gamma = 1 - \varepsilon/2$), to creating a connectivity oracle for predictable vertex failures on an undirected graph with $O(n + n^{2-2\gamma})$ vertices and $O(n^{2-\gamma})$ edges, with $d = O(n)$, and then running the update and query phases q times each, with $\eta = O(1)$.

Here is how we construct the undirected graph G. Let $V(G)$ consist of two disjoint sets A and B. The vertices in $A = \{a_1, a_2, \dots, a_{|F|}\}$ represent the sets in $F = \{S_1, S_2, \dots, S_{|F|}\}$, and the vertices in $B = \{b_u \mid u \in U\}$ represent the elements in U. We put edges between A and B so that $a_i \in A$ is connected to $b_u \in B$ if and only if $u \in S_i$. Note that, for $i \neq j$, $S_i \cap S_j \neq \emptyset$ if and only if a_i and a_j are connected in the subgraph of G induced on $B \cup \{a_i, a_j\}$. The constructed graph has $|A| + |B| = |F| + |U| = O(n + n^{2-2\gamma})$ vertices and $O(|F| \cdot \max_i |S_i|) = O(n^{2-\gamma})$ edges.

In the preprocessing phase we set $\widehat{D} = A$. Then, for each input query pair (S_i, S_j), we set $D = \widehat{D} \setminus \{a_i, a_j\}$, do the update, and then query connectivity between a_i and a_j.

Let t_1 denote the preprocessing time; t_2 denote the update time; and t_3 denote the query time. Then, the total time to solve SetDisjointness is $t_1 + q \cdot (t_2 + t_3)$. By Theorem 8, we get that $t_1 + q \cdot (t_2 + t_3) \geqslant n^{2-o(1)}$, unless the Exact Triangle Hypothesis fails. Since $\eta = 2$ is a constant, we know that $t_2 = n^{o(1)}$ and $t_3 = n^{o(1)}$, and hence $q \cdot (t_2 + t_3) = n^{1+\gamma+o(1)}$. This forces $t_1 \geqslant n^{2-o(1)}$. Recall that $d = |\widehat{D}| = |A| = |F| = O(n)$, and $m = O(n^{2-\gamma})$. If $t_1 = O(d^{1-\varepsilon} m)$ (or $O(dm^{1-\varepsilon})$), then by setting $\gamma = 1 - \varepsilon/2$ we would get that $t_1 = O(n^{2-\varepsilon/2})$ and hence the Exact Triangle Hypothesis would fail. ◀

References

1. Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.
2. Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic DFS in undirected graphs: Breaking the o(m) barrier. *SIAM J. Comput.*, 48(4):1335–1363, 2019. doi:10.1137/17M114306X.
3. Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.
4. Timothy M. Chan, Kasper Green Larsen, and Mihai Puatracscu. Orthogonal range searching on the ram, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.
5. Timothy M. Chan, Mihai Puatracscu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM J. Comput.*, 40(2):333–349, 2011. doi:10.1137/090751670.
6. Michal Dory and Merav Parter. Fault-tolerant labeling and compact routing schemes. In *PODC '21: ACM Symposium on Principles of Distributed Computing*, pages 445–455. ACM, 2021. doi:10.1145/3465084.3467929.
7. Ran Duan. New data structures for subgraph connectivity. In *Automata, Languages and Programming, 37th International Colloquium, ICALP*, volume 6198 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 2010. doi:10.1007/978-3-642-14165-2_18.

8 Ran Duan and Seth Pettie. Connectivity oracles for failure prone graphs. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 465–474. ACM, 2010. doi:10.1145/1806689.1806754.

9 Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 490–509. SIAM, 2017. doi:10.1137/1.9781611974782.31.

10 Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. *SIAM J. Comput.*, 49(6):1363–1396, 2020. doi:10.1137/17M1146610.

11 Ran Duan and Le Zhang. Faster randomized worst-case update time for dynamic subgraph connectivity. In *Algorithms and Data Structures - 15th International Symposium, WADS*, volume 10389 of *Lecture Notes in Computer Science*, pages 337–348. Springer, 2017. doi:10.1007/978-3-319-62127-2_29.

12 Paolo Ferragina and Giorgio Vinciguerra. Learned data structures. In *Recent Trends in Learning From Data - Tutorials from the INNS Big Data and Deep Learning Conference (INNSBDDL 2019)*, volume 896 of *Studies in Computational Intelligence*, pages 5–41. Springer, 2019. doi:10.1007/978-3-030-43883-8_2.

13 Daniele Frigioni and Giuseppe F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000. doi:10.1007/S004530010032.

14 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.

15 Monika Henzinger and Stefan Neumann. Incremental and fully dynamic subgraph connectivity for emergency planning. In *24th Annual European Symposium on Algorithms, ESA 2016*, volume 57 of *LIPIcs*, pages 48:1–48:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.ESA.2016.48.

16 Monika Henzinger, Barna Saha, Martin P. Seybold, and Christopher Ye. On the complexity of algorithms with predictions for dynamic graph problems. In *15th Innovations in Theoretical Computer Science Conference, ITCS 2024*, volume 287 of *LIPIcs*, pages 62:1–62:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ITCS.2024.62.

17 Taisuke Izumi, Yuval Emek, Tadashi Wadayama, and Toshimitsu Masuzawa. Deterministic fault-tolerant connectivity labeling scheme. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC*, pages 190–199. ACM, 2023. doi:10.1145/3583668.3594584.

18 Ce Jin and Yinzhan Xu. Tight dynamic problem lower bounds from generalized BMM and omv. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1515–1528. ACM, 2022. doi:10.1145/3519935.3520036.

19 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 1131–1142. SIAM, 2013. doi:10.1137/1.9781611973105.81.

20 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3SUM conjecture. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 2016, pages 1272–1287. SIAM, 2016. doi:10.1137/1.9781611974331.ch89.

21 Evangelos Kosinas. Connectivity queries under vertex failures: Not optimal, but practical. In *31st Annual European Symposium on Algorithms, ESA 2023*, volume 274 of *LIPIcs*, pages 75:1–75:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ESA.2023.75.

22 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018*, pages 489–504. ACM, 2018. doi:10.1145/3183713.3196909.

23 Honghao Lin, Tian Luo, and David P. Woodruff. Learning augmented binary search trees. In *International Conference on Machine Learning, ICML 2022*, volume 162 of *Proceedings of Machine Learning Research*, pages 13431–13440. PMLR, 2022. URL: https://proceedings.mlr.press/v162/lin22f.html.

24 Alexander Lindermayr and Nicole Megow. Algorithms with predictions. https://algorithms-with-predictions.github.io, 2022. Accessed 8 September 2023.

25 Quanquan C. Liu and Vaidehi Srinivas. The predicted-updates dynamic model: Offline, incremental, and decremental to fully dynamic transformations. In *The Thirty Seventh Annual Conference on Learning Theory, COLT 2024*, Proceedings of Machine Learning Research. PMLR, 2024.

26 Yaowei Long and Thatchaphol Saranurak. Near-optimal deterministic vertex-failure connectivity oracles. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022*, pages 1002–1010. IEEE, 2022. doi:10.1109/FOCS54457.2022.00098.

27 Yaowei Long and Yunfan Wang. Better decremental and fully dynamic sensitivity oracles for subgraph connectivity. In *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

28 Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. In Tim Roughgarden, editor, *Beyond the Worst-Case Analysis of Algorithms*, pages 646–662. Cambridge University Press, 2020. doi:10.1017/9781108637435.037.

29 Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Commun. ACM*, 65(7):33–35, 2022. doi:10.1145/3528087.

30 Merav Parter and Asaf Petruschka. Õptimal dual vertex failure connectivity labels. In *36th International Symposium on Distributed Computing, DISC*, volume 246 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.DISC.2022.32.

31 Merav Parter, Asaf Petruschka, and Seth Pettie. Connectivity labeling and routing with multiple vertex failures. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024*, pages 823–834. ACM, 2024. doi:10.1145/3618260.3649729.

32 Mihai Pătraşcu and Mikkel Thorup. Planning for fast connectivity updates. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007)*, pages 263–271. IEEE Computer Society, 2007. doi:10.1109/FOCS.2007.54.

33 David Peleg. Informative labeling schemes for graphs. *Theor. Comput. Sci.*, 340(3):577–593, 2005. doi:10.1016/J.TCS.2005.03.015.

34 Michal Pilipczuk, Nicole Schirrmacher, Sebastian Siebertz, Szymon Torunczyk, and Alexandre Vigny. Algorithms and data structures for first-order logic with connectivity under vertex failures. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022*, volume 229 of *LIPIcs*, pages 102:1–102:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.102.

35 Jan van den Brand, Sebastian Forster, Yasamin Nazari, and Adam Polak. On dynamic graph algorithms with predictions. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3534–3557. SIAM, 2024. doi:10.1137/1.9781611977912.126.

36 Jan van den Brand and Thatchaphol Saranurak. Sensitive distance and reachability oracles for large batch updates. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 424–435. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00034.

37 Virginia Vassilevska Williams and Yinzhan Xu. Monochromatic triangles, triangle listing and APSP. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 786–797. IEEE, 2020. doi:10.1109/FOCS46700.2020.00078.