Connectivity Oracles for Failure Prone Graphs*

Ran Duan
University of Michigan
duanran@umich.edu

Seth Pettie
University of Michigan
pettie@umich.edu

ABSTRACT

Dynamic graph connectivity algorithms have been studied for many years, but typically in the most general possible setting, where the graph can evolve in completely arbitrary ways. In this paper we consider a *dynamic subgraph* model. We assume there is some fixed, underlying graph that can be preprocessed ahead of time. The graph is subject only to vertices and edges flipping "off" (failing) and "on" (recovering), where queries naturally apply to the subgraph on edges/vertices currently flipped on. This model fits most real world scenarios, where the topology of the graph in question (say a router network or road network) is constantly evolving due to temporary failures but never deviates too far from the ideal failure-free state.

We present the first efficient *connectivity oracle* for graphs susceptible to vertex failures. Given vertices u and v and a set D of d failed vertices, we can determine if there is a path from u to v avoiding D in time polynomial in $d \log n$. There is a tradeoff in our oracle between the space, which is roughly mn^{ϵ}, for $0 < \epsilon \leq 1$, and the polynomial query time, which depends on ϵ. If one wanted to achieve the same functionality with existing data structures (based on edge failures or twin vertex failures) the resulting connectivity oracle would either need exorbitant space ($\Omega(n^d)$) or update time $\Omega(dn)$, that is, linear in the number of vertices. Our connectivity oracle is therefore the first of its kind.

As a byproduct of our oracle for vertex failures we reduce the problem of constructing an *edge*-failure oracle to 2D range searching over the integers. We show there is an $\tilde{O}(m)$-space oracle that processes any set of d failed edges in $O(d^2 \log\log n)$ time and, thereafter, answers connectivity queries in $O(\log\log n)$ time. Our update time is exponentially faster than a recent connectivity oracle of Pătraşcu and Thorup for bounded d, but slower as a function of d.

*This work is supported by NSF CAREER grant no. CCF-0746673 and a grant from the US-Israel Binational Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
STOC'10, June 5–8, 2010, Cambridge, Massachusetts, USA
Copyright 2010 ACM 978-1-4503-0050-6/10/06 ...$10.00.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms

General Terms

Algorithms, Theory

Keywords

Connectivity

1. INTRODUCTION

Real world graphs and networks are vulnerable objects subject to unexpected node and link failures, fluctuations in congestion, and occasional augmentations of nodes and links. In the midst of such dynamism we would like to maintain as much information about the *current* network as possible and be able to perform basic tasks, for example, finding the (approximate) distance between two vertices, determining whether two vertices are simply connected (or k-vertex/edge connected) and to produce a shortest or nearly shortest path between two vertices if they are connected. Dynamic connectivity and shortest path problems have been studied since the late 1960s. However, decades of research on the subject focused on what is arguably an *un*naturally general model of a dynamic graph, namely, one subject to *completely arbitrary* insertions and deletions of nodes and links. In this paper we start from the assumption that there is some fixed underlying graph and that updates consist solely of flipping vertices and edges *on* or *off*, possibly with a restriction on the number of vertices flipped off at any time. In contrast to the general model of dynamic graphs, we will refer to this as the *dynamic subgraph* model. This model gives one the freedom to preprocess the underlying graph in order to facilitate more efficient updates and queries.

Our main result is a new, space efficient data structure that can quickly answer connectivity queries after recovering from d *vertex failures.* The recovery time is polynomial in d and $\log n$ but otherwise independent of the size of the graph. After processing the failed vertices, connectivity queries are answered in $O(d)$ time. The space used by the data structure is roughly mn^{ϵ}, for any fixed $\epsilon > 0$, where ϵ only affects the polynomial in the recovery time. The exact tradeoffs are given in Theorem 1.1. Our data structure is the first of its type. To achieve comparable query times using existing data structures we would need either $\Omega(n^d)$ space [11] or $\Omega(dn)$ recovery time [17].

Technical Challenges.

The obvious distinction between vertex failures and edge failures is proportionality: just a handful of d vertex failures can have huge impact on the graph connectivity that is completely divorced from d, whereas d edge failures have an effect proportional to d. In the happiest scenario the graph has constant degree and we can simply simulate vertex failures (with $O(1)$ slowdown) using Pătraşcu and Thorup's data structure [17] that processes a batch of edge deletions in $\tilde{O}(1)$ time per edge and answers connectivity queries in $\tilde{O}(1)$ time. Given this trivial observation, a natural (but doomed) way to attack the vertex-failure problem is to find a *sparse surrogate* of the graph that behaves like the input graph (in terms of connectivity) when there are at most d vertex failures, and that has low degree, say polynomial in d. It is easy to see that such sparse surrogates do not exist in general. However, in designing our data structure we *do* achieve something comparable, the details of which we will attempt to sketch.

The first component of our overall data structure is one that, given a spanning tree T, will recover from d vertex failures in time polynomial in the degrees of the failed vertices *in* T, but independent of their degrees in the overall graph. In other words, the *relevant degree* of a vertex can be much smaller than its actual degree and all but the relevant edges incident to a vertex can be deleted implicitly. (Our structure is essentially a reduction from a special subgraph connectivity problem to a 2D range reporting problem.)

The second component of our data structure is a redundant representation of the graph, parameterized by d, the maximum number of failures, and a degree threshold s, with the following properties: (1) there are roughly $n^{\log_{s/d} d}$ different representations of the graph such that (2) for any set of at most d vertex failures, in at least *one* of the graph representations, the *relevant* degree of each failed vertex is at most s, and (3) given the failed vertices, the correct graph representation can be selected in time polynomial in d. In other words, the redundant graph representation effectively sparsifies the graph, assuming that we only want to model the effect of d vertex failures. The space for each graph representation in (1) is roughly linear. If we select $s = d^{1+c}$, for $c \geq 1$, then the overall space for the data structure is roughly $mn^{1/c}$ and the time to recover from vertex failures is polynomial in d and s, i.e., polynomial in d. Notice that increasing c drives the space arbitrarily close to linear and increases the recovery time. Theorem 1.1 gives a precise statement of the capabilities and time-space tradeoffs of our structure:

THEOREM 1.1. *Let $G = (V, E)$ be a graph with m edges and n vertices and let $c \geq 1$ be an integer. A data structure with size $S = O(d^{1-2/c} mn^{1/c - 1/(c \log(2d))} \log^2 n)$ can be constructed in $\tilde{O}(S)$ time that supports the following operations. Given a set D of at most d failed vertices, D can be processed in $O(d^{2c+4} \log^2 n \log\log n)$ time so that connectivity queries w.r.t. the graph induced by $V \backslash D$ can be answered in $O(d)$ time.*

To guarantee that the processing time is polynomial in $|D|$ rather than the upper bound d we simply build a version of the structure for $d = 2, 4, 8, \ldots$. Note that for $c \in \{1, 2\}$ the size of the structures is stable or geometrically decreasing with d and the overall space[1] is $\tilde{O}(mn^{1/c})$.

Prior Work on Subgraph Connectivity.

It is rather astonishing that dynamic *subgraph* connectivity problems—in which vertices and edges are flipped on and off—have been largely ignored until very recently [6, 7, 9, 11, 10, 5, 17, 13]. The premises of this graph model were laid out by Frigioni and Italiano [13] in 2000, who showed that in undirected planar graphs, flipping vertices and answering connectivity queries could all be performed in $O(\log^3 n)$ amortized time. Chan, Pătraşcu, and Roditty [6, 7] considered the same problem on general undirected graphs and showed that vertex flips can be supported in $\tilde{O}(m^{2/3})$ amortized time while answering connectivity queries in $\tilde{O}(m^{1/3})$ time. Pătraşcu and Thorup [17] gave a data structure that could process any d edge deletions in $O(d \log^2 n \log\log n)$ time and thereafter answer connectivity queries in $O(\log\log n)$ time. One downside of this data structure is its *exponential* construction time, due to the need to compute sparsest cuts. If one uses $O(\sqrt{\log n})$-approximate sparsest cuts [3, 2, 20], their construction time becomes polynomial and the deletion time becomes $O(d \log^{2.5} n \log\log n)$.

Demetrescu et al. [9] (see also [5]) considered the problem of answering *distance* queries in directed graphs in the absence of precisely one vertex or edge failure. They gave an $\tilde{O}(n^2)$-space structure that answered such queries in $O(1)$ time. This was later improved by Duan and Pettie [11], who showed that distance queries avoiding two edge/vertex failures could be answered in $O(\log n)$ time with a $\tilde{O}(n^2)$-space structure. No non-trivial results are known for three or more failures. For a survey of recent *fully* dynamic graph algorithms (i.e., not dynamic subgraph algorithms), refer to [15, 18, 22, 19, 8, 21].

Overview.

In Section 2 we present the *Euler tour structure*, which plays a key role in our vertex-failure oracle and can be used independently as an edge-failure oracle. In Sections 3 and 4 we define and analyze the redundant graph representation (called the *high degree hierarchy*) mentioned earlier. In Section 5 we provide algorithms to recover from vertex failures and answer connectivity queries.

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 [17] in nearly every respect.[2] The strength of the

[1]Also note that for moderate values of d the total space is $o(mn^{1/c})$, for any c. Specifically, for $d \ll 2^{\sqrt{(\log n)/c}}$ the factor $d^{1-2/c} n^{-1/(c \log(2d))} \log^2 n$ is $o(1)$. It is never worthwhile making $d > m^{1/(2c+4)}$ since we can trivially answer connectivity queries in $O(1)$ time after $O(m)$ preprocessing, so the maximum space is $\tilde{O}(m^{1+\frac{c-2}{c(2c+4)}} n^{1/c})$, which is always $o(mn^{2/c})$ since $m < n^2$.

[2]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: $O(m \log^{\epsilon} n)$ vs. $O(m)$. It handles d edge deletions exponentially

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 $\mathcal{F} = \{T_1, \ldots, T_t\}$ be a set of vertex disjoint trees in G. (Each T_i does not necessarily span a connected component of G.) There is a data structure $\mathbf{ET}(G, \mathcal{F})$ occupying space $O(m \log^\epsilon n)$ (for any fixed $\epsilon > 0$) 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 $O(d^2 \log\log n + d')$ time we can report which pairs of trees in $\mathcal{F}'$ are connected by an edge in $E \backslash D$. In $O(\min\{\log\log n, \log d\})$ time we can determine which tree in $\mathcal{F}'$ contains a given vertex.*

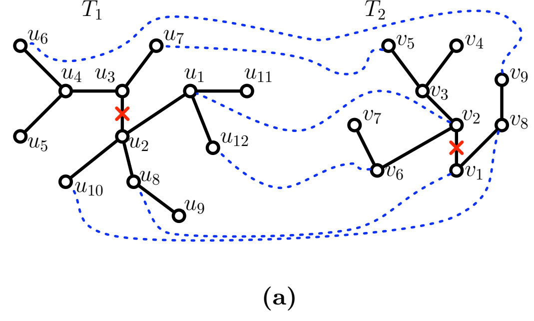
Our data structure uses as a subroutine Alstrup et al.'s data structure [1] for range reporting on the integer grid $[U] \times [U]$. They showed that given a set of N points, there is a data structure with size $O(N \log^\epsilon N)$, where $\epsilon > 0$ is fixed, such that given $x, y, w, z \in [U]$, the set of k points in $[x, y] \times [w, z]$ can be reported in $O(\log\log U + k)$ time. Moreover, the structure can be built in $O(N \log N)$ time.

For a tree T, let $L(T)$ 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 $L(T)$ 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 $L(T_1) = (u_1, \ldots, u_{|T_1|})$, $L(T_2) = (v_1, \ldots, v_{|T_2|})$, and let $U = \max\{|T_1|, |T_2|\}$. We define the point set $P \subseteq [U] \times [U]$ to be $P = \{(i, j) \mid (u_i, v_j) \in E\}$.[3] 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 $L(T_1)$ into a set $I_1 = \{[x_i, y_i]\}_i$ of $2d_1 + 1$ intervals and $L(T_2)$ 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 $E \backslash D$. 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 $O(d_1 d_2 \log\log U + |D|) = O(d_1 d_2 \log\log n + d')$. See Figure 1 for an illustration.

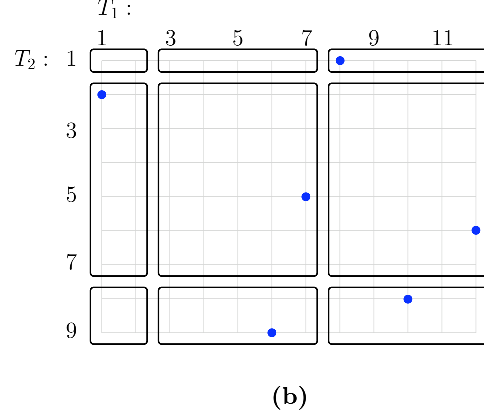
The space for the data structure (restricted to T_1 and T_2) is $O(|T_1| + |T_2| + m' \log^\epsilon n)$. We can assume without loss of generality that $|T_1| + |T_2| < 4m'$,[4] so the space for the ET-structure on T_1 and T_2 is $O(m' \log^\epsilon n)$. Since each non-tree edge only appears in one such structure the overall space for $\mathbf{ET}(G, \mathcal{F})$ is $O(m \log^\epsilon n)$. 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 $O(\min\{\log\log n, \log d_1\})$ query time using a van Emde Boas tree [23] or sorted list, whichever is faster.

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 Pǎtraşcu-Thorup structure requires modification and an additional logarithmic factor in the update time to maintain a spanning tree.

[3] The idea of representing the cross-product of trees as 2D point sets has been used in other contexts. See Grossi and Italiano [14].



(a)



(b)

Figure 1: (a) Dashed edges connect trees T_1 and T_2. The edges (u_2, u_3) and (v_1, v_2) have failed. (b) The point (i, j) is in our point set if (v_i, u_j) is a non-tree edge. The set of 2D range queries are indicated by solid boxes.

Corollary 2.2 demonstrates how $\mathbf{ET}(G, \cdot)$ can be used to answer connectivity queries avoiding edge and vertex failures.

COROLLARY 2.2. *The data structure $\mathbf{ET}(G, \{T\})$, where T is a spanning tree of $G = (V, E)$, supports the following operations. Given a set $D \subset E$ of edge failures, D can be processed in $O(|D|^2 \log\log n)$ time so that connectivity queries in the graph $(V, E \backslash D)$ can be answered in $O(\min\{\log\log n, \log|D|\})$ time. If $D \subset V$ is a set of vertex failures, the update time is $O((\sum_{v \in D} \deg_T(v))^2 \log\log n)$ and the query time $O(\min\{\log\log n, \log(\sum_{v \in D} \deg_T(v))\})$. (Note that the update time is independent of $\sum_{v \in D} \deg(v)$.)*

PROOF. Let d be the number of failed edges in T (or edges in T incident to failed vertices). Using $\mathbf{ET}(G, \{T\})$ we split

[4] The idea is simply to remove irrelevant vertices and contract long paths of degree-2 vertices. We leave this as an exercise.

T into $d+1$ subtrees and $L(T)$ 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 $O(d^2)$ time we find the connected components[5] of $V \backslash D$ 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 $O(\min\{\log\log n, \log d\})$ time. □

It is possible to reduce the query time using more complicated predecessor data structures [4, 12, 16]. However, in our vertex-failure connectivity oracle this cost is not the bottleneck.

3. CONSTRUCTING THE HIGH-DEGREE HIERARCHY

Theorem 2.1 and Corollary 2.2 demonstrate that given a spanning tree T with maximum degree t, we can processes d vertex failures in time roughly $(dt)^2$. However, there is no way to bound t as a function of d. Our solution is to build a *high-degree hierarchy* that represents the graph in a redundant fashion so that given d vertex failures, in some representation of the graph all failed vertices have low (relevant) degree.

3.1 Definitions

Let $\deg_H(v)$ be the degree of v in the graph H and let $\text{Hi}(H) = \{v \in V(H) \mid \deg_H(v) > s\}$ be the set of *high degree* vertices in H, where $s = \Omega(d^2)$ is a fixed parameter of the construction and d is an *upper bound* on the number of vertex failures. Increasing s will increase the update time and decrease the space.

We assign arbitrary distinct weights to the edges of the input graph $G = (V, E)$, which guarantees that every subgraph has a unique minimum spanning forest. Let X and Y be arbitrary subsets of vertices. We define F_X to be the minimum spanning forest of the graph $G \backslash X$. (The notation $G \backslash X$ is short for "the graph induced by $V \backslash X$.") Let $F_X(Y)$ be the subforest of F_X that preserves the connectivity of $Y \backslash X$, i.e., an edge appears in $F_X(Y)$ if it is on the path in F_X between two vertices in $Y \backslash X$. If X is omitted it is $\emptyset$. Note that $F_X(Y)$ may contain branching vertices that are not in $Y \backslash X$.

Lemma 3.1. *For sets X, Y, $|\text{Hi}(F_X(Y))| \le \lfloor \frac{|Y \backslash X| - 2}{s-1} \rfloor$.*

Proof. Note that all leaves of $F_X(Y)$ belong to $Y \backslash X$. We prove by induction that the maximum number of vertices with degree at least $s+1$ (the threshold for being high degree) in a tree with l leaves is precisely $\lfloor (l-2)/(s-1) \rfloor$. This upper bound holds whenever there is one internal vertex, and is clearly tight when $l \le s+1$. Given a tree with $l > s+1$ leaves and at least two internal vertices, select an internal vertex v adjacent to exactly one internal vertex and a maximum number of leaves. If v is incident to fewer than s leaves it can be spliced out without decreasing the number of high-degree vertices, so assume the number of incident leaves is at least s. Trimming the adjacent leaves of v leaves a tree with a net loss of $s-1$ leaves and 1 high degree vertex. The claim then follows from the inductive hypothesis. □

[5]This involves performing a depth first search of the graph whose vertices correspond to intervals in I.

3.2 The Hierarchy Tree and Its Properties

Definition 3.2 describes the hierarchy tree, and in fact shows that it is constructible in roughly linear time per hierarchy node. See Figure 2 for a diagram.

Definition 3.2. *The* hierarchy tree *is a rooted tree that is uniquely determined by the graph $G = (V, E)$, its artificial edge weights, and the parameters d and s. Nodes in the tree are identified with subsets of V. The root is V and every internal node has precisely d children. A (not necessarily spanning) forest of G is associated with each node and each edge in the hierarchy tree. The tree is constructed as follows:*

1. *Let W be a node with parent $p(W) = U$. We associate the forest $F(U)$ with U and $F_W(U)$ with the edge (U, W).*
2. *Let U be a node. If $F(U)$ has no high degree vertices then U is a leaf; otherwise it has children $W_1, \ldots, W_d$ defined as follows. (Here $\text{subtree}(X)$ is the set of descendants of X in the hierarchy, including X.)*

$$W_1 = \text{Hi}(F(U))$$

$$W_i = W_{i-1} \cup \left[U \cap \left(\bigcup_{\substack{W' \in \text{subtree}(W_{i-1}) \\ U' = p(W')}} \text{Hi}(F_{W'}(U')) \right) \right]$$

In other words, W_i inherits all the vertices from W_{i-1} and adds all vertices that are both in U and high-degree in some forest associated with an edge (W', U'), where W' is a descendant of W_{i-1}. Note that this includes the forest $F_{W_{i-1}}(U)$.

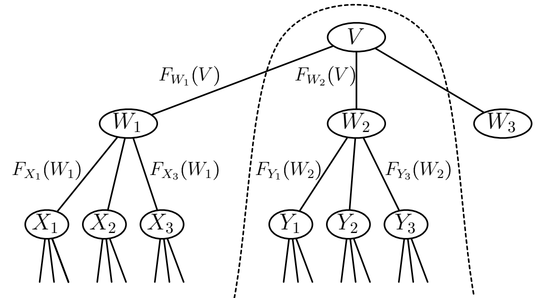


Figure 2: After W_1, W_2 and all their descendants have been constructed we construct W_3 as follows. Include all members of W_2 in W_3. Second, look at *all* hierarchy edges (X', U') where X' is in W_2's subtree and U' is the parent of X' (i.e., all edges under the dashed curve), and include all the high degree vertices in $F_{X'}(U')$ in W_3. In this example W_3 includes $\text{Hi}(F_{W_2}(V)), \text{Hi}(F_{Y_1}(W_2)), \text{Hi}(F_{Y_2}(W_2))$, and so on.

It is regretful that Definition 3.2(2) is so stubbornly unintuitive. We do not have a clean justification for it, except that it guarantees all the properties we require of the hierarchy: that it is small, shallow, and effectively represents the

1. $U_0 \leftarrow V$
2. For i from 1 to ∞ : {
3. If U_{i-1} is a leaf set $p \leftarrow i-1$ and HALT. (I.e., $U_{i-1} = U_p$ is the last node on the path.)
4. Let $W_1, \ldots, W_d$ be the children of U_{i-1} and artificially define $W_0 = \emptyset$ and $W_{d+1} = W_d$.
5. Let $j \in [0, d]$ be minimal such that $D \cap (W_{j+1} \backslash W_j) = \emptyset$.
6. If $j = 0$ set $p \leftarrow i-1$ and HALT. (I.e., $U_{i-1} = U_p$ is the last node on the path.)
7. Otherwise $U_i \leftarrow W_j$
8. }

Figure 3: A procedure for finding a path through the hierarchy, given a set D of failed vertices.

graph in many ways so that given d vertex failures, failed vertices have low degree in some graph representation. After establishing Lemmas 3.3–3.5, Definition 3.2(2) does not play any further role in the data structure whatsoever. Proofs of Lemmas 3.3 and 3.4 appear in the appendix.

LEMMA 3.3. **(Containment of Hierarchy Nodes)** *Let U be a node in the hierarchy tree with children $W_1, \ldots, W_d$. Then $\text{Hi}(F(U)) \subseteq U$ and $W_1 \subseteq \cdots \subseteq W_d \subseteq U$.*

LEMMA 3.4. **(Hierarchy Size and Depth)** *Consider the hierarchy tree constructed with high-degree threshold $s = (2d)^{c+1} + 1$, for some integer $c \geq 1$. Then:*

1. *The depth of the hierarchy is at most*

$$k = \lceil \log_{(s-1)/2d} n \rceil \leq \lceil (\log n)/(c \log(2d)) \rceil.$$

2. *The number of nodes in the hierarchy is on the order of $d^{-2/c} n^{1/c - 1/(c \log 2d)}$.*

For the remainder of the paper the variable k is fixed, as defined above. Aside from bounds on its size and depth, the only other property we require from the hierarchy tree is that, for any set of d vertex failures, all failures have low degree in forests along *some* path in the hierarchy. More formally:

LEMMA 3.5. *For any set D of at most d failed vertices, there exists a path $V = U_0, U_1, \ldots, U_p$ in the hierarchy tree such that all vertices in D have low degree in the forests $F_{U_1}(U_0), \ldots, F_{U_p}(U_{p-1}), F(U_p)$. Furthermore, this path can be found in $O(d(p+1)) = O(dk)$ time.*

PROOF. We construct the path $V = U_0, U_1, \ldots$ one node at a time using the procedure in Figure 3. First let us note that in Line 5 there always exists such a j, since we defined the artificial set $W_{d+1} = W_d$, and that this procedure eventually halts since the hierarchy tree is finite. If, during the construction of the hierarchy, we record for each $v \in U_{i-1}$ the *first* child of U_{i-1} in which v appears, Line 5 can easily be implemented in $O(d)$ time, for a total of $O((p+1)d) = O(dk)$ time.

Define $D_i = D \cap U_i$. It follows from Lemma 3.3 that $U_0 \supseteq \cdots \supseteq U_p$ and therefore that $D = D_0 \supseteq \cdots \supseteq D_p$. In the remainder of the proof we will show that:

(A) When the procedure halts, in Line 3 or 6, D is disjoint from $\text{Hi}(F(U_p))$.

(B) For each $i \in [1, p]$ and $i' \in [i, p]$, $D_{i-1} \backslash D_i$ is disjoint from $\text{Hi}(F_{U_{i'}}(U_{i'-1}))$.

Regarding (B), notice that for $i' \in [1, i)$, $D_{i-1} \backslash D_i$ is trivially disjoint from $\text{Hi}(F_{U_{i'}}(U_{i'-1}))$ because vertices in $D_{i-1} \backslash D_i \subseteq U_{i-1} \subseteq U_{i'}$ are specifically excluded from $F_{U_{i'}}(U_{i'-1})$. Thus, the lemma will follow directly from (A) and (B).

Proof of (A) Suppose the procedure halts at Line 3, i.e., $U_{i-1} = U_p$ is a leaf. By Definition 3.2(2), $\text{Hi}(F(U_p)) = \emptyset$ and is trivially disjoint from D. The procedure would halt at Line 6 if $j = 0$, meaning $W_1 \backslash W_0 = W_1$ is disjoint from D, where W_1 is the first child of $U_{i-1} = U_p$. This implies $\text{Hi}(F(U_p))$ is also disjoint from D since $W_1 = \text{Hi}(F(U_p))$ *by definition*.

Proof of (B) Fix an $i \in [1, p]$ and let $W_j = U_i$ be the child of U_{i-1} selected in Line 5. We first argue that if $j = d$ there is nothing to prove, then deal with the case $j \in [1, d-1]$. If $j = d$ that means the d disjoint sets $W_1, W_2 \backslash W_1, \ldots, W_d \backslash W_{d-1}$ each intersect D, implying that $U_i = W_d \supseteq D$ and therefore $D_i = D$. Thus $D_{i-1} \backslash D_i = \emptyset$ is disjoint from any set. Consider now the case when $j < d$, i.e., the node W_{j+1} exists and $W_{j+1} \backslash W_j$ is disjoint from D. By Definition 3.2(2) and the fact that $U_i, \ldots, U_p$ are descendants of $W_j = U_i$, we know that W_{j+1} includes all the high-degree vertices in $F_{U_i}(U_{i-1}), \ldots, F_{U_p}(U_{p-1})$ that are also in U_{i-1}. By definition, $D_{i-1} \backslash D_i$ is contained in U_{i-1} and disjoint from $U_i, \ldots, U_p$, implying that no vertex in $D_{i-1} \backslash D_i$ has high-degree in $F_{U_i}(U_{i-1}), \ldots, F_{U_p}(U_{p-1})$. If one did, it would have been put in W_{j+1} (as dictated by Definition 3.2(2)) and $W_{j+1} \backslash W_j$ would not have been disjoint from D, contradicting the choice of j. □

4. INSIDE THE HIERARCHY TREE

Lemma 3.5 guarantees that for any set D of d vertex failures, there exists a path of hierarchy nodes $V = U_0, \ldots, U_p$ such that all failures have low degree in the forests $F_{U_1}(U_0), \ldots, F_{U_p}(U_{p-1}), F(U_p)$. Using the ET-structure from Section 2 we can delete the failed vertices and reconnect the disconnected trees in $O(d^2 s^2 \log \log n)$ time for each of the $p+1$ levels of forests. This will allow us to quickly answer connectivity queries *within one level*, i.e., whether two vertices are connected in the subgraph induced by $V(F_{U_{i+1}}(U_i)) \backslash D$. However, to correctly answer connectivity queries we must consider paths that traverse many levels.

Our solution, following an idea of Chan et al. [7], is to augment the graph with *artificial* edges that capture the fact that vertices at one level (say in $U_i \backslash U_{i+1}$) are connected by a path whose intermediate vertices come from lower levels, in $V \backslash U_i$. 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 some artificial edges may become invalid and must be removed, which increases the recovery time. (In other words, an artificial edge (u, v) between $u, v \in U_i \backslash U_{i+1}$ indicates a u-to-v path via $V \backslash U_i$. If $V \backslash U_i$ suffers vertex failures then this path may no longer exist and the edge (u, v) is presumed invalid.) We add artificial edges so that after d vertex fail-

ures, we only need to remove a number of artificial edges that is polynomial in d, s, and $\log n$.

4.1 Stocking the Hierarchy Tree

The data structure described in this section (as well as all notation) are for a *fixed* path $V = U_0, \ldots, U_p$ in the hierarchy tree. In other words, for *each* path from the root to a descendant in the hierarchy we build a completely distinct data structure. In order to have a uniform notation for the forests at each level we artificially define $U_{p+1} = \emptyset$, so $F(U_p) = F_{U_{p+1}}(U_p)$. Furthermore, we let $\mathcal{F}$ represent the collection of forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$, which we construe as having *disjoint* vertex sets. For $i > j$ we say vertices in $U_i \backslash U_{i+1}$ are at a *higher level* than those in $U_j \backslash U_{j+1}$ and say the trees in the forest $F_{U_{i+1}}(U_i)$ are at a higher level than those in $F_{U_{j+1}}(U_j)$. Recall that $F_{U_{i+1}}(U_i)$ is the minimum spanning forest connecting $U_i \setminus U_{i+1}$ in the graph $G \backslash U_{i+1}$ and may contain vertices at lower levels. See Figure 4 in the appendix. We distinguish these two types of vertices:

Definition 4.1. **(Major Vertices)** *Vertices in $F_{U_{i+1}}(U_i)$ that are also in $U_i \setminus U_{i+1}$ are* major. *Let $T(u)$ be the* unique *tree in $\mathcal{F}$ in which u is a major vertex.*

It is not clear that the trees in $\mathcal{F}$ have any coherent organization. Lemma 4.3 shows that they naturally form a hierarchy, with trees in $F_{U_{p+1}}(U_p)$ on top. Below we give the definition of *ancestry* between trees and show each tree has exactly one ancestor at each higher level. See Figure 4 in the appendix for an illustration of Definitions 4.1 and 4.2.

Definition 4.2. **(Ancestry Between Trees)** *Let $0 \leq j \leq i \leq p$ and let T and T' be trees in $F_{U_{j+1}}(U_j)$ and $F_{U_{i+1}}(U_i)$, respectively. Call T' an* ancestor *of T (and T a descendant of T') if T and T' are in the same connected component in the graph $G \backslash U_{i+1}$. Notice that T is both an ancestor and descendant of itself.*

Lemma 4.3. **(Unique Ancestors)** *For $j \leq i \leq p$, each tree T in $F_{U_{j+1}}(U_j)$ has at most one ancestor in $F_{U_{i+1}}(U_i)$.*

Proof. Let T_1, T_2 be ancestors of T in $F_{U_{i+1}}(U_i)$, i.e., T_1 and T_2 span connected components in $G \backslash U_{i+1}$. Since they are *both* connected to T in $G \backslash U_{i+1}$ (which contains T since $U_{i+1} \subseteq U_{j+1}$), T_1 and T_2 are connected in $G \backslash U_{i+1}$ and cannot be distinct trees in $F_{U_{i+1}}(U_i)$. $\square$

Observe that the ancestry relation between trees T in $F_{U_{j+1}}(U_j)$ and T' in $F_{U_{i+1}}(U_i)$ is the *reverse* of the ancestry relation between the nodes U_j and U_i in the hierarchy tree. That is, if $j < i$, T' is an ancestor of T but U_j is an ancestor of U_i in the hierarchy tree.

Definition 4.4. **(Descendant Sets)** *The* descendant set *of T is $\Delta(T) = \{v \mid T(v)$ is a descendant of $T\}$. Equivalently, if T is in $F_{U_{i+1}}(U_i)$ then $\Delta(T)$ is the set of vertices in the connected component of $G \backslash U_{i+1}$ containing T.*

Lemma 4.5 is a simple consequence of the definitions of ancestry and descendant set, and one that will justify the way we augment the graph with artificial edges.

Lemma 4.5. **(Paths and Unique Descendant Sets)** *Consider a path between two vertices u and v and let w be an intermediate vertex (i.e., not u or v) with highest level. Then all intermediate vertices are in $\Delta(T(w))$ and each of $T(u)$ and $T(v)$ is either an ancestor or descendant of $T(w)$.*

Proof. This follows from the definition of $\Delta(\cdot)$. $\square$

Now that we have notions of ancestry and descendent sets, we are almost ready to describe exactly how we generate artificial edges. Recall that we are dealing with a fixed path $V = U_0, \ldots, U_p$ in the hierarchy tree. We construct a graph H that, among many other edges, includes all forest edges in $\mathcal{F}$. Since we interpret these forests as being on distinct vertex sets, H has at most $(p+1)n$ vertices, n of which are distinguished as major. In addition H includes all original edges of G, that is, for $(u, v) \in E(G)$ there is an edge in $E(H)$ connecting the major copies of u and v. Finally, H includes a collection of artificial edges; for each tree T in $\mathcal{F}$ there are edges that represent connectivity from ancestors of T via paths whose intermediate vertices are in $\Delta(T)$. To define the artificial edges with precision we must introduce some additional concepts. A d-adjacency list is essentially a path that is augmented to be resilient (in terms of connectivity) to up to d vertex failures.

Definition 4.6. **(d-Adjacency List)** *Let $L = (v_1, \ldots, v_r)$ be a list of vertices and $d \geq 1$ be an integer. The d-*adjacency edges *$\Lambda_d(L)$ connect all vertices at distance at most $d+1$ in the list L:*

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

Before proceeding we make some simple observations about d-adjacency lists.

Lemma 4.7. **(Properties of d-Adjacency Lists)** *The following properties hold for any vertex list L:*

1. *$\Lambda_d(L)$ contains fewer than $(d+1)|L|$ edges.*
2. *If a set D of d vertices are removed from L, the subgraph of $\Lambda_d(L)$ induced by $L \backslash D$ remains connected.*
3. *If L is split into lists L_1 and L_2, then we must remove $O(d^2)$ edges from $\Lambda_d(L)$ to obtain $\Lambda_d(L_1)$ and $\Lambda_d(L_2)$.*

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

We now have all the terminology necessary to define the graph H.

Definition 4.8. **(The Graph H)** *The edge set of H includes the forests in $\mathcal{F}$, the original edges in G (connecting their major counterparts), and $\bigcup_T C(T)$, where the union is over all trees T in $\mathcal{F}$ and $C(T)$ is constructed as follows:*

- *Let the strict ancestors of T be $T_1, T_2, \ldots, T_q$.*
- *For $1 \leq i \leq q$, let $A(T, T_i)$ be a list of the major vertices in T_i that are incident to some vertex in $\Delta(T)$, ordered according to an Euler tour of T_i. (This is done exactly as in Section 2.) Let $A(T)$ be the concatenation of $A(T, T_1), \ldots, A(T, T_q)$.*
- *Define $C(T)$ to be the edge set $\Lambda_d(A(T))$.*

See Figure 5 in the appendix for an illustration of how $C(T)$ is constructed. Lemma 4.9 exhibits the two salient properties of H: that it encodes useful connectivity information and that it is economical to effectively destroy $C(T)$ when it is no longer valid, often in time sublinear in $|C(T)|$.

LEMMA 4.9. **(Disconnecting $C(T)$)** *Consider a $C(T) \subseteq E(H)$, where T is a tree in $\mathcal{F}$.*

1. *Suppose d vertices fail, none of which are in $\Delta(T)$, and let u and v be major vertices in ancestors of T that are adjacent to at least one vertex in $\Delta(T)$. Then u and v remain connected in the original graph and remain connected in H.*

2. *Suppose the proper ancestors of T are $T_1, \ldots, T_q$ and a total of f edges are removed from these trees, breaking them into subtrees $T'_1, \ldots, T'_{q+f}$. Then at most $O(d^2(q+f))$ edges must be removed from $C(T)$ such that no remaining edge in $C(T)$ connects distinct trees T'_i and T'_j.*

PROOF. For Part (1), the vertices u and v are connected in the original graph because they are each adjacent to vertices in $\Delta(T)$ and, absent any failures, all vertices in $\Delta(T)$ are connected, by definition. By Definition 4.8, u and v appear in $C(T)$ and, by Lemma 4.7, $C(T)$ remains connected after the removal of any d vertices. Turning to Part (2), recall from Definition 4.8 that $A(T)$ was the concatenation of $A(T, T_1), \ldots, A(T, T_q)$ and each $A(T, T_i)$ was ordered according to an Euler tour of T_i. Removing f edges from $T_1, \ldots, T_q$ separates their Euler tours (and, hence, the lists $\{A(T, T_i)\}_i$) into at most $2f + q$ intervals. (This is exactly the same reasoning used in Section 2.) By Lemma 4.7 we need to remove at most $(2f + q - 1) \cdot O(d^2)$ edges from $C(T)$ to guarantee that all remaining edges are internal to one such interval, and therefore internal to one of the trees $T'_1, \ldots, T'_{q+f}$. Note that $C(T)$ is now "logically" deleted since remaining edges internal to some T'_i do not add any connectivity. $\square$

Finally, we associate the ET-structure $\mathbf{ET}(H, \mathcal{F})$ with the path $V = U_0, \ldots, U_p$. Lemma 4.10 bounds the space for the overall data structure.

LEMMA 4.10. **(Space Bounds)** *Given a graph G with m edges, n vertices, and parameters d and $s = (2d)^{c+1} + 1$, where $c \geq 1$, the space for a d-failure connectivity oracle is $O(d^{1-2/c} m n^{1/c - 1/(c \log(2d))} \log^2 n)$.*

PROOF. Recall that $k = \log_{(s-1)/2d} n < \log n$ is the height of the hierarchy. The graph H has at most $(p+1)n \leq kn$ vertices, and, we claim, less than $kn + [(d+1)k + 1]m$ edges. There are less than kn edges in the forests $\mathcal{F}$. Each original edge (u, v) appears in H and causes v to make an appearance in the list $A(T, T(v))$, whenever $u \in \Delta(T)$. There are at most k such lists. Moreover, v's appearance in $A(T, T(v))$ (and hence $A(T)$) contributes at most $d + 1$ edges to $C(T) = \Lambda_d(A(T))$. By Theorem 2.1, each edge in H contributes $O(\log n)$ space in $\mathbf{ET}(H, \mathcal{F})$ for a total of $O((dkm + kn) \log n) = O(dm \log^2 n)$ space for *one* hierarchy node. By Lemma 3.4 there are $d^{-2/c} n^{1/c - 1/(c \log(2d))}$ hierarchy tree nodes, which gives the claimed bound. $\square$

5. RECOVERY FROM FAILURES

In this section we describe how, given up to d failed vertices, the structure can be updated in $O((dsk)^2 \log \log n)$ time 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.

5.1 Deleting Failed Vertices

Step 1. Given the set D of at most d failed vertices, we begin by identifying a path $V = U_0, \ldots, U_p$ in the hierarchy in which D have low degree in the $p + 1$ levels of forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$. By Lemma 3.5 this takes $O(d \log n)$ time.

In subsequent steps we delete all failed vertices in each of their appearances in the forests, i.e., up to $p + 1 \leq k$ copies for each failed vertex. Edges remaining in H (between vertices not in D) represent original edges, which are obviously still valid, or edges in $\bigcup_T C(T)$, which might no longer be valid. Recall that $C(T)$ represents connectivity via a path whose intermediate vertices are in the descendant set $\Delta(T)$. If $\Delta(T)$ contains failed vertices then that path may no longer exist, so *all edges* in $C(T)$ become suspect, and are presumed invalid. Although $C(T)$ may contain many edges, Lemma 4.9(2) implies that $C(T)$ can be *logically* destroyed in time polynomial in d and s. Before describing the next steps in detail we need to distinguish affected from unaffected trees.

DEFINITION 5.1. **(Affected Trees)** *If a tree T in $\mathcal{F}$ intersects the set of failed vertices D, T and all ancestors of T are* affected. *Equivalently, T is affected if $\Delta(T)$ contains a failed vertex. If T is affected, the connected subtrees of T induced by $V(T) \backslash D$ (i.e., the subtrees remaining after vertices in D fail) are called* affected subtrees*.*

LEMMA 5.2. **(The Number of Affected Trees)** *The number of affected trees is at most kd. The number of affected subtrees is at most $kd(s+1)$.*

PROOF. If u is a major vertex in T, u can only appear in ancestors of T. Thus, when u fails it can cause at most k trees to become affected. Since, by choice of U_p, all failed vertices have low degree in the trees in which they appear, at most kds tree edges are deleted, yielding $kd(s+1)$ affected subtrees. $\square$

Step 2. We identify the affected trees in $O(kd)$ time and mark as deleted the tree edges incident to failed vertices in $O(kds)$ time. Deleting $O(kds)$ tree edges effectively splits the Euler tours of the affected trees into $O(kds)$ intervals, where each affected subtree is the union of some subset of the intervals.

Step 3. Recall from the discussion above that if T is an affected tree then $\Delta(T)$ contains failed vertices and the connectivity provided by $C(T)$ is presumed invalid. By Lemma 4.9 we can logically delete $C(T)$ by removing $O(d^2)$ edges for *each* edge removed from an ancestor tree of T i.e., $O(d^2 \cdot kds)$ edges need to be removed to destroy $C(T)$. (All remaining edges from $C(T)$ are internal to some affected subtree and can therefore be ignored; they do not provide additional connectivity.) There are at most dk affected trees T, so at most $O(k^2 d^4 s)$ edges need to be removed from H. Let H' be H with these edges removed.

Step 4. We now attempt to reconnect all affected subtrees using valid edges, i.e., those not deleted in Step 3. Let R be a graph whose vertices $V(R)$ represent the $O(kds)$ affected subtrees such that $(t_1, t_2) \in E(R)$ if t_1 and t_2 are connected by an edge from H'. Using the structure $\mathbf{ET}(H, \mathcal{F})$ (see Section 2, Theorem 2.1), we populate the edge set in time $O(|V(R)|^2 \log \log n + k^2 d^4 s)$, which is $O((dsk)^2 \log \log n)$ since $s > d^2$. In $O(|E(R)|) = O((dsk)^2)$ time we determine

the connected components of R and store with each affected subtree a representative vertex of its component.

This concludes the deletion algorithm. The running time is dominated by Step 4.

5.2 Answering a Connectivity Query

The deletion algorithm has already identified the path $U_0, \ldots, U_p$. 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 find $T(u)$ and $T(v)$ in $O(1)$ time; recall that these are trees in which u and v are major vertices. If $T(u)$ is unaffected, let T_1 be the most ancestral unaffected ancestor of $T(u)$, and let T_2 be defined in the same way for $T(v)$. If $T_1 = T_2$ then $\Delta(T_1)$ contains u and v but no failed vertices; if this is the case we declare u and v *connected* and stop. We can find T_1 and T_2 in $O(\log k) = O(\log \log n)$ time using a binary search over the ancestors of $T(u)$ and $T(v)$, or in $O(\log d)$ time using predecessor search and a level ancestor data structure.

Step 2. We now try to find vertices u' and v' in affected subtrees that are connected to u and v respectively. If $T(u)$ is affected then $u' = u$ clearly suffices, so we only need to consider the case when $T(u)$ is unaffected and T_1 exists. Recall from Definition 4.8 that $A(T_1)$ is the list of major vertices in proper ancestors of T_1 that are adjacent to some vertex in $\Delta(T_1)$. We scan $A(T_1)$ looking for *any* non-failed vertex u' adjacent to $\Delta(T_1)$. Since $\Delta(T_1)$ is unaffected, u is connected to u', and since T_1's parent is affected u' must be in an affected subtree. Since there are at most d failed vertices we must inspect at most $d+1$ elements of $A(T_1)$. 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. Given u' and v', in $O(\min\{\log \log n, \log d\})$ time we find the affected subtrees t_1 and t_2 containing u' and v', respectively. Note that t_1 and t_2 are vertices in R, from Step 4 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.

We now turn to the correctness of the query algorithm. If the algorithm replies *connected* in Step 1 or *disconnected* in Step 2 it is clearly correct. (This follows directly from the definitions of $\Delta(T_i)$ and $A(T_i)$, for $i \in \{1, 2\}$.) If u' and v' are discovered then u and v are clearly connected to u' and v', again, by definition of $\Delta(T_i)$ and $A(T_i)$. Thus, we may assume without loss of generality that the query vertices $u = u'$ and $v = v'$ lie in affected subtrees. The correctness of the procedure therefore hinges on whether the graph R correctly represents connectivity between affected subtrees.

LEMMA 5.3. **(Query Algorithm Correctness)** *Let u and v be vertices in affected subtrees t_u and t_v. Then there is a path from u to v avoiding failed vertices if and only if t_u and t_v are connected in R.*

PROOF. Edges in R represent either original graph edges (not incident to failed vertices) or paths whose intermediate vertices lie in some $\Delta(T)$, for an unaffected T. Thus, if there is a path in R from t_u to t_v then there is also a path from u to v avoiding failed vertices. For the reverse direction, let P be a path from u to v in the original graph avoiding failed vertices. If all intermediate vertices in P are from affected subtrees then P clearly corresponds to a path in R, since all inter-affected-tree edges in P are included in H' and eligible to appear in R. For the last case, let $P = (u, \ldots, x, x', \ldots, y', y, \ldots, v)$, where x' is the *first* vertex not in an affected tree and y is the first vertex following x' in an affected tree. That is, the subpath $(x', \ldots, y')$ lies entirely in $\Delta(T)$ for some unaffected tree T, which implies that x and y appear in $A(T)$. By Lemma 4.9, x and y remain connected in $C(T)$ even if d vertices are removed, implying that x and y remain connected in H'. Since all edges from H' are eligible to appear in R, t_x and t_y must be connected in R. Thus, u lies in t_u, which is connected to t_x in R, which is connected to t_y in R. The claim then follows by induction on the (shorter) path from y to v. $\square$

6. CONCLUSION

We presented the first space-efficient data structure for one of the most natural dynamic graph problems: given that a set of vertices has *failed*, is there still a path from point A to point B avoiding all failures? Our connectivity oracle recovers from d vertex failures in time polynomial in d and answers connectivity queries in time linear in d. Are these types of time bounds intrinsic to the problem? Excluding polylogarithmic factors the best we could hope for is $\tilde{O}(m)$ space, $\tilde{O}(d)$ time to process vertex failues and $\tilde{O}(1)$ time for queries.

In addition to our vertex-failure oracle we presented a new edge-failure connectivity oracle that is incomparable to a previous structure of Pătraşcu and Thorup [17] in many ways. We note that it excels when the number of failures is small; for $d = O(1)$ the oracle recovers from failures in $O(\log \log n)$ time and answers connectivity queries in $O(1)$ time.

7. REFERENCES

[1] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings 41st FOCS*, pages 198–207, 2000.

[2] S. Arora, E. Hazan, and S. Kale. $O(\sqrt{\log n})$ approximation to sparsest cut in $\tilde{O}(n^2)$ time. In *Proceedings 45th FOCS*, pages 238–247, 2004.

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

[4] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

[5] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings 41st STOC*, pages 101–110, 2009.

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

[7] T. M. Chan, M. Pătraşcu, and L. Roditty. Dynamic connectivity: Connecting to networks and geometry. In *Proceedings 49th FOCS*, pages 95–104, 2008.

[8] C. Demetrescu and G. F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.

[9] 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.

[10] R. Duan and S. Pettie. Bounded-leg distance and reachability oracles. In *Proceedings 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 436–445, 2008.

[11] 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.
[12] M. L. Fredman and D. E. Willard. Surpassing the information-theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.
[13] D Frigioni and G. F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000.
[14] R. Grossi and G. F. Italiano. Efficient splitting and merging algorithms for order decomposable problems. *Information and Computation*, 154(1):1–33, 1999.
[15] 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.
[16] 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.
[17] 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.
[18] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings 36th ACM Symposium on Theory of Computing (STOC)*, pages 184–191, 2004.
[19] P. Sankowski. Faster dynamic matchings and vertex connectivity. In *Proceedings 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 118–126, 2007.
[20] J. Sherman. Breaking the multicommodity flow barrier for $O(\sqrt{\log n})$-approximations to sparsest cut. *CoRR*, abs/0908.1379, 2009.
[21] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings 37th ACM Symposium on Theory of Computing (STOC)*, pages 112–119, 2005.
[22] M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.
[23] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings 39th FOCS*, pages 75–84, 1975.

APPENDIX

A. PROOFS AND FIGURES

Lemma 3.3 Let U be a node in the hierarchy tree with children $W_1, \ldots, W_d$. Then $\mathrm{Hi}(F(U)) \subseteq U$ and $W_1 \subseteq \cdots \subseteq W_d \subseteq U$.

Proof. The second claim will be established in the course of proving the first claim. We prove the first claim by induction on the preorder (depth first search traversal) of the hierarchy tree. For the root node V, $\mathrm{Hi}(V)$ is trivially a subset of V. Let W_i be a node, U be its parent, and W_1 be U's first child, which may be the same as W_i. Suppose the claim is true for all nodes preceding W_i. If it is the case that $W_i = W_1$, we have that $W_1 = \mathrm{Hi}(F(U))$ (by Definition 3.2(2)) and $\mathrm{Hi}(F(U)) \subseteq U$ (by the inductive hypothesis). Since $F(W_1)$ is a subforest of $F(U)$ (this follows from the fact that for a vertex set Y we select $F(Y)$ to be the *minimum* forest spanning Y), every high degree vertex in $F(W_1)$ also has high degree in $F(U)$, i.e., $\mathrm{Hi}(F(W_1)) \subseteq \mathrm{Hi}(F(U)) = W_1$, which establishes the claim when $W_i = W_1$. Once we know that $W_1 \subseteq U$ it follows from Definition 3.2(2) that $W_1 \subseteq \cdots \subseteq W_d \subseteq U$. By the same reasoning as above, when $W_i \neq W_1$, we have that $W_i \subseteq U$, implying that $F(W_i)$ is a subforest of $F(U)$, which implies that $\mathrm{Hi}(F(W_i)) \subseteq \mathrm{Hi}(F(U)) = W_1 \subseteq W_i$. □

The claims of Lemma 3.4 will follow directly from Parts (1) and (3) of Lemma A.1

Lemma A.1. *Consider the hierarchy tree constructed with high-degree threshold $s = (2d)^{c+1}+1$, for some integer $c \geq 1$. Let U be a non-root node in the hierarchy tree and $W_1, \ldots, W_d$ be its children, if U is not a leaf. Let $p(X)$ be the parent of X and* subtree(X) *be the set of descendants, including X, in the hierarchy tree. Then:*

1. *For $i \in [1, d]$, $|W_i| \leq 2i|U|/(s-1)$.*
2. *$\sum_{X \in \mathrm{subtree}(U)} |\mathrm{Hi}(F_X(p(X)))| \leq 2|p(U)|/(s-1)$.*
3. *The number of nodes in the hierarchy is on the order of $d^{-2/c} n^{1/c - 1/(c\log(2d))}$.*

Proof. We prove Parts (1) and (2) by induction over the postorder of the hierarchy tree. In the base case U is a leaf, (1) is vacuous and (2) is trivial, since there is one summand, namely $|\mathrm{Hi}(F_U(p(U)))|$, which is at most $(|p(U)|-2)/(s-1)$ by Lemma 3.1. For Part (1), in the base case $|W_1| < |U|/(s-1)$. For $i \in [2, d]$ we have:

$$\begin{aligned}|W_i| &\leq |W_{i-1}| + \sum_{X \in \mathrm{subtree}(W_{i-1})} |\mathrm{Hi}(F_X(p(X)))| \\ &\leq \frac{2(i-1)|U|}{s-1} + \frac{2|U|}{s-1} && \{\text{Ind. hyp. (1) and (2)}\} \\ &= \frac{2i|U|}{s-1}\end{aligned}$$

For Part (2) we have:

$$\begin{aligned}&\sum_{X \in \mathrm{subtree}(U)} |\mathrm{Hi}(F_X(p(X)))| \\ &= |\mathrm{Hi}(F_U(p(U)))| + \sum_{i=1}^{d} \sum_{X \in \mathrm{subtree}(W_i)} |\mathrm{Hi}(F_X(p(X)))| \\ &&& \{\text{Follows from defn. of subtree}\} \\ &< \frac{|p(U)|}{s-1} + \frac{2d|U|}{s-1} && \{\text{Lemma 3.1, Ind. hyp. (2)}\} \\ &\leq \frac{|p(U)|}{s-1} + \frac{2d[2d|p(U)|/(s-1)]}{s-1} && \{\text{Ind. hyp. (1)}\} \\ &\leq \frac{2|p(U)|}{s-1} && \{s \geq 4d^2+1\}\end{aligned}$$

We prove Part (3) for a slight modification of the hierarchy tree in which U is forced to be a leaf if $|U| \leq 2ds$. This change has no effect on the running time of the algorithm.[6] Consider the set of intervals $\{B_j\}$ where $B_j = [(2d)^j, (2d)^{j+1})$, and let l_j be the maximum number of leaf descendants of a node U for which $|U| \in B_j$. If $|U| \leq (2d)s$ then U is a leaf, i.e., $l_j = 1$ for $j \leq c+1$. Part (1) implies that if $|U|$ lies in B_j then each child lies in either B_{j-c-1} or B_{j-c}. Hence, $l_j \leq d \cdot l_{j-c}$, and, by induction, $l_j \leq d^{\lfloor (j-2)/c \rfloor}$. Now suppose that n lies in the interval $[(2d)^{cx+2}, (2d)^{(c+1)x+2}) = B_{cx+2} \cup \cdots \cup B_{(c+1)x+1}$. Then the number of leaf descendants of V, the hierarchy tree root, is at most $d^c < n^{1/c} 2^{-x} d^{-2/c} \leq n^{1/c - 1/(c\log(2d))} d^{-2/c}$. □

[6] We only require that in a leaf node U, any set of d failed vertices are incident to a total of $O(ds)$ tree edges from $F(U)$, i.e., that the average degree in $F(U)$ is $O(s)$. We do not require that *every* failed vertex be low degree in $F(U)$.

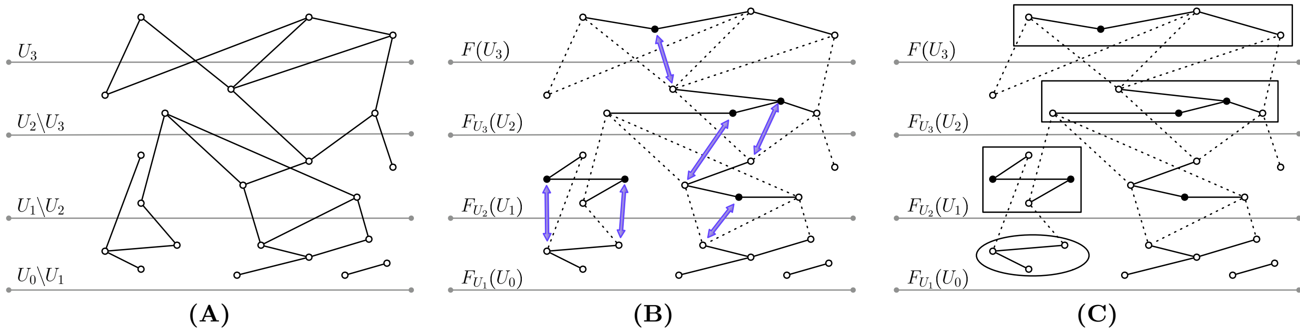


Figure 4: (A) A path $U_0, \ldots, U_3$ in the hierarchy tree (where $V = U_0$ is the root) naturally partitions the vertices into four levels $U_0\backslash U_1, U_1\backslash U_2, U_2\backslash U_3$, and U_3. (B) The forest $F_{U_{i+1}}(U_i)$ may contain "copies" of vertices from lower levels. (Hollow vertices are *major* vertices at their level; solid ones are copies from a lower level. Thick arrows associate a copy with its original major vertex.) (C) A tree T in $F_{U_{j+1}}(U_j)$ is a descendant of T' in $F_{U_{i+1}}(U_i)$ (where $j \leq i$) if T and T' are connected in $G\backslash U_{i+1}$. The tree inscribed in the oval is a descendant of those trees inscribed in rectangles.

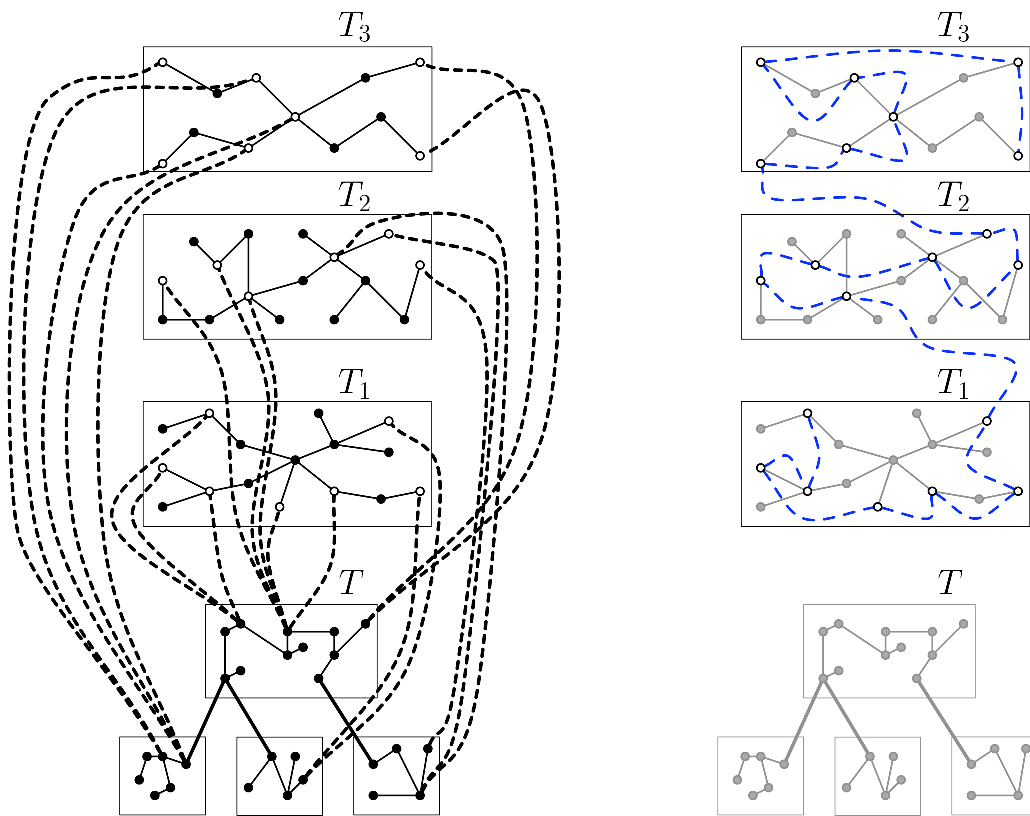


Figure 5: Left: T is a tree in some forest among $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ having three strict descendants and three ancestors T_1, T_2, T_3. Dashed curves indicate edges connecting vertices from $\Delta(T)$ (all vertices in descendants of T) to major vertices in strict ancestors of T, which are drawn as hollow. Right: The set $C(T)$ consists of, first, linking up all hollow vertices in a list that is consistent with Euler tours of T_1, T_2, T_3 (indicated by dashed curves), and second, adding edges between all hollow vertices at distance at most $d+1$ in the list.