Connectivity Oracles for Graphs Subject to Vertex Failures[*]

Ran Duan
Tsinghua University

Seth Pettie
University of Michigan

Abstract

We introduce new data structures for answering connectivity queries in graphs subject to batched *vertex failures*. Our deterministic structure processes a batch of $d \leq d_\star$ failed vertices in $\tilde{O}(d^3)$ time and thereafter answers connectivity queries in $O(d)$ time. It occupies space $O(d_\star m \log n)$. We develop a randomized Monte Carlo version of our data structure with update time $\tilde{O}(d^2)$, query time $O(d)$, and space $\tilde{O}(m)$ for any $d_\star$. This is the first connectivity oracle for general graphs that can efficiently deal with an unbounded number of vertex failures.

Our data structures are based on a new decomposition theorem for an undirected graph $G = (V, E)$, which is of independent interest. It states that for any terminal set $U \subseteq V$ we can remove a set B of $|U|/(s-2)$ vertices such that the remaining graph contains a Steiner forest for $U - B$ with maximum degree s.

1 Introduction

The *dynamic subgraph model* [19, 21, 36, 39, 37, 42, 65] is a constrained dynamic graph model. Rather than allow the graph to evolve in completely arbitrary ways (via an unbounded sequence of edge insertions and deletions), there is assumed to be a *fixed* ideal graph $G = (V, E)$ that can be preprocessed in advance. The ideal graph is susceptible only to the *failure* of edges/vertices and their subsequent *recovery*, possibly with a bound $d_\star$ on the number of failures at one time. Queries naturally answer questions about the current failure-free subgraph. This model is useful because it more accurately represents the behavior of many real-world networks: changes to the underlying topology are relatively rare but transient failures very common. More importantly, this model offers the algorithm designer the freedom to explore exotic graph representations. Because *preprocessing time* is not the most critical measure of efficiency, it may be desirable to build a specialized graph representation that facilitates more efficient updates and queries.

Dynamic Subgraph Connectivity The dynamic subgraph model was introduced by Frigioni and Italiano [42] who showed that when the ideal graph is planar, vertex failures/recoveries and connectivity queries could be handled in $O(\log^3 n)$ amortized time, after $\tilde{O}(n)$ preprocessing. Their algorithm even allowed the ideal graph to evolve via edge updates, also in $O(\log^3 n)$ amortized time, so long as it remained planar. Dynamic subgraph connectivity structures were later developed for general graphs [8, 19, 21, 36]. Chan, Pătrașcu, and Roditty [21] gave an $O(m^{4/3})$-space structure that handles vertex failures/recoveries in $\tilde{O}(m^{2/3})$ amortized time and connectivity queries in $O(m^{1/3})$ time. Duan [36] developed a different $O(m)$-space structure with the same amortized update and query time as [21], and a new $\tilde{O}(m)$-space structure with *worst case* $\tilde{O}(m^{4/5})$-time updates and $O(m^{1/5})$ time queries. Very recently Baswana et al. [8] showed how to maintain a DFS tree in the dynamic subgraph model with $\tilde{O}(\sqrt{mn})$ update time, which supports $O(1)$-time connectivity queries.

Pătrașcu and Thorup [65] considered a situation where a *batch* of d edges fail simultaneously. They showed that an $O(m)$-space structure could be constructed that handles updates in $O(d \log^2 n \log \log n)$ time and subsequently answers connectivity queries in $O(\log \log n)$ time. Moreover, they observed that the query time could not be unilaterally improved, by a reduction to the predecessor problem [64, 66]. One downside of the Pătrașcu-Thorup structure is that it requires *exponential* time to compute: it involves solving *sparsest cut* $\tilde{O}(n)$ times on various subgraphs. Using a polynomial time $O(\sqrt{\log n})$-approximate sparsest cut algorithm [4] instead increases the update time to $O(d \log^{5/2} n \log \log n)$. Pătrașcu and Thorup [65] were motivated by the absence of a fully dynamic connectivity data structure with poly($\log n$)

worst case update time.[1] Kapron, King, and Mountjoy [55] discovered a randomized dynamic connectivity structure with $O(c \cdot \text{poly}(\log n))$ update time that errs with probability n^{-c}. Gibb, Kapron, King, and Thorn [46] observed that this data structure can function correctly, w.h.p., without actually storing the graph. This leads to a d-edge failure connectivity oracle with update and query time similar to [65], but using just $\tilde{O}(n)$ space.

The analogous d-vertex failure connectivity problem is inherently more complex. Whereas removing d edges can only increase the number of connected components by d, removing d vertices can have an impact on the connectivity that is completely disproportionate to d. When $d = 1$ we can use the block tree representation of biconnected components to answer connectivity queries in constant time; see [15] for data structural details. When $d = 2$ we can use the SPQR tree [10, 15] of each biconnected component to answer queries in $O(1)$ time. A data structure of Kanevsky et al. [54] can answer queries in $O(1)$ time when $d = 3$. Similar ad hoc solutions can also be designed for d-edge failure connectivity oracles, for constant $d \leq 4$ [35, 45, 68, 70]. However, scaling these solutions up, even to an arbitrarily large constant d, becomes prohibitively complex, even in the simpler case of edge failures. In a λ-edge connected graph, encoding all λ-edge cuts is simple with the *cactus* [30] representation, but the simplicity is lost when encoding both λ- and $(\lambda + 1)$-edge cuts. See [32, 33, 34].

In previous work [39] we designed a d-edge failure oracle that reduces the problem to 2D orthogonal range reporting. Using the range reporting structure of Chan, Larsen, and Pătraşcu [20] gives a d-edge failure structure with $O(d^2 \log \log n)$ update time, $O(\min\{\frac{\log d}{\log \log n}, \frac{\log \log n}{\log \log \log n}\})$ query time, and $O(m \log \log n)$ space, or a somewhat slower update time with $O(m)$ space. By itself, this structure compares favorably with the d-edge failure oracles of [65, 55] when $d = O(\log n)$. However, it has additional properties that make it attractive for use in d-vertex failure oracles. Specifically, if D is the set of failed vertices, the update time is actually $O\left((\sum_{v \in D} \deg_T(v))^2 \log \log n\right)$, where T is any

---

[1]There are dynamic connectivity structures with *amortized* poly$(\log n)$ update time [50, 71, 51]. However, the fastest worst-case update time is $O(\sqrt{\frac{n(\log \log n)^2}{\log n}})$ [56, 40], a small improvement over the long-standing $O(\sqrt{n})$ bound of [41, 40].

spanning tree of the graph. In other words, the update time is quadratic *in the sum of the T-degrees*, independent of their degrees in G.

If G were guaranteed to have an $O(1)$-degree spanning tree we would immediately have a satisfactory d-vertex failure connectivity oracle with update time $\tilde{O}(d^2)$ and query time $\tilde{O}(1)$. Of course, there is no such guarantee. Every *bridge* edge appears in every spanning tree T, so a vertex incident to many bridges must have high T-degree. Since bridges are easy to deal with this is not a very convincing counterexample. One might hope that if G had sufficient connectivity, a low-degree spanning tree could be found. This is the approach taken by Borradaile, Pettie, and Wulff-Nilsen's [15] d-failure connectivity oracles for *planar* graphs. Barnette's theorem [6] states that every triconnected planar graph has a degree-3 spanning tree, which can be found in linear time [28, 69]. However, the analogues of Barnette's theorem for general graphs are too weak to be of any use. Czumaj and Strothmann [28, 69] proved that a k-connected graph with maximum degree $\Delta(G) \leq k(\Delta_T - 2) + 2$ has a degree-Δ_T spanning tree, which can be found in polynomial time. If, however, the maximum degree is at least $\Delta(G) \geq k(\Delta_T - 1)$ it is NP-hard to decide if there is a degree-Δ_T spanning tree. Thus, even if we could force G to be k-connected for some large constant k, it would not help to find a low-degree spanning tree.

In [39] we developed a $d_\star$-vertex failure connectivity oracle that offers a tradeoff between update time and size. For any integer parameter $c \geq 1$, the space of the data structure is $O(d_\star^{1-2/c} m n^{1/c - 1/(c \log(2d_\star))} \log^2 n)$ and the time to process $d \leq d_\star$ vertex failures is $O(d^{2c+4} \log^2 n \log \log n)$. Thereafter connectivity queries can be answered in $O(d)$ time. The main drawbacks of [39] are its conceptual complexity and very poor tradeoff between space and update time. Henzinger and Neumann [49] recently showed how any d-vertex failure connectivity oracle could be transformed to support fully dynamic updates in the dynamic subgraph model, where vertices fail and recover individually.

New Results In this paper we present dramatically better d-vertex failure connectivity oracles that match or improve on [39] in every measure of efficiency except construction time. Using space $O(d_\star m \log n)$, a batch D of $d \leq d_\star$ vertex failures is processed in $O(d^3 \log^3 n)$ time such that connectivity queries in $G - D$ can be answered in

$O(d)$ time.[2] The construction time is $O(mn \log n)$. Note that there is now no tradeoff between space and update time. Clearly any pair of $(d_\star + 1)$-connected vertices cannot be disconnected by d failures. By preprocessing the graph with the linear time Nagamochi-Ibaraki algorithm [60], we can replace $E(G)$ by an equivalent subgraph containing $\bar{m} = \min\{m, (d_\star + 1)n\}$ edges. Thus, the factors of m in the space and construction time can be replaced with $\bar{m}$.

By incorporating graph sketching techniques from [3, 55] we are able to reduce the update time to $O(d^2 \log^5 n)$ and space to $O(m \log^4 n)$, even for $d_\star = n$. The cost of these savings is the possibility of undetected false negatives: a connectivity query may incorrectly report that two vertices are disconnected in $G - D$, with probability $1/\text{poly}(n)$.

Our data structures are based on a new graph decomposition theorem, which is obtained from a recursive version of the Fürer-Raghavachari [43] algorithm for approximating the minimum degree spanning tree. The theorem states that for any undirected graph $G = (V, E)$, terminal set $U \subseteq V$, and integer s, there exists a set of $|U|/(s - 2)$ vertices B that can be removed, such that $U - B$ is spanned by a degree-s Steiner forest in the graph $G - B$. We believe this decomposition theorem is of independent interest.

Lower Bounds One question raised by [39] is whether it is possible for a *d-vertex* failure oracle to match the $\tilde{O}(1)$ query time of existing *d-edge* failure oracles [65, 39, 55]. There is now strong circumstantial evidence that no such data structure exists with reasonable update time. In particular, if the *Integer 3SUM Conjecture* holds[3] then any d-vertex failure connectivity oracle with subquadratic preprocessing and reasonable update time must have $\Omega(d^{1/2-o(1)})$ query time [57]. Henzinger et al. [48] showed that the *OMv conjecture*[4] on the

hardness of online matrix-vector multiplication implies an $\Omega(d^{1-o(1)})$ query lower bound, even if any polynomial preprocessing is allowed. Thus, beating $O(d)$ query time would require refuting a plausible conjecture. Of course, the plausibility of the 3SUM and OMv conjectures continue to be actively scrutinized. Stronger forms of the 3SUM and OMv conjectures have already been refuted; see [47, 59].

Related Work Much of the previous work in the d-failure model has focussed on computing approximate shortest paths avoiding edge and vertex failures. Demetrescu et al. [29] gave an exact shortest path oracle for weighted directed graphs subject to $d = 1$ failure. It occupies $O(n^2 \log n)$ space and answers queries in constant time. The construction time for this oracle was later improved by Bernstein and Karger [13]. An analogous result for $d = 2$ failures was presented by Duan and Pettie [38] (see also [37]), which uses space $O(n^2 \log^3 n)$ and query time $O(\log n)$. *Approximate* distance oracles for d edge failures were given for general graphs [24], with stretch that grows linearly in d.

These problems have also been studied on special graph classes. Borradaile et al. [15] described connectivity oracles for planar graphs subject to d-edge failures or d-vertex failures. See Baswana et al. [7] for exact distance oracles for planar graphs avoiding $d = 1$ failure, and Abraham et al. [1, 2] for approximate distance oracles for planar graphs and graphs of bounded doubling dimension.

Parter and Peleg [62] considered the problem of computing a subgraph that preserves shortest paths from s sources after a single edge or vertex failure. They proved that $\Theta(s^{1/2}n^{3/2})$ edges are necessary and sufficient, for every s. See also [14, 17, 18, 23, 31, 61, 63] for spanners (subgraphs) that preserve approximate distances subject to edge or vertex failures.

Very recently researchers have considered reachability problems on directed graphs subject to vertex failures. Choudhary [27] gave an optimal $O(n)$-space, $O(1)$-query time reachability oracle for $d = 2$ failures. Baswana, Choudhary, and Roditty [9] considered the problem of finding a sparse subgraph that preserves reachability from a single source, subject to d vertex failures. They proved that $\Theta(2^d n)$ edges are necessary and sufficient.

1.1 Organization In Section 2 we review the Euler Tour structure of [39] for handling d edge failures. We begin Section 3 with a sketch of the Fürer-Raghavachari algorithm FR-Tree, then describe our

---

[2]The notation $G - D$ is short for the subgraph of G induced by $V(G) - D$.

[3]The 3SUM problem is, given a set A of n numbers, to determine if there exist $a, b, c \in A$ for which $a + b + c = 0$. There are now known to be $O(n^2/\text{poly}(\log n))$ algorithms for both integer inputs [5] and real inputs [47]. The Integer 3SUM Conjecture asserts that the problem requires $\Omega(n^{2-o(1)})$ time, even if $A \subset \{-n^3, \ldots, n^3\}$.

[4]The OMv conjecture is that given a matrix $M \in \{0, 1\}^{n \times n}$ to be preprocessed and n vectors $v_1, \ldots, v_n \in \{0, 1\}^n$ presented online, the total cost of preprocessing and computing the products $\{Mv_i\}_{1 \le i \le n}$ is $\Omega(n^{3-o(1)})$. Note that fast matrix multiplication is not obviously helpful in this context since Mv_i must be reported before receiving v_{i+1}.

d-Edge Failure Structures and Lower Bounds

	Update	Query	Space	Preprocessing
Pătraşcu & Thorup (2007)	$O(d \log^2 \log \log n)$	$O(\log \log n)$	$O(m)$	exponential
	$O(d \log^{5/2} \log \log n)$			polynomial
	$O(d \operatorname{poly}(\log n))$	$\Omega(\log \log n)$	any	any
Duan & Pettie (2010)	$O(d^2 \log \log n)$	$\min \left\{ O(\frac{\log d}{\log \log n}), \right.$	$O(m \log \log n)$	linear in space
	$O(d^2 \log^\epsilon n)$	$\left. O(\frac{\log \log n}{\log \log \log n}) \right\}$	$O(m)$	
Kapron, King & Mountjoy (2013) Gibb, Kapron, King, Thorn (2015)	$O(d \log^3 n \log d)$	$O(\log \log n)$, correct w.h.p.	$O(n \log^2 n)$	near linear (Rand.)

d-Vertex Failure Structures and Lower Bounds

	Update	Query	Space	Preprocessing
$d = 1$: Block tree; see [15]	$O(1)$	$O(1)$	$O(n)$	linear
$d = 2$: SPQR tree; see [15, 10]	$O(1)$	$O(1)$	$O(n)$	linear
$d = 3$: Kanevsky et al. [54]	$O(1)$	$O(1)$	$O(n)$	near linear
Duan & Pettie (2010)	$\tilde{O}(d^{2c+4})$	$O(d)$	$\tilde{O}(d_\star^{1-\frac{2}{c}} \bar{m} n^{\frac{1}{c} - \frac{1}{c \log(2d_\star)}})$	linear in space
Henzinger et al. (2014) (assuming OMv Conjecture)	$O(\operatorname{poly}(d, \log n))$ $O((dn)^{1-\epsilon})$	$\Omega(d^{1-o(1)})$	any	$O(\operatorname{poly}(n))$
Kopelowitz, Pettie & Porat (2016) (assuming 3SUM Conjecture)	$O(\operatorname{poly}(d, \log n))$ $O((dn)^{1-\epsilon})$	$\Omega(d^{1/2-o(1)})$	any	$O(mn^{1-\epsilon} \operatorname{poly}(d))$
New	$O(d^3 \log^3 n)$	$O(d)$	$O(d_\star \bar{m} \log n)$	$O(\bar{m} n \log n)$ (Determ.)
	$O(d^2(d^2 + \log^\epsilon n) \log^2 n)$			
	$O(d^2(d^2 + \log \log n) \log^2 n)$		$O(d_\star \bar{m} \log n \log \log n)$	
	$O(d^2 \log^5 n)$	$O(d)$, correct w.h.p.	$O(m \log^4 n)$	$O(mn \log n)$ (Rand.)

Table 1: The lower bounds of Pătraşcu and Thorup are unconditional whereas the lower bounds of Kopelowitz et al. and Henzinger et al. rely on unproven conjectures. Whenever $d_\star$ is available at construction time we can replace m (the number of edges) with $\bar{m} \overset{\text{def}}{=} \min\{m, (d_\star + 1)n\}$.

decomposition algorithm Decomp. In Section 4 we observe that by applying Decomp iteratively, we naturally obtain a representation of the graph as a *low degree hierarchy*. Section 4 describes how to build a *d*-failure connectivity oracle, by supplementing the low degree hierarchy with suitable data structures. The algorithms for deleting failed vertices and answering connectivity queries are presented in Section 5. The basic algorithm for deleting failed vertices takes $\tilde{O}(d^4)$ time using standard 2D range reporting data structures. In Section 6 we give three distinct ways to reduce this to $\tilde{O}(d^3)$ using more sophisticated range searching structures. In Section 7 we present a randomized Monte Carlo version of our data structure with update time $\tilde{O}(d^2)$ and space $\tilde{O}(m)$. We discuss open problems in Section 8.

2   The Euler Tour Structure

In this section we describe the *ET-structure* for handling connectivity queries avoiding multiple vertex and edge failures. When handling only d edge failures, the performance of the ET-structure is incomparable to that of Pătraşcu and Thorup [65] in nearly every respect.[5] The strength of the ET-structure is that if the graph can be covered by a low-degree tree T, the time to delete a *vertex* is a function of its degree in T; incident edges not in T are deleted implicitly. We prove Theorem 2.1 in the remainder of this section.

THEOREM 2.1. *Let $G = (V, E)$ be a graph, with $m = |E|$ and $n = |V|$, and let $\mathcal{F} = \{T_1, \ldots, T_{|\mathcal{F}|}\}$ be a set of vertex disjoint trees in G. ($\mathcal{F}$ does not necessarily span connected components of G.) There is a data structure $\mathbf{ET}(G, \mathcal{F})$ that supports the following operations. Suppose D is a set of failed edges, of which d are tree edges in $\mathcal{F}$ and d' are non-tree edges. Deleting D splits some subset of the trees in $\mathcal{F}$ into at most $2d$ trees $\mathcal{F}' = \{T_1', \ldots, T_{2d}'\}$. In $O(d^2 q + d')$ time we can report which pairs of trees in $\mathcal{F}'$ are connected by an edge*

---

[5]The ET-structure is significantly faster in terms of construction time (near-linear vs. a large polynomial or exponential time) though it uses slightly more space: $O(m \log \log n)$ vs. $O(m)$. It handles d edge deletions exponentially faster for bounded d ($O(\log \log n)$ vs. $\Omega(\log^2 n \log \log n)$) but is slower as a function of d: $O(d^2 \log \log n)$ vs. $O(d \log^2 n \log \log n)$ time. The query time is the same for both structures, namely $O(\log \log n)$. Whereas the ET-structure naturally maintains a certificate of connectivity (a spanning tree), the Pătraşcu-Thorup structure requires modification and an additional logarithmic factor in the update time to maintain a spanning tree.

in $E - D$. In $O(\min\left\{\frac{\log \log n}{\log \log \log n}, \frac{\log d}{\log \log n}\right\})$ time we can determine which tree in $\mathcal{F}'$ contains a given vertex. Using space $O(m \log \log n)$ the value of q is $O(\log \log n)$; using space $O(m)$ the value of q is $O(\log^\epsilon n)$ for any fixed $\epsilon > 0$.

Our data structure uses Chan, Larsen, and Pătraşcu's [20] structure for orthogonal range reporting on the integer grid $[U] \times [U]$. They showed that given a set of N points, there is a data structure with size $O(N \log \log N)$ such that given $x, y, w, z \in [U]$, the set of points in $[x, y] \times [w, z]$ can be reported in $O(\log \log U + k)$ time, where k is the number of reported points. If the space is reduced to $O(N)$ the update time becomes $O(\log^\epsilon U + k)$ for any fixed $\epsilon > 0$.

For a tree T, let $\text{Euler}(T)$ be a list of its vertices encountered during an Euler tour of T (an undirected edge is treated as two directed edges), where we only keep the *first* occurrence of each vertex. One may easily verify that removing f edges from T partitions it into $f + 1$ connected subtrees and splits $\text{Euler}(T)$ into at most $2f + 1$ intervals, where the vertices of a connected subtree are the union of some subset of the intervals. To build $\mathbf{ET}(G = (V, E), \mathcal{F})$ we build the following structure for each pair of trees $(T_1, T_2) \in \mathcal{F} \times \mathcal{F}$; note that T_1 and T_2 may be the same. Let m' be the number of edges connecting T_1 and T_2. Let $\text{Euler}(T_1) = (u_1, \ldots, u_{|T_1|})$, $\text{Euler}(T_2) = (v_1, \ldots, v_{|T_2|})$, and $U = \max\{|T_1|, |T_2|\}$. We define the point set $P \subseteq [U] \times [U]$ to be $P = \{(i, j) \mid (u_i, v_j) \in E\}$. Suppose D is a set of edge failures including d_1 edges in T_1, d_2 in T_2, and d' non-tree edges. Removing D splits T_1 and T_2 into $d_1 + d_2 + 2$ connected subtrees and partitions $\text{Euler}(T_1)$ into a set $I_1 = \{[x_i, y_i]\}_i$ of $2d_1 + 1$ intervals and $\text{Euler}(T_2)$ into a set $I_2 = \{[w_i, z_i]\}_i$ of $2d_2 + 1$ intervals. For each pair i, j we query the 2D range reporting data structure for points in $[x_i, y_i] \times [w_j, z_j] \cap P$. However, we stop the query the moment it reports some point corresponding to a non-failed edge, i.e., one in $E - D$. Since there are $(2d_1 + 1) \times (2d_2 + 1)$ queries and each failed edge in D can only be reported in *one* such query, the total query time is $O(d_1 d_2 q + d')$, where q is either $\log \log n$ or $\log^\epsilon n$, depending on the space usage. See Figure 1 for an illustration.

Assuming that $m' \geq 1$, the space for the data structure restricted to T_1 and T_2 is $O(m' \log \log n)$ or $O(m')$. In order to avoid spending any space on pairs (T_1, T_2) with $m' = 0$, we maintain a hash table of tree-pairs with at least one edge between them. Since each non-tree edge contributes to the space
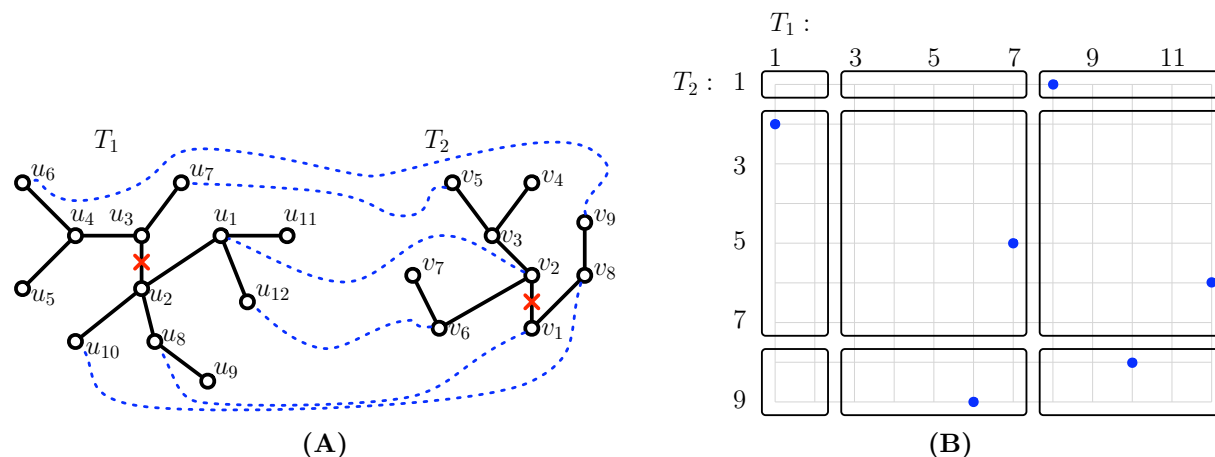
Figure 1: **(A)** Here T_1 and T_2 are two trees and $\mathrm{Euler}(T_1) = (u_1, \ldots, u_{12})$ and $\mathrm{Euler}(T_2) = (v_1, \ldots, v_9)$ are their vertices, listed by their first appearance in some Euler tours of T_1 and T_2. (It does not matter which Euler tour we pick.) There are six non-tree edges connecting T_1 and T_2, marked by dashed curves. If the edges (u_2, u_3) and (v_1, v_2) are removed, T_1 and T_2 are split into four subtrees, say T_1', T_2', T_3', T_4', and both $\mathrm{Euler}(T_1)$ and $\mathrm{Euler}(T_2)$ are split into three intervals, namely $X_1 = (u_1, u_2)$, $X_2 = (u_3, \ldots, u_7)$, $X_3 = (u_8, \ldots, u_{12})$, $Y_1 = (v_1)$, $Y_2 = (v_2, \ldots, v_7)$, and $Y_3 = (v_8, v_9)$. Each tree T_i' is identified with some subset of the intervals: $T_1', \ldots, T_4'$ are identified with $\{X_1, X_3\}, \{X_2\}, \{Y_1, Y_3\}$, and $\{Y_2\}$. **(B)** The point (i, j) (marked by a diamond) is in our point set if (v_i, u_j) is a non-tree edge. To determine if, for example, T_1' and T_4' are connected by an edge, we perform two 2D range queries, $X_1 \times Y_2$ and $X_3 \times Y_2$, and keep at most one point (i.e., a non-tree edge) for each query. In general, removing d_1 edges from T_1 and d_2 edges from T_2 necessitates $(2d_1 + 1)(2d_2 + 1)$ 2D range queries to determine incidences between all pairs of subtrees. In this example we require nine 2D range queries, indicated by boxes in the point set diagram.

of at most one tree pair (T_1, T_2), the overall space for $\mathbf{ET}(G, \mathcal{F})$ is $O(m \log \log n)$ or $O(m)$. For the last claim of the Theorem, observe that if a vertex u lies in an original tree $T_1 \in \mathcal{F}$, we can determine which tree in $\mathcal{F}'$ contains it by performing a predecessor search over the left endpoints of intervals in I_1. This can be accomplished in the minimum of $O(\frac{\log \log n}{\log \log \log n})$ time [64] or $O(\frac{\log d}{\log \log n})$ time [67] after $O(d^2)$ preprocessing on a $\Theta(\log n)$-bit word-RAM.

Corollary 2.1 demonstrates how $\mathbf{ET}(G, \cdot)$ can be used to answer connectivity queries avoiding edge and vertex failures.

COROLLARY 2.1. *Let T be any spanning tree of $G = (V, E)$. The data structure $\mathbf{ET}(G, \{T\})$ occupies space $O(m \log \log n)$ (or $O(m)$) and supports the following operations. Given a set $D \subset E$ of edge failures, d of which are tree edges and d' are non-tree edges, D can be processed in $O(d^2 \log \log n + d')$ time (or $O(d^2 \log^\epsilon n + d')$ time) so that connectivity queries in the graph $(V, E - D)$ can be answered in $O(\min \left\{ \frac{\log \log n}{\log \log \log n}, \frac{\log d}{\log \log n} \right\})$ time. If $D \subset V$ is a set of vertex failures, let $d = \sum_{v \in D} \deg_T(v)$ be the sum of their T-degrees. The update time is $O(d^2 \log \log n)$ (or $O(d^2 \log^\epsilon n)$) and the query time*

is $O(\min \left\{ \frac{\log \log n}{\log \log \log n}, \frac{\log d}{\log \log n} \right\})$.

Proof. Using $\mathbf{ET}(G, \{T\})$ we split T into $d+1$ subtrees and $\mathrm{Euler}(T)$ into a set I of $2d+1$ connected intervals, in which each connected subtree is made up of some subset of the intervals. Using $O(d^2)$ 2D range queries, in $O(d^2 \log \log n + d')$ time we find at most one edge connecting each pair in $I \times I$. (In the case of vertex failures, no range queries are performed for the intervals containing singleton vertices in D.) In $O(d^2)$ time we find the connected components of $E - D$ or $V - D$ and store with each interval a representative vertex from its component. To answer a query (u, v) we only need to determine which subtree u and v are in, which involves two predecessor queries over the left endpoints of intervals in I. This takes $O(\min \left\{ \frac{\log \log n}{\log \log \log n}, \frac{\log d}{\log \log n} \right\})$ time.

Corollary 2.1 motivates us to look for conditions under which G contains a low degree spanning forest, say with degree at most s. In the next section we show that although G may not have a degree-s spanning forest, there are $O(n/s)$ critical nodes that, if they were removed, would let the remaining graph be spanned by a degree-s spanning forest.

3 A New Graph Decomposition Theorem

Let $G = (V, E)$ be an undirected graph and $U \subseteq V$ be a set of *terminals.* We call a forest $T \subseteq E$ a *Steiner forest* for U if $u, v \in U$ are connected in T if and only if they are connected in G. Fürer and Raghavachari [43] proved that the minimum degree spanning forest (if $U = V$) and minimum degree Steiner forest could be approximated to within 1 of optimal in polynomial time.[6]

THEOREM 3.1. *(Fürer and Raghavachari [43]) Suppose G contains a Steiner forest for U with maximum degree Δ^*. A Steiner forest T for U with maximum degree $\Delta^* + 1$ can be computed in $O(|U|m \log |U|)$ time.*

Let $\mathsf{FR\text{-}Tree}(G, U)$ be the procedure that computes T. Our decomposition theorem is not concerned with Δ^*, but with *other* properties of the forest T. In order to see how these properties arise, we sketch how the $\mathsf{FR\text{-}Tree}(G, U)$ algorithm works in the simpler case in which $U = V$. Let $\Delta(G')$ denote the maximum degree in the graph G'.

The algorithm begins with any spanning forest T_0 and iteratively tries to *improve* T_0, yielding $T_1, T_2, \ldots, T_\omega$, such that (i) $\Delta(T_{i+1}) \leq \Delta(T_i)$, and (ii) the set of degree-$\Delta(T_i)$ nodes in T_{i+1} is a *strict* subset of the degree-$\Delta(T_i)$ nodes in T_i. The number of improvements is clearly finite. Since any tree contains fewer than $n/(k-1)$ nodes with degree at least k, for $k \geq 2$, the total number of improvements is at most $\sum_{k=\Delta(T_\omega)}^{\Delta(T_0)} n/(k-1) = O(n \log \frac{\Delta(T_0)}{\Delta(T_\omega)}) = O(n \log n)$.

The $\mathsf{FR\text{-}Tree}$ algorithm only searches for a particular class of improvements that can be found in linear time, leading to an $O(mn \log n)$ time bound. Let T_0 be the current spanning tree. All vertices with degree $\Delta(T_0)$ and $\Delta(T_0) - 1$ are initially marked *bad* and all others *good*. (In the diagrams below white nodes have degree $\Delta(T_0)$, gray nodes have degree $\Delta(T_0) - 1$, and black nodes have degrees less than $\Delta(T_0) - 1$.) The simplest *single-swap* improvement arises if there is a non-T_0 edge (u, v) such that u and v are good (black) and a bad vertex x with degree $\Delta(T_0)$ appears on the unique cycle of $T \cup \{(u, v)\}$. In this case we choose any edge (x, y) incident to x on the cycle and set
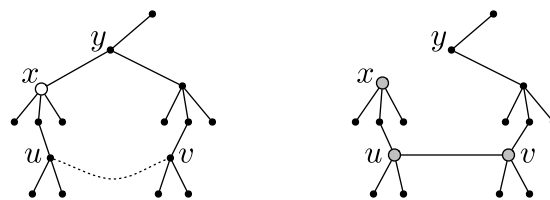

Figure 2: Swapping (u, v) for (x, y) yields a new tree with at least one fewer node with degree $\Delta(T_0)$.

$T_1 \leftarrow T_0 - \{(x, y)\} \cup \{(u, v)\}$, thereby eliminating a degree-$\Delta(T_0)$ vertex (namely x, and perhaps even y) but possibly increasing the number of degree-$(\Delta(T_0) - 1)$ vertices (namely u and v).

In general the $\mathsf{FR\text{-}Tree}$ algorithm considers improvements composed of an arbitrarily large number of edge-swaps. While there exists an unscanned edge (u, v) where both u and v are marked good, it marks all bad vertices *good* on the fundamental cycle of $T_0 \cup \{(u, v)\}$. Thus, a formerly-bad good vertex is one whose degree can be reduced by 1 via a sequence of edge-swaps that does not introduce any degree-$\Delta(T_0)$ vertices. If a degree-$\Delta(T_0)$ vertex is ever marked good, an improvement has been detected and the sequence of swap edges that created it can easily be reconstructed. See Figure 3. Every
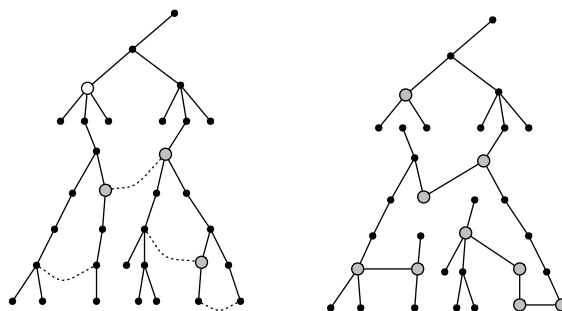

Figure 3: A sequence of edge-swaps that reduces the number of degree-$\Delta(T_0)$ vertices but may increase the number of degree-$(\Delta(T_0) - 1)$ vertices.

time this procedure finds an improvement we obtain a new spanning tree and begin a search for another improvement from scratch. Let T_ω be the spanning tree for which this procedure fails to find an improvement. Let B be the set of vertices still marked *bad*. By definition B includes *all* vertices with degree $\Delta(T_\omega)$ in T_ω and some subset of the vertices with degree $\Delta(T_\omega) - 1$. Consider what happens to G and T_ω if we removed all B-vertices from the graph. $\mathsf{FR\text{-}Tree}$'s search for improvements guarantees that $T_\omega - B$ is a *spanning* forest of the graph $G - B$.

---

[6]Fürer and Raghavachari [43] claimed a running time of $O(|U|m\alpha(m, n) \log |U|)$. The $\alpha(m, n)$ factor can be removed using the incremental-tree set-union structure of Gabow and Tarjan [44].

Indeed, if there *were* an edge (u, v) connecting two distinct trees of $T_\omega - B$ then all B-vertices on the fundamental cycle of $T_\omega \cup \{(u, v)\}$ would have been marked *good* and therefore u and v would not have been in distinct trees of $T_\omega - B$ after all. In general, the output of FR-Tree(G, U) is the pair (T_ω, B). Theorem 3.2 summarizes the properties of the FR-Tree algorithm that we actually use.

THEOREM 3.2. *([43]) The* FR-Tree(G, U) *algorithm returns a pair* (T, B)*, where T is a Steiner forest for U and $B \subset V$ comprises all vertices with T-degree $\Delta(T)$ and some subset of vertices with T-degree $\Delta(T) - 1$. If $u, v \in U$ are disconnected in $T - B$ then they are also disconnected in $G - B$.*

The degree $\Delta(T - B)$ is by definition at most $\Delta(T) - 1$, which may still be too large. Theorem 3.3 shows that by iteratively applying the FR-Tree algorithm to the components of $T - B$ we can reduce the maximum degree to any desired bound $s \geq 3$, at the cost of increasing the set B of "bad" vertices.

THEOREM 3.3. **(The Decomposition Theorem)** *Let $U \subseteq V$ be a terminal set in a graph $G = (V, E)$ and $s \geq 3$. There is an algorithm* Decomp(G, U, s) *that returns a pair* (T, B) *such that the following hold.*

1. *T is a Steiner forest for U and $T - B$ is a Steiner forest for $U - B$.*

2. *$\Delta(T - B) \leq s$.*

3. *$|B| < |U|/(s - 2)$ and $|B \cap U| < |U|/(s - 1)$.*

The running time of Decomp *is* $O(|U| m \log |U|)$*.*

In the remainder of this section we give the Decomp(G, U, s) algorithm and prove Theorem 3.3. An invocation of Decomp consists of the following three steps.

Step 1. Let (T', B') be the output of FR-Tree(G, U). If $\Delta(T') \leq s$ then we are done, and return the pair $(T', \emptyset)$.

Step 2. Partition the edge set of T' into minimal trees $\{t_i\}$ such that the leaves of each t_i are either B'-nodes or leaves of T', and hence U-nodes. Let $B'[t_i]$ be the B' nodes in t_i and $V[t_i]$ be the set of all vertices in $G - B'$ reachable from vertices in $V(t_i) - B'[t_i]$. (When $U = V$, $V[t_i]$ is exactly $V(t_i) - B'[t_i]$; in general $V[t_i]$ may contain vertices outside of $V(T')$.) Let $G[t_i]$ be the graph whose vertex set is $V[t_i] \cup B'[t_i]$ and whose edge set includes all edges induced by $V[t_i]$ and, for each

$u \in B'[t_i]$, the unique T'-edge connecting u to $V(t_i)$. For each t_i, obtain a pair (T_i, B_i) by recursively calling Decomp$(G[t_i], (V[t_i] \cap U) \cup B'[t_i], s)$. Observe that $B'[t_i]$ are included as terminals in the recursive call, even if they are not members of U. See Figure 4 for an illustrative example.

Step 3. Return the pair (T, B) where

$$T = \bigcup_i T_i \quad \text{and} \quad B = B' \cup \bigcup_i B_i.$$

We need to establish all the claims: that $T - B$ is, in fact, a Steiner forest of $U - B$ with maximum degree s, that B has the right cardinality, and that the running time is $O(|U| m \log |U|)$.

If the algorithm halts at Step 1 then T' is, by Theorem 3.2, a Steiner forest for U in G. Suppose that the algorithm does not halt at Step 1 and let $P(u_0, u_k)$ be a path in T' between $u_0, u_k \in U$. Partition it into subpaths $P(u_0, u_1), \ldots, P(u_{k-1}, u_k)$, where $u_1, \ldots, u_{k-1}$ are all the B'-nodes encountered on the path. By construction, each $P(u_i, u_{i+1})$ is completely contained in some tree t_i and the endpoints of this path are terminals in the recursive call to Decomp$(G[t_i], (V[t_i] \cap U) \cup B'[t_i], s)$, so, by the inductive hypothesis, the tree T_i returned contains a (possibly different) path between u_i and u_{i+1}. By Theorem 3.2 again, the graphs $\{G[t_i]\}$ intersect only at B'-nodes, which necessarily occur as leaves in the $\{T_i\}$ trees, so the edge-set $T = \bigcup_i T_i$ returned is, in fact, a Steiner forest for U. By Theorem 3.2, all nodes in B have T-degree at least s and all nodes in $T - B$ have T-degree at most s. Moreover, if $u, v \in U$ are disconnected in $T - B$ then they are disconnected in $G - B$. This follows from Theorem 3.2 if u and v are in different trees t_i, t_j, and by induction on the output of Decomp$(G[t_i], (V[t_i] \cap U) \cup B'[t_i], s)$ if u, v are both in t_i.

We now prove that B has the claimed cardinality, using the property that all B-nodes have degree at least s in T.

LEMMA 3.1. *Let T be any minimal Steiner tree for U. The number of nodes in T with T-degree at least s is at most $g(|U|) = \lfloor \frac{|U| - 2}{s - 2} \rfloor$. The number of U-nodes in T with T-degree at least s is at most $h(|U|) = \lfloor \frac{|U| - 2}{s - 1} \rfloor$.*

Proof. Due to the minimality of T, all leaves are necessarily U-nodes. Moreover, we can assume without loss of generality that all internal nodes have degree at least 3, by splicing out paths of
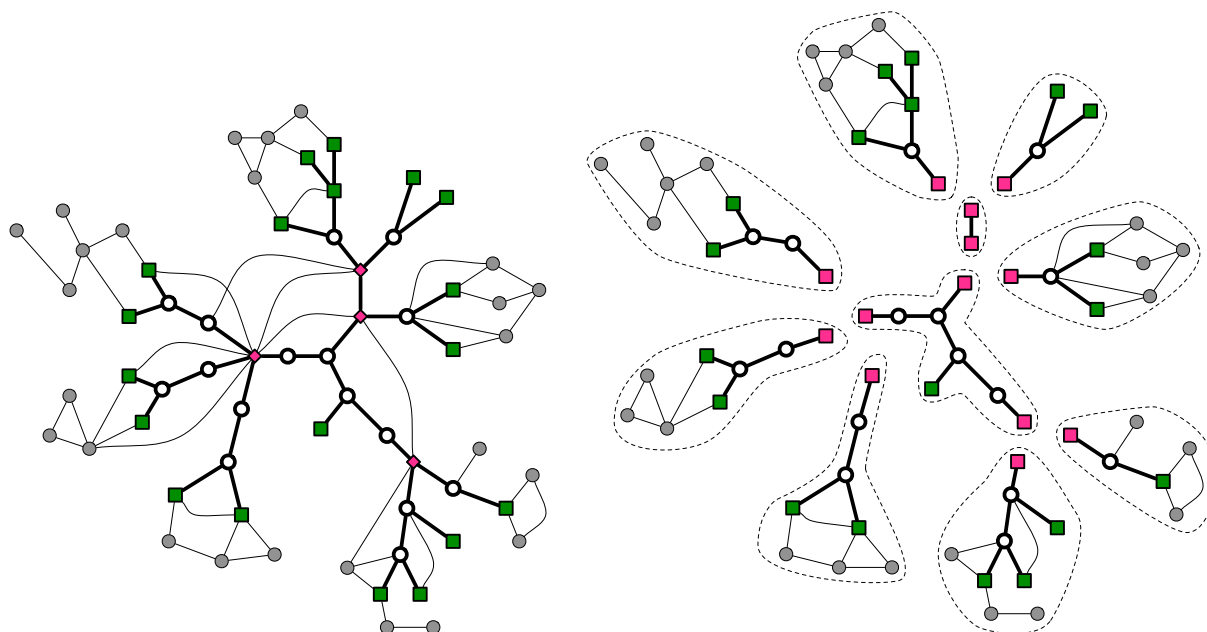
Figure 4: Left: the output of FR-Tree. Square green nodes are terminals; pink diamonds are B'-nodes (and may be terminals); thick edges are part of T'; gray vertices are outside $V(T')$. Right: detaching the edges adjacent to B' nodes creates ten subtrees; non-$V(T')$ nodes are connected to at most one subtree; Decomp is called recursively on each subgraph; B'-nodes have degree 1 in these recursive calls and are designated terminals (square nodes).

degree-2 vertices. When $|U| \leq s - 1$ we have $g(|U|) = 0$ and when $|U| \leq s$ we have $h(|U|) = 0$. The claimed bounds on g and h hold when there is exactly one internal node. In general, choose an internal node u adjacent to exactly one internal node. If u is adjacent to at least $s-1$ leaves then it contributes 1 to the $g(|U|)$ tally; remove its incident leaves and designate u a U-node. We preserve the property that all leaves are U-nodes, and since the net loss in the number of U-nodes is at least $s - 2$, we have $g(|U|) \leq g(|U|-(s-2))+1$. Observe that u only contributes to the $h(|U|)$ tally if it is already a U-node. In this case we have a loss of $s-1$ U-nodes, which implies that $h(|U|) \leq h(|U|-(s-1))+1$. The claimed bounds on g and h follow by induction on $|U|$.

To analyze the running time we imagine that a *single* global Steiner tree for U is being maintained, which is the union of the current Steiner trees in the deepest recursive calls. The initial tree provided to a call to FR-Tree is therefore just a fragment of the global Steiner tree, whose maximum degree is some $k \geq s + 1$. Each iteration of this call to FR-Tree, except the last, finds an *improvement*, which

reduces the number of maximum-degree nodes in its fragment by at least one. Say a k-improvement is one that reduces the number of degree-k nodes. If the current global Steiner tree has maximum degree k, the total number of k-improvements that can be found, in all recursive calls, is at most $|U|/(k-2)$. The initial value of k is certainly at most $|U|$. Since each improvement takes linear time, the total time for all improvements is $O(m) \cdot \sum_{k=s+1}^{|U|} |U|/(k-2) = O(|U|m \log(|U|/s))$.

4 The Low Degree Hierarchy

We can apply Theorem 3.3 iteratively to create a *low degree hierarchy*. Fix $s = 4$ and generate a set of pairs $\{(T_i, B_i)\}$ as follows:

$$(T_0, B_0) \leftarrow \mathsf{Decomp}(G, V, 4),$$
$$(T_1, B_1) \leftarrow \mathsf{Decomp}(G, B_0, 4),$$
$$\cdots$$
$$(T_i, B_i) \leftarrow \mathsf{Decomp}(G, B_{i-1}, 4),$$
$$\cdots$$
$$(T_p, \emptyset) \leftarrow \mathsf{Decomp}(G, B_{p-1}, 4).$$

In other words, the "bad" vertices for T_0 form the terminal set for T_1 and in general, the bad vertices for T_{i-1} form the terminal set for T_i. We end, of course, at the first T_p with degree at most $s = 4$, so $B_p = \emptyset$. It follows from Theorem 3.3 that $|B_0| < n/3$ and in general, that $|B_i| < |B_{i-1}|/2$, so $p < \log n - 1$ levels suffice.

Define $\mathcal{T}_i$ to be the set of trees in the forest $T_i - B_i$ and $\mathcal{T}$ to be the set of all trees in $\mathcal{T}_0, \ldots, \mathcal{T}_p$, as if each forest were on a disjoint vertex set. Theorem 3.3 implies that $\mathcal{T}_i$ has two useful properties: it has maximum degree 4, and it is a Steiner forest for $B_{i-1} - B_i$.

DEFINITION 1. *Suppose $\tau_i \in \mathcal{T}_i$ and $\tau_{i'} \in \mathcal{T}_{i'}$, $i \leq i'$. We say τ_i is a* descendant *of $\tau_{i'}$ if a connected component of $G - B_{i'}$ contains $V(\tau_{i'})$ and at least one vertex of $V(\tau_i)$.*

Observe that if $V(\tau_i) \cap B_{i'} = \emptyset$ then τ_i can only have one ancestor at level i'; if it had two distinct ancestors then they would be connected by a path in $G - B_{i'}$, contradicting Theorem 3.3. Unfortunately, it seems that $V(\tau_i)$ *can* intersect $B_{i'}$, so in general the ancestry relation between trees in $\mathcal{T}$ induces a $(p+1)$-level *dag*, not a rooted tree. Algorithmically it is much easier to deal with trees rather than dags. For this reason we define a variant hierarchy that is more structured, but loses some useful properties of $\mathcal{T}$.

DEFINITION 2. *Define $\mathcal{C}_i$ to be the set of connected components of $G - (B_i \cup B_{i+1} \cup \cdots \cup B_{p-1})$ containing at least one B_{i-1} (terminal) vertex. Suppose $\gamma_i \in \mathcal{C}_i$ and $\gamma_{i'} \in \mathcal{C}_{i'}$, where $i \leq i'$. We say γ_i is a* descendant *of $\gamma_{i'}$, written $\gamma_i \preceq \gamma_{i'}$, if $V(\gamma_i) \cap V(\gamma_{i'}) \neq \emptyset$.*

Lemma 4.1 identifies the critical properties of $\{\mathcal{C}_i\}$ used by our algorithm.

LEMMA 4.1. *Consider the hierarchy of components $\{\mathcal{C}_i\}_{i \in [0,p]}$.*

1. *Each $\gamma \in \mathcal{C}_i$ has at most one ancestor in $\mathcal{C}_{i'}$, for each $i' \in [i, p]$.*

2. *$V(\gamma) \subseteq V(\gamma')$ for each $\gamma \preceq \gamma'$.*

3. *If $(u, v) \in E$ and $u \in V(\gamma), v \in V(\gamma')$, then $\gamma \preceq \gamma'$ or $\gamma' \prec \gamma$.*

4. *If $\gamma \in \mathcal{C}_i$, the terminals $V(\gamma) \cap B_{i-1}$ are contained in a single tree in $\mathcal{T}_i$, denoted $\tau(\gamma)$.*

Proof. For Part 1, note that any two distinct components $\gamma', \gamma'' \in \mathcal{C}_{i'}$ have $V(\gamma') \cap V(\gamma'') = \emptyset$. Since, by construction, $V(\gamma) \cap (B_{i'} \cup \cdots \cup B_{p-1}) = \emptyset$, γ cannot share vertices with both γ' and γ''. We now turn to Part 2. Suppose $\gamma \in \mathcal{C}_i, \gamma' \in \mathcal{C}_{i'}$ with $i < i'$. If γ and γ' share one vertex then $V(\gamma) \subset V(\gamma')$ since γ is connected and $V(\gamma) \cap (B_{i'} \cup \cdots \cup B_{p-1}) = \emptyset$. If Part 3 were false then γ and γ' would be unrelated. Without loss of generality, suppose γ' is at a higher level than γ, and let γ'' be the ancestor of γ at the same level as γ'. Thus, γ', γ'' are two distinct components in some $\mathcal{C}_i$. Part 2 implies $u \in V(\gamma'')$, meaning γ' and γ'' are joined by an edge (u, v), and are therefore not distinct components in $\mathcal{C}_i$. For Part 4, consider a tree $\tau \in \mathcal{T}_i = T_i - B_i$. By Theorem 3.3, τ spans the terminals (B_{i-1}-nodes) in a connected component of $G - B_i$. A $\gamma \in \mathcal{C}_i$ represents a connected component in $G - (B_i \cup \cdots \cup B_{p-1})$, so if $V(\gamma)$ intersects $V(\tau)$ at one terminal, every terminal of $V(\gamma)$ must be contained in $V(\tau)$.

Lemma 4.1(1) (unique ancestors) shows that the ancestry relationship on $\mathcal{C}_0, \ldots, \mathcal{C}_p$ can be succinctly encoded as a forest of rooted trees. Let $\mathcal{C}$ be the *component hierarchy* defined by the $\prec$ relation. The nodes of $\mathcal{C}$ are in one-to-one correspondence with the components of $\mathcal{C}_0, \ldots, \mathcal{C}_p$, where $\mathcal{C}_0$ form the leaves of $\mathcal{C}$. Slightly abusing notation, we shall say "$\gamma \in \mathcal{C}$" to mean that γ is a node in $\mathcal{C}$ or that γ is a component in some $\mathcal{C}_i$.

4.1 Stocking the Low Degree Hierarchy
Our goal is to supplement $\mathcal{C}$ and $\mathcal{T}$ with useful data structures that allow us to reconnect the graph after a set of vertices fail. Recall that $\mathcal{T}$ is composed of trees with maximum degree at most 4. If a single tree $\tau \in \mathcal{T}$ experiences the failure of some vertex set $D \subset V$, we can find *individual* edges that reconnect the subtrees of $\tau - D$ using $O(|D|^2)$ 2D range queries (Theorem 2.1). However, individual edges are, in general, insufficient to reconnect the subtrees. There could be long paths that go through vertices that appear in ancestors or descendants of τ in $\mathcal{T}$. In order to quickly detect the existence of these paths we follow an idea from [21] and introduce *artificial* edges that capture connectivity via paths. We do not want to add too many artificial edges, for two reasons. First, they take up space, which we want to conserve, and second, after deleting vertices from the graph the *validity* of many artificial edges may be cast into doubt. Any invalid artificial edges must be ignored when reestablishing connectivity, so it is important that the algorithm

not encounter too many of these edges. Before saying exactly how artificial edges are added we must recall the concept of a $d_\star$-*adjacency list* [39]. Recall that $d_\star$ is the maximum number of vertex failures.

DEFINITION 3. ([39]) *Let* $L = (v_1, v_2, \ldots, v_r)$ *be a list of vertices and* $d_\star \geq 1$ *be an integer. The* $d_\star$-*adjacency edges* $\Lambda_{d_\star}(L)$ *connect all vertices at distance at most* $d_\star + 1$ *in the list* L:

$$\Lambda_{d_\star}(L) = \{(v_i, v_j) \mid 1 \leq i < j \leq r \text{ and } j - i \leq d_\star + 1\}$$

LEMMA 4.2. *The following properties hold for any vertex list* L:

1. $\Lambda_{d_\star}(L)$ *contains fewer than* $(d_\star + 1)|L|$ *edges.*

2. *If a set* D *of at most* $d_\star$ *vertices are removed from* L *then the subgraph of* $\Lambda_{d_\star}(L)$ *induced by* $L - D$ *remains connected.*

3. *Suppose* L *is partitioned into consecutive sublists* L_1 *and* L_2. *Then at most* $O(d_\star^2)$ *edges from* $\Lambda_{d_\star}(L)$ *cross the partition* (L_1, L_2).

Proof. Part (1) is trivial, as is (2), since each pair of consecutive undeleted vertices is at distance at most $d_\star + 1$, and therefore adjacent. Part (3) is also trivial: the number of edges connecting any prefix and suffix of L is at most $(d_\star + 1)(d_\star + 2)/2$.

Fix a $\gamma_i \in \mathcal{C}_i$ and let $\gamma_{i+1}, \ldots, \gamma_p$ be its ancestors in $\mathcal{C}$. Recall that the terminals of γ_i are contained in a single tree $\tau(\gamma_i) \in \mathcal{T}_i$. The mapping τ is not necessarily injective: one tree in $\mathcal{T}_i$ could be the host for many components in $\mathcal{C}_i$. Define $A(\gamma_i, \gamma_j)$ to be a list of the terminals in $V(\gamma_j)$ that are adjacent to at least one vertex in $V(\gamma_i)$, listed according to an Euler tour Euler$(\tau(\gamma_j))$. (Recall that the terminals in $V(\gamma_j)$ are the vertex set $V(\gamma_j) \cap (B_{j-1} - (B_j \cup \cdots \cup B_{p-1}))$.) Let $A(\gamma_i)$ be the concatenation of $A(\gamma_i, \gamma_{i+1}), \ldots, A(\gamma_i, \gamma_p)$. We interpret elements of $A(\gamma_i)$ as the principal copies of vertices in $\mathcal{T}$. If u is a terminal in $V(\gamma_j)$, its copy in $\tau(\gamma_j)$ is the *principal copy* of u in $\mathcal{T}$; all other occurrences of u in $\mathcal{T}$ (at levels other than j) are *non-principal* copies.

DEFINITION 4. *The multigraph* H *is on the vertex set of* $\mathcal{T}$. *The edge set of* H *includes* all *tree edges in* $\{\mathcal{T}_i\}_i$. *For each* $(u, v) \in E$, H *contains an* original *edge connecting the principal copies of* u *and* v. *For each component* $\gamma \in \mathcal{C}$, H *includes* $\Lambda(\gamma) \overset{\text{def}}{=} \Lambda_{d_\star}(A(\gamma))$. *Each edge in* H *is labeled with its provenance: either* original, tree edge, *or the name of a* γ *if it appears in* $\Lambda(\gamma)$. *Note that* H *may contain multiple edges with the same endpoints, but with different provenances.*

Lemma 4.3 exhibits the two salient properties of $\Lambda(\gamma)$: that it encodes useful connectivity information and that it is economical to effectively destroy $\Lambda(\gamma)$ when it is no longer valid, often in time sublinear in $|\Lambda(\gamma)|$.

LEMMA 4.3. ([39]) *Consider a* $\Lambda(\gamma_i) \subset E(H)$.

1. *Suppose* $d \leq d_\star$ *vertices fail, none of which are in* $V(\gamma_i)$, *and suppose* u *and* v *are in components of ancestors of* γ_i *and are each adjacent to at least one vertex in* $V(\gamma_i)$. *Then* u *and* v *remain connected in the original graph and remain connected in* H.

2. *Suppose the proper ancestors of* γ_i *are* $\gamma_{i+1}, \ldots, \gamma_p$ *and a total of* f *edges are removed from* $\tau(\gamma_{i+1}), \ldots, \tau(\gamma_p)$, *breaking their Euler tours into intervals* $I_1, \ldots, I_{p-i+2f}$. *Then at most* $O(d_\star^2(p+f))$ *edges of* $\Lambda(\gamma_i)$ *connect distinct intervals* $I_j, I_{j'}$.

Proof. For Part (1), the vertices u and v are connected in the original graph because they are each adjacent to vertices in $V(\gamma_i)$ and, absent any failures, all vertices in $V(\gamma_i)$ remain connected. By Definition 4, u and v appear in $\Lambda(\gamma_i)$ and, by Lemma 4.2, $\Lambda(\gamma_i)$ remains connected after the removal of any d vertices. Turning to Part (2), recall from Definition 4 that $A(\gamma_i)$ was the concatenation of $A(\gamma_i, \gamma_{i+1}), \ldots, A(\gamma_i, \gamma_p)$ and each $A(\gamma_i, \gamma_{i'})$ was ordered according to an Euler tour of $\tau(\gamma_{i'}) \in \mathcal{T}_{i'}$. Removing f edges from $\tau(\gamma_{i+1}), \ldots, \tau(\gamma_p)$ separates their Euler tours (and, hence, the lists $\{A(\gamma_i, \gamma_{i'})\}_{i'}$) into at most $2f + p - i$ intervals. By Lemma 4.2 at most $(2f + p - i) \cdot O(d_\star^2)$ edges from $\Lambda(\gamma_i)$ connect distinct intervals. In other words, in order to "logically" delete $\Lambda(\gamma_i)$ it suffices to delete $O(d_\star^2(p+f))$ edges from $\Lambda(\gamma_i)$ since all remaining edges do not add to the connectivity of the remaining graph.

We apply Theorem 2.1 and generate an ET-structure $\mathbf{ET}(H, \mathcal{T})$ for H. Lemma 4.4 bounds the space for the overall data structure.

LEMMA 4.4. *Given a graph* G *with* m *edges,* n *vertices, and a parameter* $d_\star \geq 1$, *the* $d_\star$-*failure connectivity oracle consists of* $\mathcal{C}, \mathbf{ET}(H, \mathcal{T})$, *and various linear-space data structures supporting navigation around* $\mathcal{C}$. *The space required by the oracle is* $O(d_\star m \log n \log \log n)$ *or* $O(d_\star m \log n)$, *depending on the 2D range searching structure used in* $\mathbf{ET}(H, \mathcal{T})$, *and its construction time is* $O(mn \log n)$.

Proof. The number of vertices in H is at most $(p+1)n$, n per $\mathcal{T}_i$. (This is a pessimistic bound. We are unable to conceive of any graph G for which this is achieved.) The number of tree edges in H is less than $(p+1)n$ and the number of original edges in H is m. Each original edge contributes a vertex to at most p lists $A(\gamma)$, and each member of $A(\gamma)$ contributes at most $d_\star + 1$ edges to $\Lambda(\gamma)$. The number of edges in H is therefore less than $m + (p+1)n + p(d_\star + 1)m = O(d_\star m \log n)$. By Theorem 2.1, each edge in H contributes $O(\log \log n)$ or $O(1)$ space to $\mathbf{ET}(H, \mathcal{T})$. Regarding construction time, by Theorem 3.3 the time to compute (T_0, B_0) is $O(mn \log n)$, and more generally, the time to compute (T_{i+1}, B_{i+1}) is $O(m|B_i| \log |B_i|)$ time, where $|B_i| < n/(s-2)^i = n/2^i$ decays geometrically with i. Thus, the total time to compute $\mathcal{T}$ and $\mathcal{C}$ is $O(mn \log n)$.

5 Recovery From Failures

In this section we describe how, given a set of $d \leq d_\star$ failed vertices, the data structure can be updated in time $\tilde{O}(d^2 d_\star^2)$ such that connectivity queries can be answered in $O(d)$ time. Section 5.1 gives the algorithm to delete failed vertices and Section 5.2 gives the query algorithm and proof of correctness.

5.1 Deleting Failed Vertices Let $D \subset V$ be the set of d failed vertices.

Step 1. Begin by marking any $\gamma \in \mathcal{C}$ *affected* if $V(\gamma) \cap D \neq \emptyset$, and mark the corresponding tree $\tau(\gamma) \in \mathcal{T}$ affected as well. For each affected $\tau(\gamma)$, mark each D-node and its incident tree edges as deleted. This breaks up $\tau(\gamma)$ into *affected subtrees*, which must be reconnected, if possible.

LEMMA 5.1. *The number of affected trees is at most $d(p+1)$. The number of affected subtrees is at most $4d(p+1)$.*

Proof. By Lemma 4.1, any $u \in D$ appears in at most $p+1$ components of $\mathcal{C}$. Since all failed vertices have degree at most $s = 4$ in the $\mathcal{T}$ trees in which they appear, there are at most $4d(p+1)$ affected subtrees.

Recall from the discussion above that if γ is affected then $V(\gamma)$ contains failed vertices and the connectivity provided by $\Lambda(\gamma)$ is presumed invalid. By Lemma 4.3 we can logically delete $\Lambda(\gamma)$ by ignoring $O(d_\star^2)$ edges for *each* of $O(pd)$ breaks in the list $A(\gamma)$. Since there are at most $O(pd)$ affected (sub)trees, the number of edges that need to be

ignored is $O((pd)^2 d_\star^2)$. Let H' denote the graph H with these $O((pd)^2 d_\star^2)$ edges removed.

Step 2. We now attempt to reconnect all affected subtrees using valid edges in H'. Let R be a graph whose vertex set $V(R)$ represent the $O(pd)$ affected subtrees such that $(t_1, t_2) \in E(R)$ if t_1 and t_2 are connected by an edge in H'. Using the structure $\mathbf{ET}(H, \mathcal{T})$ (see Theorem 2.1) we populate the edge set of R in time $O(|V(R)|^2 q + (pd)^2 d_\star^2)$, where $q = \log \log n$ or $\log^\epsilon n$, depending on the space of the 2D range structure [20]. For each 2D range query, we halt the enumeration of points/edges as soon as an H'-edge is reported. Recall that a point/edge is tagged with its provenance, so we can check in $O(1)$ time whether it came from an affected $\Lambda(\gamma)$ and must be discarded. Since $V(R) = O(pd)$ and $p < \log n$, the time to perform these queries is $O(d^2(q + d_\star^2) \log^2 n)$. In $O(|E(R)|) = O((pd)^2)$ time we determine the connected components of R.

This concludes the deletion algorithm. The running time is dominated by Step 2. By building several copies of the data structure with exponentially decaying values of $d_\star$ we can guarantee that $d \geq d_\star/2$ in some copy of the data structure, so the deletion time becomes $O(d^2(d^2 + q) \log^2 n)$.

5.2 Answering a Connectivity Query To answer a connectivity query between u and v we first check to see if there is a path between them that avoids affected trees, then consider paths that intersect one or more affected trees.

Step 1. We first find the lowest-level components in $\mathcal{C}$ containing u and v; let them be $\gamma(u)$ and $\gamma(v)$. If $\gamma(u)$ is unaffected, let γ_1 be the most ancestral unaffected ancestor of $\gamma(u)$, and let γ_2 be defined in the same way for $\gamma(v)$. If $\gamma_1 = \gamma_2$ then $V(\gamma_1)$ contains u and v but no failed vertices. If this is the case we declare u and v *connected* and stop.

We can find γ_1 and γ_2 in $O(\log p) = O(\log \log n)$ time using a binary search over the ancestors of $\gamma(u)$ and $\gamma(v)$. Alternatively, we can find them in time $O(\log d)$, independent of n, using relatively simple data structures. Fix any postordering of the nodes of $\mathcal{C}$. Find the predecessor γ_{pred} and successor γ_{succ} of $\gamma(u)$ among all components whose *terminal set* contains a D-vertex. There are at most d such nodes, so the cost to find them is $O(\log d)$ via binary search. Let $\gamma_{\mathrm{pred}}^{\mathrm{lca}}, \gamma_{\mathrm{succ}}^{\mathrm{lca}}$ be the least common ancestors of $\gamma(u)$ and $\gamma_{\mathrm{pred}}, \gamma_{\mathrm{succ}}$, respectively. Without loss of generality suppose

$\gamma_{\text{pred}}^{\text{lca}}$ is closer to $\gamma(u)$. Since $V(\gamma_{\text{pred}}^{\text{lca}}) \cap D \neq \emptyset$, $\gamma_{\text{pred}}^{\text{lca}}$ is affected. If $\gamma_{\text{pred}}^{\text{lca}}$ is at depth k from its root in $\mathcal{C}$, the node γ_1 we are looking for is the ancestor of $\gamma(u)$ at depth $k+1$. Refer to [11, 12] for linear space data structures for least common ancestor and level ancestors.

Step 2. We now try to find vertices u' and v' in affected subtrees that are connected to u and v respectively. If $\gamma(u)$ is affected then $u' = u$ clearly suffices, so we only need to consider the case when $\gamma(u)$ is unaffected and γ_1 exists. Recall from Definition 4 that $A(\gamma_1)$ is the list of terminals in proper ancestors of γ_1 that are adjacent to some vertex in $V(\gamma_1)$. We scan $A(\gamma_1)$ looking for *any* non-failed vertex u' adjacent to $V(\gamma_1)$. Since $V(\gamma_1)$ is unaffected, u is connected to u', and since all of γ_1's proper ancestors are affected, u' must appear in an affected subtree in $\mathcal{T}$. Since there are at most d failed vertices we must inspect at most $d + 1$ elements of $A(\gamma_1)$. This takes $O(d)$ time to find u' and v', if they exist. If one or both of u' and v' does not exist we declare u and v *disconnected* and stop.

Step 3. We have the principal copies of u' and v' in $\mathcal{T}$. In $O(\min\left\{\frac{\log \log n}{\log \log \log n}, \frac{\log d}{\log \log n}\right\})$ time we find the affected subtrees t_1' and t_2' containing u' and v', respectively, via predecessor search [64, 67] over the left endpoints of the Euler-tour intervals that remain after deleting D and their incident tree edges. Note that t_1' and t_2' are vertices in R, from Step 2 of the deletion algorithm. We declare u and v to be connected if and only if t_1' and t_2' are in the same connected component of R. This takes $O(1)$ time.

The running time of this procedure is dominated by the cost of finding u' and v'.

LEMMA 5.2. *The query algorithm correctly determines whether u and v are connected in $G - D$, in $O(d)$ time.*

Proof. If the query algorithm halts in Step 1 it is because both u and v are in the unaffected component γ_1, and since $V(\gamma_1) \cap D = \emptyset$, all vertices in γ_1 are still connected. If the query algorithm halts in Step 2 it is because $u \in V(\gamma_1)$, $v \notin V(\gamma_1)$, and $A(\gamma_1) - D = \emptyset$. Since $A(\gamma_1)$ contains *all* vertices adjacent to γ_1 there can be no path from u to v in $G - D$.

At Step 3 we have discovered u', v' such that u is connected to u', which appears as a principal vertex in an affected subtree t_1' and similarly for v, v', and

t_2'. Since t_1', t_2' are vertices in R, the correctness of the query algorithm hinges on whether the graph R correctly represents the connectivity between affected subtrees.

We first argue that if t_1' and t_2' are connected by a path in R then they are connected in $G - D$. Each edge on this path is either an original edge or a $\Lambda(\gamma)$-edge for some unaffected γ. All original edges not incident to D are still valid and each $\Lambda(\gamma)$ edge can, when γ is unaffected, be replaced by a path in $G - D$ using intermediate nodes in $V(\gamma)$.

We now argue that if P is a u'–v' path in $G - D$,

$$P = (u' = u_0, u_1, \ldots, u_{|P|} = v'),$$

then there exists a t_1'–t_2' path in R. Partition $P = P_1 P_2 \ldots P_\omega$ into maximal subpaths ($P_i = (u_{a(i)}, \ldots, u_{b(i)})$) such that $V(P_i)$ is either

(i) contained in a single affected subtree, or

(ii) contained in $V(\gamma)$ for some unaffected $\gamma \in \mathcal{C}$.

Observe that because of the maximality criterion, no two type-(ii) subpaths can be adjacent. Since P_1 and P_ω contain u' and v', they must be type-(i) subpaths. We want to show that all type-(i) subpaths are connected in R by considering how consecutive type-(i) subpaths could be connected by valid edges in H'. (Recall that H' is H after deleting all $\Lambda(\gamma)$ edges for affected $\gamma \in \mathcal{C}$.) There are two cases to consider.

Case 1. Suppose P_i and P_{i+1} are type-(i) subpaths. Then $(u_{b(i)}, u_{a(i+1)})$ is an original edge in H', so it or some other edge will be discovered that puts the affected subtrees of P_i and P_{i+1} in the same connected component in R.

Case 2. Suppose P_i and P_{i+2} are type-(i) subpaths, but P_{i+1} is a type-(ii) subpath. Let $\gamma \in \mathcal{C}$ be the component for which $V(P_{i+1}) \subset V(\gamma)$, so $u_{b(i)}, u_{a(i+2)} \notin V(\gamma)$. It must be that $u_{b(i)}, u_{a(i+2)} \in A(\gamma)$, and since $\Lambda(\gamma)$ remains connected after any d vertex deletions, $u_{b(i)}$ and $u_{a(i+2)}$ are connected by a path in $\Lambda(\gamma) - D$. All the $\Lambda(\gamma) - D$ edges straddling two affected subtrees are eligible to be discovered when populating the edge-set of R, so the affected subtrees of P_i and P_{i+2} must be in the same connected component in R.

6 Improving the Update Time

In this section we present not one, not two, but three different methods to reduce the update time from $\tilde{O}(d^2 d_\star^2)$ to $\tilde{O}(d^3)$. Each of the three methods uses a different, more sophisticated orthogonal

range searching structure. In Section 6.1 we show how $\tilde{O}(d^3)$ time can be achieved with a 2D *colored* (aka *categorical*) range searching structure [58]. Section 6.2 uses a 2D range *counting* [22] data structure, and Section 6.3 uses a 3D range *emptiness* data structure [20]. The method of Section 6.3 was suggested to us by Shiri Chechik.

6.1 Method 1: Colored Range Searching

We use the following theorem from Larsen and van Walderveen [58].

THEOREM 6.1. *([58]) Given a multiset $P \subset [U] \times [U]$ of n points and coloring $\phi : P \to \mathbb{N}$, there is a data structure occupying space $O(n \log n)$ that answers the following type of query. Given x, x', y, y', report the color set $\Phi = \{\phi(p) \mid p \in P \cap [x, y] \times [x', y']\}$. The query time is $O(\log \log U + |\Phi|)$.*

Assign each component $\gamma \in \mathcal{C}$ a distinct color $\phi(\gamma) \in \{1, \ldots, |\mathcal{C}|\}$. Recall that each edge in H is tagged with its provenance. All *original* and *tree* edges receive color zero and all $\Lambda(\gamma)$ edges receive color $\phi(\gamma)$. Each 2D range query now returns a list of colors in the query rectangle. We halt the search the moment it returns color 0 (an original or tree edge), or the color of any unaffected component. Since there are at most $d(p+1)$ affected components, each of the $O((pd)^2)$ 2D range queries is halted after time $O(\log \log n + pd)$.

Using Method 1 the space of our $d_\star$-failure connectivity oracle becomes $O(d_\star m \log^2 n)$ and the update time becomes $O((pd)^3) = O(d^3 \log^3 n)$.

6.2 Method 2: 2D Range Counting

We use the following theorem of JaJa, Mortensen, and Shi [52].

THEOREM 6.2. *([52]) Given a multiset $P \subset [U] \times [U]$ of n points there is an $O(n)$-space data structure answering the following type of query in $O(\log n / \log \log n)$ time. Given x, x', y, y', report the number $k = |P \cap [x, y] \times [x', y']|$.*

Consider an affected component γ_i and recall that its adjacency list $A(\gamma_i)$ is the concatenation of $A(\gamma_i, \gamma_{i+1}), \ldots, A(\gamma_i, \gamma_p)$, where $\gamma_{i+1}, \ldots, \gamma_p$ are its ancestors in $\mathcal{C}$. The 2D range queries that are influenced by $\Lambda(\gamma_i)$ involve two trees, say $\tau = \tau(\gamma_j)$ and $\tau' = \tau(\gamma_{j'})$ where $i < j \leq j' \leq p$. Each query is the product $Q = I \times I'$ of an interval $I \subset \text{Euler}(\tau)$ and another $I' \subset \text{Euler}(\tau')$. Given the indices of the first and last elements of $A(\gamma_i, \gamma_j) \cap I$ and $A(\gamma_i, \gamma_{j'}) \cap I'$, we can determine in $O(1)$ time

how many $\Lambda(\gamma_i)$ edges (points) appear in Q. Call these *affected points*. For each affected component γ and each query Q to be performed by the update algorithm, we calculate the number of affected $\Lambda(\gamma)$ points in Q. This takes time $O(pd \cdot (pd)^2) = O(d^3 \log^3 n)$.

Let k_Q be the *total* number of affected points in Q, over all affected γ. In $O(\log n / \log \log n)$ time we compute the number k of points in Q. If $k = k_Q$ then there are no unaffected points in Q, and if $k > k_Q$ we deduce that there is an unaffected point (an valid edge reconnecting two affected subtrees). The total time for all $O((pd)^2)$ queries is therefore $O(d^2 \log^3 n / \log \log n)$ time. The bottleneck in this approach is computing the set $\{k_Q\}$ of critical thresholds.

Using Method 2 the space of our $d_\star$-failure connectivity oracle is $O(d_\star m \log n)$ and the update time is $O(d^3 \log^3 n)$.

6.3 Method 3: 3D Range Emptiness

We use the following theorem of Chan, Larsen, and Pătraşcu [20].

THEOREM 6.3. *([20]) Given a set $P \subset [U] \times [U] \times [U]$ of n points there is an $O(n \log^{1+\epsilon} n)$-space data structure answering queries of the following type in $O(\log \log U)$ time. Given x, x', x'', y, y', y'', determine if $P \cap [x, y] \times [x', y'] \times [x'', y''] = \emptyset$.*

List the nodes in $\mathcal{C}$ as $\{\gamma_1, \ldots, \gamma_{|\mathcal{C}|}\}$. Suppose that (u, v) is an original or tree edge in H and t_u, t_v are the trees in $\mathcal{T}$ containing principal copies of u and v, where u appears at position i in $\text{Euler}(t_u)$ and v appears at position j of $\text{Euler}(t_v)$. Rather than map (u, v) to the point (i, j) in the 2D structure of $\mathbf{ET}(H, \mathcal{T})$ we map it to the 3D point $(i, j, 0)$. If (u, v) is an edge of $\Lambda(\gamma_k)$ we map it to the point (i, j, k).

Let $(\gamma_{k_1}, \gamma_{k_2}, \ldots, \gamma_{k_{d(p+1)}})$ be the affected components and Q be a 2D query performed by the update algorithm. We are interested in knowing whether there is a point whose first two coordinates are in Q and whose third coordinate is *not* a member of $\{k_1, \ldots, k_{d(p+1)}\}$. Thus the 2D query Q can be reduced to $d(p + 1)$ 3D emptiness queries $Q \times [0, k_1)$, $Q \times (k_1, k_2)$, and so on. Each 3D query is answered in $O(\log \log n)$ time, so the total update time is $O(d^3 \log^3 n \log \log n)$.

With the current state-of-the-art range searching data structures [20, 22, 52, 58], Method 2 is always strictly superior to Methods 1 and 3 in update time

or space or both. Method 2 also leaves the most room for improvement since the bottleneck is not range counting queries *per se*, but computing the critical thresholds $\{k_Q\}$ for the queries.

7 Monte Carlo Connectivity Oracles

By introducing Monte Carlo randomization we are able to simultaneously reduce both the update time to $\tilde{O}(d^2)$ and the space to $\tilde{O}(m)$, *independent of* $d_\star$. The cost of this adaptation is the possibility of false negatives, i.e., a connectivity query reporting that u and v are disconnected in $G - D$ when they are, in fact, connected. Connectivity queries are answered correctly with probability $1 - 1/\operatorname{poly}(n)$. Our approach is along the same lines as Kapron, King, and Mountjoy [55, 46] and Ahn, Guha, and McGregor [3], but requires new ideas specific to handling vertex failures. Since $\tilde{O}(d^2)$ update time is trivial when $d \geq d_\star = \Omega(\sqrt{m}/\operatorname{poly}(\log n))$, we begin by computing $\mathcal{T}, \mathcal{C}$, and H with $d_\star$ as the maximum number of vertex deletions. The time to construct $\mathcal{T}$ and $\mathcal{C}$ is $O(mn \log n)$, and the time to construct H is linear in its size, $O(d_\star m \log n) = o(m^{3/2}) = o(mn)$.

Without loss of generality, we assume the vertices of $\mathcal{T}$ are assigned IDs from $\{1, \ldots, pn - 1\}$. Let $\langle u \rangle$ be the $\log_2(pn)$-bit encoding of u's ID. The graphs we represent are multigraphs: multiple edges may have the same endpoints but different provenances. Suppose $e = ((u, v), x)$ is an edge connecting u and v with provence tag x and $\langle u \rangle < \langle v \rangle$ (lexicographically). We encode e by the $O(\log n)$-bit string $\langle e \rangle = \langle u, v, x \rangle$. The central observation of [55, 46, 3] is that edges have two endpoints and that $\langle e \rangle \oplus \langle e \rangle = 0$. Here $\oplus$ is bit-wise XOR.

At construction time we generate $O(\log^2 n)$ edge sets $E_{\alpha,\beta}$, $\alpha \in [c \log n], \beta \in [\log |E(H)|]$, where $E_{\alpha,\beta}$ is obtained by sampling each edge in H independently with probability $1/2^\beta$. Let $\Upsilon(u)$ be a $(c \log n) \times (\log |E(H)|)$ *sample matrix* for edges incident to u, where

$$\Upsilon(u)(\alpha, \beta) = \bigoplus_{e=((u,v),x) \in E_{\alpha,\beta}} \langle e \rangle.$$

Our data structure stores *original* and *tree edges* explicitly, but not $\Lambda(\gamma)$-edges. Nonetheless, we are able to verify whether a bit string $\langle u, v, x \rangle$ corresponds to an actual edge in H in $O(1)$ time. If $x = \gamma$, we confirm that $u, v \in A(\gamma)$ and that their positions in this list differ by at most $d_\star + 1$. Lemma 7.1 captures why the sample matrices are useful for finding connecting edges between subgraphs.

LEMMA 7.1. *([3, 55]) Let $S \subset V$ and $\Upsilon(S) = \bigoplus_{u \in S} \Upsilon(u)$ be the component-wise XOR of the sample matrices of vertices in S. If any edges in $E(H)$ cross the cut $(S, V - S)$ then for each α, with constant probability some entry in the row $\Upsilon(S)(\alpha, \star)$ is the encoding of an edge crossing the cut $(S, V - S)$.*

Proof. First observe that any edge with both endpoints in S is counted exactly twice or zero times at each entry $\Upsilon(S)(\alpha, \beta)$. Since $\langle e \rangle \oplus \langle e \rangle = 0$, $\Upsilon(S)(\alpha, \beta)$ is the XOR of all edges in $E_{\alpha,\beta}$ crossing the cut $(S, V - S)$. Suppose the number of edges crossing the cut is in the range $(2^{\beta-1}, 2^\beta]$. Then with constant probability, only one cut-edge is sampled for inclusion in $\Upsilon(S)(\alpha, \beta)$. For each (α, β), we can check in $O(1)$ time whether $\Upsilon(S)(\alpha, \beta)$ is the encoding $\langle e \rangle$ of a single edge $e \in E(H)$ crossing the cut $(S, V-S)$, or whether it is random garbage from the XOR of multiple edge encodings.

Our goal is to use Lemma 7.1 to find edges that reconnect the affected subtrees. However, we need to ignore or subtract off (XOR) three types of edges: those connecting affected subtrees to some unaffected tree, those edges incident to D, and those edges in $\Lambda(\gamma)$ for some affected γ. To this end we store the following interval-sum data structures. Each can be implemented with standard binary trees. Since all "logarithmic time" operations on these trees are actually operating on sample matrices, which take $O(\log^2 n)$ space, the query time for each of these structures is $O(\log^3 n)$.

$\mathcal{S}[\tau, \tau']$. Both τ, τ' are trees in $\mathcal{T}$, and it may be that $\tau = \tau'$. For any vertex $u \in V(\tau)$, let $\Upsilon^{\tau,\tau'}(u)$ be the sample matrix for $E(H) \cap (\{u\} \times V(\tau'))$. Given an interval I of Euler(τ), $\mathcal{S}[\tau, \tau']$ reports $\bigoplus_{u \in I} \Upsilon^{\tau,\tau'}(u)$.

$\mathcal{S}[v, \tau]$. Here v is the principal copy of v in some $\tau' \in \mathcal{T}$, where τ and τ' could be identical. Given any interval I of Euler(τ), $\mathcal{S}[v, \tau]$ reports the sample matrix $\Upsilon^{\tau,I}(v)$ for the edge set $E(H) \cap (\{v\} \times I)$.

$\mathcal{S}[\gamma, \tau]$. Here $\tau \in \mathcal{T}$ and $\gamma \in \mathcal{C}$. Let $\Upsilon^\gamma(u)$ be the sample matrix for $\Lambda(\gamma) \cap (\{u\} \times V(H))$, that is, all $\Lambda(\gamma)$-edges incident to u. Given an interval I of Euler(τ), $\mathcal{S}[\gamma, \tau]$ reports the sample matrix $\bigoplus_{u \in I} \Upsilon^\gamma(u)$.

Space Analysis. We only need to store $\mathcal{S}[\tau, \tau']$ if the edge-set could be non-empty, i.e., if $\tau = \tau(\gamma)$ and $\tau' = \tau(\gamma')$ for some $\gamma, \gamma' \in \mathcal{C}$ with $\gamma \preceq \gamma'$ or $\gamma' \prec \gamma$. The space is $O(\log^2 n)$ times the number of $V(\tau)$-vertices with non-zero degree. Each original edge contributes to at most one $\mathcal{S}[\tau, \tau']$ structure, so these are easy to bound. Bounding the contribution of $\Lambda(\gamma)$-type edges is less straightforward. Let Γ be the set of strict ancestors of γ in $\mathcal{C}$ and $\tau(\Gamma)$ be the corresponding trees in $\mathcal{T}$. Clearly $|\Gamma| \leq p$. The list $A(\gamma)$ includes principal vertices in potentially all the trees in $\tau(\Gamma)$. Observe that for sufficiently large $d_\star$, $\Lambda(\gamma)$ becomes a clique on $A(\gamma)$. Thus, for any $\tau, \tau' \in \tau(\Gamma)$, potentially all $A(\gamma)$-vertices in $V(\tau)$ will have non-zero degree and contribute $O(\log^2 n)$ space to $\mathcal{S}[\tau, \tau']$. There are at most p choices for $\tau' \in \tau(\Gamma)$. Thus, each $\gamma \in \mathcal{C}$ contributes $O(|A(\gamma)| \cdot p \log^2 n)$ space to all $\mathcal{S}[\tau, \tau']$ structures, and since $\sum_\gamma |A(\gamma)| \leq pm$, the total space is $O(mp^2 \log^2 n) = O(m \log^4 n)$. By a similar analysis, the space for $\mathcal{S}[v, \tau]$ and $\mathcal{S}[\gamma, \tau]$ are also upper bounded by $O(m \log^4 n)$.

Processing Vertex Failures. Step 1 of the deletion algorithm is the same: we mark at most $d(p+1)$ components $\gamma \in \mathcal{C}$ *affected*, and mark tree edges incident to D-vertices in $\tau(\gamma) \in \mathcal{T}$ as deleted. If f such edges are deleted, it breaks $\text{Euler}(\tau(\gamma))$ into $2f+1$ intervals such that each affected subtree in $\tau(\gamma)$ is made up of the union of a subset of the intervals.

Let $\{t_1, \ldots, t_{O(dp)}\}$ be the initial set of affected subtrees. In order to implement Step 2 of the deletion algorithm we first obtain a sample matrix $\Upsilon(t_i)$ for each t_i that includes H'-edges incident to $V(t_i)$, that is, it excludes deleted edges in $V(t_i) \times D$ and excludes artificial edges in $\Lambda(\gamma)$ for an affected γ. If t_i was originally in $\tau \in \mathcal{T}$ we start by asking $\mathcal{S}[\tau, \tau]$ for a sample matrix $\Upsilon^{\tau, \tau}(t_i)$ covering edges in the subgraph induced by $V(\tau)$. For each affected $\tau' \neq \tau$, we ask $\mathcal{S}[\tau, \tau']$ for a sample matrix $\Upsilon^{\tau, \tau'}(t_i)$ covering edges in $V(\tau) \times V(\tau')$. For each $x \in D$ we ask $\mathcal{S}[x, \tau]$ for a sample matrix $\Upsilon^{x, \tau}(t_i)$ covering the edge set $\{x\} \times V(t_i)$. Finally, for each affected $\gamma \in \mathcal{C}$, we ask $\mathcal{S}[\gamma, \tau]$ for $\Upsilon^\gamma(t_i)$ covering $\Lambda(\gamma)$. The matrix $\Upsilon(t_i)$ we want is now computed as

$$\Upsilon(t_i) \leftarrow \bigoplus_{\tau'} \Upsilon^{\tau, \tau'}(t_i) \oplus \bigoplus_{x \in D} \Upsilon^{x, \tau}(t_i) \oplus \bigoplus_\gamma \Upsilon^\gamma(t_i),$$

where the variables τ' and γ range over all affected trees in $\mathcal{T}$ and affected components in $\mathcal{C}$. The time to construct one $\Upsilon(t_i)$ is $O(dp \cdot \log^3 n)$, and total time to construct all $\{\Upsilon(t_i)\}_i$ is $O((dp)^2 \log^3 n) =$

$O(d^2 \log^5 n)$.

The second stage of the deletion algorithm is to use the $\{\Upsilon(t_i)\}$ structures to determine which affected subtrees are in the same connected component. We proceed as in Borůvka's MST algorithm [16] and many parallel connectivity algorithms that use the "hook and contract" technique [25, 26, 53]. In each round, each affected subtree will pick an arbitrary edge joining it to a different affected subtree. The affected subtrees will be merged into larger affected subtrees, which participate in the next round. Under error-free conditions—which we do not have—this process will halt after $\log_2(O(dp))$ rounds since each round reduces the number of non-isolated affected subtrees by at least half.

The formal procedure is as follows. Let $C_k = \{t_{k,1}, t_{k,2}, \ldots, t_{k,|C_k|}\}$ be the affected trees after k rounds, where $C_0 = \{t_{0,1}, \ldots, t_{0,O(dp)}\}$. We maintain the invariant that after k rounds we have, for each $t_{k,l}$, a sample matrix of $\Upsilon(t_{k,l})$ of all H' edges incident to $t_{k,l}$. In the $(k+1)$th round, loop over each $t_{k,l} \in C_k$ and scan the entries in the $(k+1)$th row of its sample matrix, $\Upsilon(t_{k,l})(k+1, \star)$, looking for the encoding of an edge $e_{k,l}$ with exactly one endpoint in $V(t_{k,l})$. Let C_{k+1} be the components induced by the C_k trees and the inter-tree edges $\{e_{k,l}\}$ just selected. Suppose the constituent trees of $t_{k+1,l'} \in C_{k+1}$ are $S \subseteq C_k$. The sample matrix for $t_{k+1,l'}$ is easily computed as $\bigoplus_{t \in S} \Upsilon(t)$. The total time to compute C_{k+1} sample matrices is just $O((|C_k| - |C_{k+1}|) \log^2 n)$.

Observe that just before executing the $(k+1)$th round we have only examined the contents of rows 1 through k of the sample matrices. Thus, the contents of the $(k+1)$th row are not affected by the measurements performed so far and we can apply Lemma 7.1. Lemma 7.1 implies that if $t_{k,l}$ is not already isolated, then with constant probability we will find an edge with one endpoint in $V(t_{k,l})$ in $\Upsilon(t_{k,l})(k+1, \star)$. Letting $\|C_k\|$ be the number of non-isolated components in C_k, we have $\mathbb{E}[\|C_{k+1}\|] \leq (1-\epsilon)\|C_k\|$ for some absolute constant $\epsilon > 0$. Thus, after $c \log n$ rounds $\mathbb{E}[\|C_{c \log n}\|] \leq (1-\epsilon)^{c \log n}|C_0| < n^{-\Omega(c)}$ and by Markov's inequality, the probability that $C_{c \log n}$ has non-isolated components (an error) is $n^{-\Omega(c)}$.

THEOREM 7.1. *In $O(mn \log n)$ time we can construct an $O(m \log^4 n)$-space data structure supporting the following operations. Given a set $D \subset V$ with $|D| = d$, we can process D in $O(d^2 \log^5 n)$ time and thereafter answer pairwise connectivity queries*

in the graph $G - D$, in $O(d)$ time.

REMARK 1. *The time to delete D can be reduced to $O(d^2 \log(d \log n) \log^4 n)$ with probability $1 - 1/\operatorname{poly}(d \log n)$. Observe that there are $O(dp) = O(d \log n)$ affected subtrees, so $O(\log(dp))$ rounds of hook and contract suffice to coalesce the affected subtrees into their connected components in $G - D$. Thus, when forming sample matrices we can focus exclusively on the first $O(\log(dp))$ rows: all $\log^2 n$ factors in the running time are replaced by $\log n \log(dp)$. If the first $O(\log(dp))$ rows do not suffice we can always generate the remaining $O(\log n)$ rows afterward to obtain a negligible error bound.*

Kapron et al. [55] stated that their d-edge failure oracle has update time $O(d \log^3 n \log d)$. Our analysis above, together with the observation that it is not necessary to explicitly store the graph [3, 46] gives slightly improved bounds for d-edge failure connectivity oracles.

THEOREM 7.2. *([55, 3, 46]) Given an undirected graph $G = (V, E)$, a data structure with size $O(n \log^2 n)$ can be constructed in $\tilde{O}(m)$ time supporting the following operations. Any set $D \subset E$ of d edges can be processed in $O(d \log^2 n \log d)$ time with probability $1 - 1/\operatorname{poly}(d)$ (or $O(d \log^3 n)$ time w.h.p.) such that thereafter, connectivity queries in the graph $(V, E - D)$ can be answered in $O(\min\{\log \log n, \log d / \log \log n\})$ time. Each connectivity query is answered correctly with probability $1 - 1/\operatorname{poly}(n)$.*

8 Conclusions

In this paper we illustrated the power of a new graph decomposition theorem by giving time- and space-efficient connectivity oracles for graphs subject to vertex failures. Our data structures perform well in all the major measures of efficiency (space, update time, query time, and preprocessing time) but leave many opportunities for improvement. We highlight the most interesting open problems.

- The Fürer-Raghavachari [43] algorithm FR-Tree for computing near-minimum degree spanning tree takes $O(mn \log n)$ time, which is the sole bottleneck in our construction algorithm. Is it possible to reduce the running time of FR-Tree to $\tilde{O}(m)$, or compute spanning trees with similar decomposition properties in $\tilde{O}(m)$ time?

- The conditional lower bounds of [57, 48] show that any connectivity oracle with reasonable

update time cannot have $\tilde{O}(1)$ query time, independent of d, but they do not preclude a data structure having both query and update time $\tilde{O}(d)$. Is it possible to reduce the update time below $O(d^2)$ without disturbing the space or query time?

- Is it possible to reduce the space of our deterministic connectivity oracle to $\tilde{O}(m)$, independent of $d_\star$?

Acknowledgement. We would like to thank Kasper Green Larsen and Peyman Afshani for help with the navigating the range searching literature, Shiri Chechik for suggesting the reduction to 3D range searching in Section 6.3, and Veronika Loitzenbauer for bringing [49] to our attention and suggesting several improvements to the presentation.

References

[1] I. Abraham, S. Chechik, and C. Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings 44th ACM Symposium on Theory of Computing (STOC)*, pages 1199–1218, 2012.

[2] I. Abraham, S. Chechik, C. Gavoille, and D. Peleg. Forbidden-set distance labels for graphs of bounded doubling dimension. *ACM Trans. on Algorithms*, 12(2):22, 2016.

[3] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 459–467, 2012.

[4] S. Arora, S. Rao, and U. V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2), 2009.

[5] I. Baran, E. D. Demaine, and M. Pătraşcu. Subquadratic algorithms for 3SUM. *Algorithmica*, 50(4):584–596, 2008.

[6] D. Barnette. Trees in polyhedral graphs. *Canadian Journal of Mathematics*, 18:731–736, 1966.

[7] S. Basawna, U. Lath, and A. S. Mehta. Single source distance oracle for planar digraphs avoiding any failed node or link. In *Proceedings 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 223–232, 2012.

[8] S. Baswana, S. R. Chaudhury, K. Choudhary, and S. Khan. Dynamic DFS in undirected graphs: breaking the $o(m)$ barrier. In *Proceedings 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 730–739, 2016.

[9] S. Baswana, K. Choudhary, and L. Roditty. Fault tolerant subgraph for single source reachability: generic and optimal. In *Proceedings of the 48th Annual ACM Symposium on Theory of Computing (STOC)*, pages 509–518, 2016.

[10] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 15:302–318, 1996.

[11] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings 4th Latin American Symp. on Theoretical Informatics (LATIN), LNCS Vol. 1776*, pages 88–94, 2000.

[12] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[13] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 101–110, 2009.

[14] D. Bilò, F. Grandoni, L. Gualà, S. Leucci, and G. Proietti. Improved purely additive fault-tolerant spanners. In *Proceedings 23rd Annual European Symposium on Algorithms (ESA)*, pages 167–178, 2015.

[15] G. Borradaile, S. Pettie, and C. Wulff-Nilsen. Connectivity oracles for planar graphs. In *Proceedings 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 316–327, 2012. Full version available at arXiv:1204.4159.

[16] O. Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3:37–58, 1926. In Czech.

[17] G. Braunschvig, S. Chechik, and D. Peleg. Fault tolerant additive spanners. In *Proceedings 38th Int'l Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 206–214, 2012.

[18] G. Braunschvig, S. Chechik, D. Peleg, and A. Sealfon. Fault tolerant additive and (μ, α)-spanners. *Theoretical Computer Science*, 580:94–100, 2015.

[19] T. Chan. Dynamic subgraph connectivity with geometric applications. *SIAM J. Comput.*, 36(3):681–694, 2006.

[20] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings 27th ACM Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[21] T. M. Chan, M. Pătraşcu, and L. Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM J. Comput.*, 40(2):333–349, 2011.

[22] T. M. Chan and B. T. Wilkinson. Adaptive and approximate orthogonal range counting. In *Proceedings 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 241–251, 2013.

[23] S. Chechik, M. Langberg, D. Peleg, and L. Roditty. Fault tolerant spanners for general graphs. *SIAM*

[24] S. Chechik, M. Langberg, D. Peleg, and L. Roditty. *f*-sensitivity distance oracles and routing schemes. *Algorithmica*, 63(4):861–882, 2012.

[25] K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48(2):297–323, 2001.

[26] K. W. Chong and T. W. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. *J. Algor.*, 18(3):378–402, 1995.

[27] K. Choudhary. An optimal dual fault tolerant reachability oracle. In *Proceedings 43rd Int'l Colloq. on Automata, Languages, and Programming (ICALP)*, 2016.

[28] A. Czumaj and W.-B. Strothmann. Bounded degree spanning trees. In *Proceedings 5th Annual European Symposium on Algorithms (ESA)*, pages 104–117, 1997.

[29] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008.

[30] E. A. Dinic, A. V. Karzanov, and M. V. Lomonosov. On the structure of the system of minimum edge cuts in a graph. *Studies in Discrete Optimization*, pages 290–306, 1976. (in Russian).

[31] M. Dinitz and R. Krauthgamer. Fault-tolerant spanners: better and simpler. In *Proceedings 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 169–178, 2011.

[32] Y. Dinitz and Z. Nutov. A 2-level cactus model for the system of minimum and minimum+1 edge-cuts in a graph and its incremental maintenance. In *Proceedings 27th ACM Symposium on Theory of Computing (STOC)*, pages 509–518, 1995.

[33] Y. Dinitz and Z. Nutov. A 2-level cactus tree model for the system of minimum and minimum+1 edge cuts of a graph and its incremental maintenance. Part I: the odd case. Unpublished manuscript, 1999.

[34] Y. Dinitz and Z. Nutov. A 2-level cactus tree model for the system of minimum and minimum+1 edge cuts of a graph and its incremental maintenance. Part II: the even case. Unpublished manuscript, 1999.

[35] Y. Dinitz and J. Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica*, 20(3):242–276, 1998.

[36] R. Duan. New data structures for subgraph connectivity. In *Proceedings 37th Int'l Colloquium on Automata, Languages and Programming (ICALP)*, pages 201–212, 2010.

[37] R. Duan. *Algorithms and Dynamic Data Structures for Basic Graph Optimization Problems*. Ph.D. Dissertation, University of Michigan, 2011.

J. Comput., 39(7):3403–3423, 2010.

[38] R. Duan and S. Pettie. Dual-failure distance and connectivity oracles. In *Proceedings 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 506–515, 2009.

[39] R. Duan and S. Pettie. Connectivity oracles for failure prone graphs. In *Proceedings 42nd ACM Symposium on Theory of Computing*, pages 465–474, 2010.

[40] D. Eppstein, Z. Galil, G. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[41] G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[42] D Frigioni and G. F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000.

[43] M. Fürer and B. Raghavachari. Approximating the minimum-degree steiner tree to within one of optimal. *J. Algor.*, 17(3):409–423, 1994.

[44] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.

[45] Z. Galil and G. Italiano. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput.*, 22(1):11–28, 1993.

[46] D. Gibb, B. M. Kapron, V. King, and N. Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

[47] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings 55th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 621–630, 2014.

[48] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings 47th Annual ACM Symposium on Theory of Computing (STOC)*, pages 21–30, 2015.

[49] M. Henzinger and S. Neumann. Incremental and fully dynamic subgraph connectivity for emergency planning. In *Proceedings 24th Annual European Symposium on Algorithms (ESA)*, pages 48:1-48:11, 2016.

[50] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[51] S.-E. Huang, D. Huang, T. Kopelowitz, and S. Pettie. Fully Dynamic Connectivity in $O(\log n(\log \log n)^2)$ Amortized Expected Time. In *Proceedings 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2017.

[52] J. JaJa, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proceedings 15th Int'l Symposium on Algorithms and Computation (ISAAC)*, pages 558–568, 2004.

[53] D. B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. *J. Comput. Syst. Sci.*, 54(2):227–242, 1997.

[54] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. On-line maintenance of the four-connected components of a graph. In *Proceedings 32nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 793–801, 1991.

[55] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1131–1142, 2013.

[56] C. Kejlberg-Rasmussen, T. Kopelowitz, S. Pettie, and M. Thorup. Faster worst case deterministic dynamic connectivity. In *Proceedings 24th European Symposium on Algorithms (ESA)*, pages 53:1–53:15, 2016.

[57] T. Kopelowitz, S. Pettie, and E. Porat. Higher lower bounds from the 3SUM conjecture. In *Proceedings 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1272–1287, 2016.

[58] K. G. Larsen and F. van Walderveen. Near-optimal range reporting structures for categorical data. In *Proceedings 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 265–276, 2013.

[59] K. G. Larsen and R. Williams. Faster online matrix-vector multiplication. In *Proceedings 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2017.

[60] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.

[61] M. Parter. Fault-tolerant logical network structures. *Bulletin of the EATCS*, 118, 2016.

[62] M. Parter and D. Peleg. Sparse fault-tolerant BFS trees. In *Proceedings 21st Annual European Symposium on Algorithms (ESA)*, pages 779–790, 2013.

[63] M. Parter and D. Peleg. Fault tolerant approximate BFS structures. In *Proceedings 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1073–1092, 2014.

[64] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[65] M. Pătraşcu and M. Thorup. Planning for fast connectivity updates. In *Proceedings 48th IEEE*

Symposium on Foundations of Computer Science (FOCS), pages 263–271, 2007.

[66] M. Patrascu and M. Thorup. Randomization does not help searching predecessors. In *Proceedings 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 555–564, 2007.

[67] M. Patrascu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proceedings 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 166–175, 2014.

[68] J. A. La Poutré, J. van Leeuwen, and M. H. Overmars. Maintenance of 2- and 3-edge- connected components of graphs I. *Discrete Mathematics*, 114(1–3):329–359, 1993.

[69] W.-B. Strothmann. *Bounded Degree Spanning Trees*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Theoretische Informatik, 1997.

[70] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components online. *Algorithmica*, 7(5&6):433–464, 1992.

[71] C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1757–1769, 2013.