Connectivity Queries Under Vertex Failures: Not Optimal, but Practical

Evangelos Kosinas
Department of Computer Science & Engineering, University of Ioannina, Greece

Abstract

We revisit once more the problem of designing an oracle for answering connectivity queries in undirected graphs in the presence of vertex failures. Specifically, given an undirected graph G with n vertices and m edges and an integer $d_\star \ll n$, the goal is to preprocess the graph in order to construct a data structure $\mathcal{D}$ such that, given a set of vertices F with $|F| = d \leq d_\star$, we can derive an oracle from $\mathcal{D}$ that can efficiently answer queries of the form "is x connected with y in $G \setminus F$?". Very recently, Long and Saranurak (FOCS 2022) provided a solution to this problem that is almost optimal with respect to the preprocessing time, the space usage, the update time, and the query time. However, their solution is highly complicated, and it seems very difficult to be implemented efficiently. Furthermore, it does not settle the complexity of the problem in the regime where $d_\star$ is a constant. Here, we provide a much simpler solution to this problem, that uses only textbook data structures. Our algorithm is deterministic, it has preprocessing time and space complexity $O(d_\star m \log n)$, update time $O(d^4 \log n)$, and query time $O(d)$. These bounds compare very well with the previous best, especially considering the simplicity of our approach. In fact, if we assume that $d_\star$ is a constant ($d_\star \geq 4$), then our algorithm provides some trade-offs that improve the state of the art in some respects. Finally, the data structure that we provide is flexible with respect to $d_\star$: it can be adapted to increases and decreases, in time and space that are almost proportional to the change in $d_\star$ and the size of the graph.

2012 ACM Subject Classification Mathematics of computing → Paths and connectivity problems; Theory of computation → Graph algorithms analysis

Keywords and phrases Graphs, Connectivity, Fault-Tolerant, Oracles

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.75

Related Version *Full Version*: `https://arxiv.org/abs/2305.01756`

Funding The research work was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the 3rd Call for HFRI PhD Fellowships (Fellowship Number: 6547.).

Acknowledgements I want to thank my advisor, Loukas Georgiadis, for helpful comments on this manuscript. I also want to thank the anonymous reviewers for their useful suggestions.

1 Introduction

In this paper we deal with the following problem. Given an undirected graph G with n vertices and m edges, and a fixed integer $d_\star$ ($d_\star \ll n$), the goal is to construct a data structure $\mathcal{D}$ that can be used in order to answer connectivity queries in the presence of at most $d_\star$ vertex-failures. More precisely, given a set of vertices F, with $|F| \leq d_\star$, we must be able to efficiently derive an oracle from $\mathcal{D}$, which can efficiently answer queries of the form "are the vertices x and y connected in $G \setminus F$?". In this problem, we want to simultaneously optimize the following parameters: (1) the construction time of $\mathcal{D}$ (preprocessing time), (2) the space usage of $\mathcal{D}$, (3) the time to derive the oracle from $\mathcal{D}$ given F (update time), and (4) the time to answer a connectivity query in $G \setminus F$. This problem is very well motivated; it has attracted the attention of researchers for more than a decade now, and it has many interesting variations. The reader is referred to [5] or [7] for the details on its history and its variations.

1.1 Previous work

Despite being extensively studied, it is only very recently that an almost optimal solution was provided by Long and Saranurak [7]. Specifically, they provided a deterministic algorithm that has $\hat{O}(m) + \tilde{O}(d_\star m)$ preprocessing time, uses $O(m \log^* n)$ space, and has $\hat{O}(d^2)$ update time and $O(d)$ query time.[1] This improves on the previous best deterministic solution by Duan and Pettie [5], that has $O(mn \log n)$ preprocessing time, uses $O(d_\star m \log n)$ space, and has $O(d^3 \log^3 n)$ update time and $O(d)$ query time. We note that there are more solutions to this problem, that optimize some parameters while sacrifising others (e.g., in the solution of Pilipczuk et al. [9], there is no dependency on n in the update time, but this is superexponential in $d_\star$, and the preprocessing time is $O(mn^2 2^{2^{O(d_\star)}})$). We refer to Table 1 in reference [7] for more details on the best known (upper) bounds for this problem. We also refer to Theorem 1.2 in [7] for a summary of known (conditional) lower bounds, that establish the optimality of [7].

1.2 Our contribution

The bounds that we mentioned are the best known for a deterministic solution. In practice, one would prefer the solution of Long and Saranurak [7], because that of Duan and Pettie [5] has preprocessing time $O(mn \log n)$, which can be prohibitively slow for large enough graphs. However, the solution in [7] is highly complicated, and it seems very difficult to be implemented efficiently. This is a huge gap between theory and practice. Furthermore, the (hidden) dependence on n in the time-bounds of [7] is not necessarily optimal if we assume that $d_\star$ is a constant for our problem. We note that this is a problem with various parameters, and thus it is very difficult to optimize all of them simultaneously.

Considering that this is a fundamental connectivity problem, we believe that it is important to have a solution that is relatively simple to describe and analyze, compares very well with the best known bounds (even improves them in some respects), opens a new direction to settle the complexity of the problem, and can be readily implemented efficiently.

In this paper, we exhibit a solution that has precisely those characteristics. We present a deterministic algorithm that has preprocessing time $O(d_\star m \log n)$, uses space $O(d_\star m \log n)$, and has $O(d^4 \log n)$ update time and $O(d)$ query time.[2] Our approach is arguably the simplest that has been proposed for this problem. The previous solutions rely on sophisticated tree decompositions of the original graph. Here, instead, we basically rely on a single DFS-tree, and we simply analyze its connected components after the removal of a set of vertices. It turns out that there is enough structure to allow for an efficient solution (see Section 3.2).

The bounds that we provide compare very well with the previous best, especially considering the simplicity of our approach. (See Tables 1 and 2.) In fact, as we can see in Table 1, our solution is the best choice for implementations, considering that the algorithm of Long and Saranurak is very difficult to be implemented within the claimed time-bounds. Furthermore, if we assume that $d_\star$ is a constant ($d_\star \geq 4$), then, as we can see in Table 2, our algorithm provides some trade-offs, that improve the state of the art in some respects.

---

[1] The symbol $\hat{O}$ hides subpolynomial (i.e. $n^{o(1)}$) factors, and $\tilde{O}$ hides polylogarithmic factors. The hidden expressions in the time-bounds are not specified by the authors in their overview. Also, the description for the $\log^* n$ function that appears in the space complexity is that it "can be substituted with any slowly growing function". One thing that is explicitly stated, however, is that the hidden subpolynomial factors are worse than polylogarithmic. We must emphasize that the difficulty in stating the precise bounds is partly due to there being various trade-offs in the functions involved, and is partly indicative of the complexity of the techniques that are used.

[2] The log factors in the space usage and the time for the updates can be improved with the use of more sophisticated 2D-range-emptiness data structures, such as those in [2].

Table 1 Comparison of the best-known deterministic bounds. We note that m can be replaced with $\bar{m} = min\{m, d_\star n\}$, using the sparsification of Nagamochi and Ibaraki [8]. The data structure of Pilipczuk et al. does not support an update phase, but answers queries directly, given a set of (at most $d_\star$) failed vertices and two query vertices.

	Preprocessing	Space	Update	Query
Pilipczuk et al. [9]	$O(2^{2^{O(d_\star)}} mn^2)$	$O(2^{2^{O(d_\star)}} m)$	–	$O(2^{2^{O(d_\star)}})$
Duan and Pettie [5]	$O(mn \log n)$	$O(d_\star m \log n)$	$O(d^3 \log^3 n)$	$O(d)$
Long and Saranurak [7]	$\hat{O}(m) + \tilde{O}(d_\star m)$	$O(m \log^* n)$	$\hat{O}(d^2)$	$O(d)$
This paper	$O(d_\star m \log n)$	$O(d_\star m \log n)$	$O(d^4 \log n)$	$O(d)$

Table 2 Comparison of the best-known deterministic bounds, when $d_\star$ is a fixed (small) constant. Although the algorithm of Pilipczuk et al. has the best space and query-time bounds, it has very large preprocessing time. Our solution has the best preprocessing time, and also better update time compared to the solutions of [5] and [7]. Furthermore, our space usage is almost linear.

	Preprocessing	Space	Update	Query
Pilipczuk et al. [9]	$O(mn^2)$	$O(m)$	–	$O(1)$
Duan and Pettie [5]	$O(mn \log n)$	$O(m \log n)$	$O(\log^3 n)$	$O(1)$
Long and Saranurak [7]	$\hat{O}(m) + \tilde{O}(m)$	$O(m \log^* n)$	$\hat{O}(1)$	$O(1)$
This paper	$O(m \log n)$	$O(m \log n)$	$O(\log n)$	$O(1)$

Finally, the data structure that we provide is flexible with respect to $d_\star$: it can be adapted to increases and decreases, in time and space that are almost proportional to the change in $d_\star$ and the size of the graph (see Corollary 3). We do not know if any of the previous solutions has this property. It is a natural question whether we can efficiently update the data structure so that it can handle more failures (or less, and thereby free some space). As far as we know, we are the first to take notice of this aspect of the problem.

2 Preliminaries

We assume that the reader is familiar with standard graph-theoretical terminology (see, e.g., [4]). The notation that we use is also standard. Since we deal with connectivity under *vertex* failures, it is sufficient to consider simple graphs as input to our problem (because the existence of parallel edges does not affect the connectivity relation). However, during the update phase, we construct a multigraph that represents the connectivity relationship between some connected components after removing the failed vertices (Definition 9). The parallel edges in this graph are redundant, but they may be introduced by the algorithm that we use to construct it, and it would be costly to check for redundancy throughout.

It is also sufficient to assume that the input graph G is connected. Because, otherwise, we can initialize a data structure on every connected component of G; the updates, for a given set of failures, are distributed to the data structures on the connected components, and the queries for pairs of vertices that lie in different connected components of G are always *false*. We use G to denote the input graph throughout; n and m denote its number of vertices and edges, respectively. For any two integers x, y, we use the interval notation $[x, y]$ to denote the set $\{x, x+1, \ldots, y\}$. (If $x > y$, then $[x, y] = \emptyset$.)

2.1 DFS-based concepts

Let T be a DFS-tree of G, with start vertex r [10]. We use $p(v)$ to denote the parent of every vertex $v \neq r$ in T (v is a child of $p(v)$). For any two vertices u, v, we let $T[u, v]$ denote the simple tree path from u to v on T. For every two vertices u and v, if the tree path $T[r, u]$ uses v, then we say that v is an ancestor of u (equivalently, u is a descendant of v). In particular, a vertex is considered to be an ancestor (and also a descendant) of itself. It is very useful to identify the vertices with their order of visit during the DFS, starting with $r \leftarrow 1$. Thus, if v is an ancestor of u, we have $v < u$. For any vertex v, we let $T(v)$ denote the subtree rooted at v, and we let $ND(v)$ denote the number of descendants of v (i.e., $ND(v) = |T(v)|$). Thus, we have that $T(v) = [v, v + ND(v) - 1]$, and therefore we can check the ancestry relation in constant time. Two children c and c' of a vertex v are called *consecutive children* of v (in this order), if c' is the minimum child of v with $c' > c$. Notice that, in this case, we have $T(c) \cup T(c') = [c, c' + ND(c') - 1]$.

A DFS-tree has the following extremely convenient property: the endpoints of every non-tree edge of G are related as ancestor and descendant [10], and so we call those edges *back-edges*. Our whole approach is basically an exploitation of this property, which does not hold in general rooted spanning trees of G (unless they are derived from a DFS traversal, and only then [10]). To see why this is relevant for our purposes, consider what happens when we remove a vertex $f \neq r$ from T. Let $c_1, \ldots, c_k$ be the children of f in T. Then, the connected components of $T \setminus f$ are given by $T(c_1), \ldots, T(c_k)$ and $T(r) \setminus T(f)$. A subtree $T(c_i)$, $i \in \{1, \ldots, k\}$, is connected with the rest of the graph in $G \setminus f$ if and only if there is a back-edge that stems from $T(c_i)$ and ends in a proper ancestor of f. Now, this problem has an algorithmically elegant solution. Suppose that we have computed, for every vertex $v \neq r$, the *lowest* proper ancestor of v that is connected with $T(v)$ through a back-edge. We denote this vertex as $low(v)$. Then, we may simply check whether $low(c_i) < f$, in order to determine whether $T(c_i)$ is connected with $T(r) \setminus T(f)$ in $G \setminus f$.

We extend the concept of the low points, by introducing the low_k points, for any $k \in \mathbb{N}$. These are defined recursively, for any vertex $v \neq r$, as follows. $low_1(v)$ coincides with $low(v)$. Then, supposing that we have defined $low_k(v)$ for some $k \in \mathbb{N}$, we define $low_{k+1}(v)$ as $min(\{y \mid \exists$ a back-edge (x, y) such that $x \in T(v)$ and $y < v\} \setminus \{low_1(v), \ldots, low_k(v)\})$. Notice that $low_k(v)$ may not exist for some $k \in \mathbb{N}$ (and this implies that $low_{k'}(v)$ does not exist, for any $k' > k$). If, however, $low_k(v)$ exists, then $low_{k'}(v)$, for any $k' < k$, also exists, and we have $low_1(v) < low_2(v) < \cdots < low_k(v)$. Notice that the existence of $low_k(v)$ implies that there is a back-edge $(x, low_k(v))$, where x is a descendant of v.

▶ **Proposition 1** ([6]). *Let T be a DFS-tree of a simple graph G, and assume that the adjacency list of every vertex of G is sorted in increasing order w.r.t. the DFS numbering. Suppose also that, for some $k \in \{0, \ldots, n-1\}$, we have computed the $low_1, \ldots, low_k$ points of all vertices (w.r.t. T), and the set $\{low_1(v), \ldots, low_k(v)\}$ is stored in an increasingly sorted array for every $v \neq r$. Then we can compute the low_{k+1} points of all vertices in $O(n \log(k+1))$ time.*[3]

▶ **Corollary 2** ([6]). *For any $k \in \{1, \ldots, n-1\}$, the $low_1, \ldots, low_k$ points of all vertices can be computed in $O(m + kn \log k)$ time.*

[3] We make the convention that $\log(1) = 1$, so that the time to compute the low_1 points is $O(n)$.

3 The algorithm for vertex failures

3.1 Initializing the data structure

We will need the following ingredients in order to be able to handle at most $d_\star$ failed vertices.

- **(i)** A DFS-tree T of G rooted at a vertex r. The values ND and $depth$ (w.r.t. T) must be computed for all vertices. We identify the vertices of G with the DFS numbering of T.
- **(ii)** A level-ancestor data structure on T.
- **(iii)** A 2D-range-emptiness data structure on the set of the back-edges of G w.r.t. T.
- **(iv)** The low_i points of all vertices, for every $i \in \{1, \ldots, d_\star\}$.
- **(v)** For every $i \in \{1, \ldots, d_\star\}$, a DFS-tree T_i of T rooted at r, where the adjacency lists of the vertices are given by their children lists sorted in increasing order w.r.t. the low_i point.
- **(vi)** For every $i \in \{1, \ldots, d_\star\}$, a 2D-range-emptiness data structure on the set of the back-edges of G w.r.t. T_i.

The $depth$ value in (i) refers to the depths of the vertices in T. This is defined for every vertex v as the size of the tree path $T[r, v]$. (Thus, e.g., $depth(r) = 1$.) It takes $O(n)$ additional time to compute the $depth$ values during the DFS.

The level-ancestor data structure in (ii) is used in order to answer queries of the form $\texttt{QueryLA}(v, \delta) \equiv$ "return the ancestor of v that lies at depth δ". We use those queries in order to find the children of vertices that are ancestors of other vertices. (I.e., given that u is a descendant of v, we want to know the child of v that is an ancestor of u.) For our purposes, it is sufficient to use the solution in Section 3 of [1], that preprocesses T in $O(n \log n)$ time so that it can answer level-ancestor queries in (worst-case) $O(1)$ time.

The 2D-range-emptiness data structure in (iii) is used in order to answer queries of the form $\texttt{2D\_range}([X_1, X_2] \times [Y_1, Y_2]) \equiv$ "is there a back-edge (x, y) with $x \in [X_1, X_2]$ and $y \in [Y_1, Y_2]$?".[4] We can use a standard implementation for this data structure, that has $O(m \log n)$ space and preprocessing time complexity, and can answer a query in (worst-case) $O(\log n)$ time (see, e.g., Section 5.6 in [3]). The m factor here is unavoidable, because the number of back-edges can be as large as $m - n + 1$. However, we note that we can improve the $\log n$ factor in the space and the query time if we use a more sophisticated solution, such as [2].

The $low_1, \ldots, low_{d_\star}$ points of all vertices can be computed in $O(m + d_\star n \log d_\star) = O(m + d_\star n \log n)$ time (Corollary 2). We obviously need $O(d_\star n)$ space to store them.

For (v), we just perform $d_\star$ DFS's on T, starting from r, where each time we use a different arrangement of the children lists of T as adjacency lists. This takes $O(d_\star n)$ time in total, but we do not need to actually store the trees. (In fact, the parent pointer is the same for all of them.) What we actually need here is the DFS numbering of the i-th DFS traversal, for every $i \in \{1, \ldots, d_\star\}$, which we denote as DFS_i. We keep those DFS numberings stored, and so we need $O(d_\star n)$ additional space. The usefulness of performing all those DFS's will become clear in Section 3.4. Right now, we only need to mention that, for every $i \in \{1, \ldots, d_\star\}$, the ancestry relation in T_i is the same as that in T. Thus, the $low_1, \ldots, low_{d_\star}$ points for all vertices w.r.t. T_i are the same as those w.r.t. T.

The 2D-range-emptiness data structures in (vi) are used in order to answer queries of the form $\texttt{2D\_range\_i}([X_1, X_2] \times [Y_1, Y_2]) \equiv$ "is there a back-edge (x, y) with $x \in [X_1, X_2]$ and $y \in [Y_1, Y_2]$?", where the endpoints of the query rectangle refer to the DFS_i numbering,

[4] The input to `2D_range` is just the endpoints X_1, X_2, Y_1, Y_2 of the query rectangle; we use brackets around them, and the symbol $\times$, just for readability.

for $i \in \{1, \ldots, d_\star\}$. Since the ancestry relation is the same for T_i and T, we have that the queries `2D_range`$([X_1, X_2] \times [Y_1, Y_2])$ and `2D_range_i`$([X_1, X_2]_i \times [Y_1, Y_2]_i)$ are equivalent, where the i index below the brackets means that we have translated the endpoints in the DFS_i numbering.

The construction of the 2D-range-emptiness data structures w.r.t. the DFS-trees $T_1, \ldots, T_{d_\star}$ takes $O(d_\star m \log n)$ time in total. In order to keed those data structures stored, we need $O(d_\star m \log n)$ space. Thus, the construction and the storage of the 2D-range-emptiness data structures dominate the space-time complexity overall.

It is easy to see that the list of data structures from (i) to (vi) is flexible w.r.t. $d_\star$. Thus, if $d_\star$ increases by 1, then we need to additionally compute the $low_{d_\star+1}$ points of all vertices, the $T_{d_\star+1}$ DFS-tree, and the corresponding 2D-range-emptiness data structure. Computing the $low_{d_\star+1}$ points takes $O(n \log(d_\star + 1)) = O(n \log n)$ time, and demands an additional $O(n)$ space, assuming that we have sorted the adjacency lists of G in increasing order, and that we have stored the $low_1, \ldots, low_{d_\star}$ points, for every vertex, in an increasingly sorted array (see Proposition 1).

▶ **Corollary 3.** *Suppose that we have initialized our data structure for some $d_\star$, and we want to get a data structure for $d_\star + k$. Then we can achieve this in $O(km \log n)$ time, using extra $O(km \log n)$ space.*

If $d_\star$ decreases by k, then we just have to discard the $low_{d_\star-k+1}, \ldots, low_{d_\star}$ points, the $T_{d_\star-k+1}, \ldots, T_{d_\star}$ DFS-trees, and the corresponding 2D-range-emptiness data structures. This will free $O(km \log n)$ space.

3.2 The general idea

Let F be a set of failed vertices. Then $T \setminus F$ may consist of several connected components, all of which are subtrees of T. It will be necessary to distinguish two types of connected components of $T \setminus F$. Let C be a connected component of $T \setminus F$. If no vertex in F is a descendant of C, then C is called a *hanging subtree* of $T \setminus F$. Otherwise, C is called an *internal component* of $T \setminus F$. (See Figure 1 for an illustration.) Observe that, while the number of connected components of $T \setminus F$ may be as large as $n - 1$ (even if $|F| = 1$), the number of internal components of $T \setminus F$ is at most $|F|$. This is an important observation, that allows us to reduce the connectivity of $G \setminus F$ to the connectivity of the internal components.

More precisely, we can already provide a high level description of our strategy for answering connectivity queries between pairs of vertices. Let x, y be two vertices of $G \setminus F$. Suppose first that x belongs to an internal component C_1 and y belongs to an internal component C_2. Then it is sufficient to know whether C_1 and C_2 are connected in $G \setminus F$. Otherwise, if either x or y lies in a hanging subtree C, then we can substitute C with any internal component that is connected with C in $G \setminus F$. If no such internal component exists, then x and y are connected in $G \setminus F$ if and only if they lie in the same hanging subtree.

Thus, after the deletion of F from G, it is sufficient to make provisions so as to be able to efficiently answer the following:

(1) Given a vertex x, determine the connected component of $T \setminus F$ that contains x.

(2) Given two internal components C_1 and C_2 of $T \setminus F$, determine whether C_1 and C_2 are connected in $G \setminus F$.

(3) Given a hanging subtree C of $T \setminus F$, find an internal component of $T \setminus F$ that is connected with C in $G \setminus F$, or report that no such internal component exists.

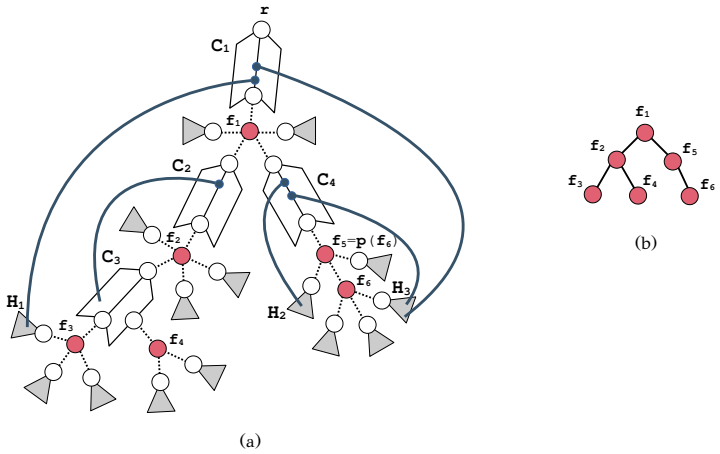

Figure 1 (a) A set of failed vertices $F = \{f_1, \ldots, f_6\}$ on a DFS-tree T, and (b) the corresponding F-forest, which shows the $parent_F$ relation between failed vertices. Notice that $T \setminus F$ is split into several connected components, but there are only four internal components, C_1, C_2, C_3 and C_4. The hanging subtrees of $T \setminus F$ are shown with gray color (e.g., H_1, H_2 and H_3). The internal components C_2 and C_3 remain connected in $G \setminus F$ through a back-edge that connects them directly. C_1 and C_4 remain connected through the hanging subtree H_3 of f_6. We have $\partial(C_1) = \{f_1\}$, $\partial(C_2) = \{f_2\}$, $\partial(C_3) = \{f_3, f_4\}$ and $\partial(C_4) = \{f_5\}$. Notice that f_6 is the only failed vertex that is not a boundary vertex of an internal component, and it has $parent_F(f_6) = p(f_6)$.

Actually, the most difficult task, and the only one that we provide a preprocessing for (during the update phase), is (2). We explain how to perform (1) and (3) during the process of answering a query, in Section 3.5. An efficient solution for (2) is provided in Section 3.4.

The general idea is that, since there are at most $d = |F|$ internal components of $T \setminus F$, we can construct a graph with $O(d)$ nodes, representing the internal components of $T \setminus F$, that captures the connectivity relation among them in $G \setminus F$ (see Lemma 10). This is basically done with the introduction of some artificial edges between the (representatives of the) internal components. In the following subsection, we state some lemmas concerning the structure of the internal components, and their connectivity relationship in $G \setminus F$. All omitted proofs are contained in the full version of our paper [6].

3.3 The structure of the internal components

We will use the roots of the connected components of $T \setminus F$ (viewed as rooted subtrees of T) as representantives of them. Now we introduce some terminology and notation. If C is a connected component of $T \setminus F$, we denote its root as r_C. If C is a hanging subtree of $T \setminus F$, then $p(r_C) = f$ is a failed vertex, and we say that C is *a hanging subtree of* f. If C, C' are two distinct connected components of $T \setminus F$ such that $r_{C'}$ is an ancestor of r_C, then we say that C' is an ancestor of C. Furthermore, if v is a vertex not in C such that v is an ancestor (resp., a descendant) of r_C, then we say that v is an ancestor (resp., a descendant) of C. If C is an internal component of $T \setminus F$ and f is a failed vertex such that $p(f) \in C$, then we say that f is a boundary vertex of C. The collection of all boundary vertices of C is denoted as $\partial(C)$. Notice that any vertex $b \in \partial(C)$ has the property that there is no failed vertex on the tree path $T[p(b), r_C]$. Conversely, a failed vertex b such that there is no failed vertex on the tree path $T[p(b), r_C]$ is a boundary vertex of C. Thus, if $b_1, \ldots, b_k$ is the collection of the boundary vertices of C, then $C = T(r_C) \setminus (T(b_1) \cup \cdots \cup T(b_k))$.

The following lemma is a set of properties that are satisfied by the internal components.

▶ **Lemma 4** ([6]). *Let C be an internal component of $T \setminus F$. Then:*

(1) *Either $r_C = r$, or $p(r_C) \in F$.*

(2) *For every vertex v that is a descendant of C, there is a unique boundary vertex of C that is an ancestor of v.*

(3) *Let $f_1, \ldots, f_k$ be the boundary vertices of C, sorted in increasing order. Then C is the union of the following subsets of consecutive vertices: $[r_C, f_1 - 1], [f_1 + ND(f_1), f_2 - 1], \ldots, [f_{k-1} + ND(f_{k-1}), f_k - 1], [f_k + ND(f_k), r_C + ND(r_C) - 1]$. (We note that some of those sets may be empty.)*

We represent the ancestry relation between failed vertices using a forest which we call the *failed vertex forest* (*F-forest*, for short). The F-forest consists of the following two elements. First, for every failed vertex f, there is a pointer $parent_F(f)$ to the nearest ancestor of f (in T) that is also a failed vertex. If there is no ancestor of f that is a failed vertex, then we let $parent_F(f) = \perp$. And second, every failed vertex f has a pointer to its list of children in the F-forest.

The F-forest can be easily constructed in $O(d^2)$ time: we just have to find, for every failed vertex f, the maximum failed vertex f' that is a proper ancestor of f; then we set $parent_F(f) = f'$, and we append f to the list of the children of f' in the F-forest.

The next lemma shows how we can check in constant time whether a failed vertex belongs to the boundary of an internal component, and how to retrieve the root of this component.

▶ **Lemma 5** ([6]). *A failed vertex f is a boundary vertex of an internal component if and only if $parent_F(f) \neq p(f)$. Now let f be a boundary vertex of an internal component C. Then, if $parent_F(f)$ exists, we have that the root of C is the child of $parent_F(f)$ that is an ancestor of f. Otherwise, the root of C is r.*

Thus, according to Lemma 5, if f is a boundary vertex of an internal component C with $r_C \neq r$, we can retrieve r_C in constant time using a level-ancestor query: i.e., we ask for the ancestor of f (in T) whose depth equals that of $parent_F(f) + 1$. We may use this fact throughout without mention.

The following lemma shows that there are two types of edges that determine the connectivity relation in $G \setminus F$ between the connected components of $T \setminus F$.

▶ **Lemma 6** ([6]). *Let e be an edge of $G \setminus F$ whose endpoints lie in different connected components of $T \setminus F$. Then e is a back-edge and either (i) both endpoints of e lie in internal components, or (ii) one endpoint of e lies in a hanging subtree H, and the other endpoint lies in an internal component C that is an ancestor of H.*

▶ **Corollary 7** ([6]). *Let C, C' be two distinct connected components of $T \setminus F$ that are connected with an edge e of $G \setminus F$. Assume w.l.o.g. that $r_{C'} < r_C$. Then C' is an ancestor of C.*

The following lemma provides an algorithmically useful criterion to determine whether a connected component of $T \setminus F$ – a hanging subtree or an internal component – is connected with an internal component of $T \setminus F$ through a back-edge.

▶ **Lemma 8** ([6]). *Let C, C' be two connected components of $T \setminus F$ such that C' is an internal component that is an ancestor of C, and let b be the boundary vertex of C' that is an ancestor of C. Then there is a back-edge from C to C' if and only if there is a back-edge from C whose lower end lies in $[r_{C'}, p(b)]$.*

▶ **Definition 9** (Connectivity graph). *Let $\mathcal{R}$ be a multigraph where $V(\mathcal{R})$ is the set of the roots of the internal components of $T \setminus F$, and $E(\mathcal{R})$ satisfies the following three properties:*

(1) *For every back-edge connecting two internal components C and C', there is an edge $(r_C, r_{C'})$ in $\mathcal{R}$.*
(2) *Let H be a hanging subtree of a failed vertex f, and let $C_1, \ldots, C_k$ be the internal components that are connected with H through a back-edge. (By Lemma 6, all of $C_1, \ldots, C_k$ are ancestors of H.) Assume w.l.o.g. that C_k is an ancestor of all $C_1, \ldots, C_{k-1}$. Then $\mathcal{R}$ contains the edges $(r_{C_1}, r_{C_k}), (r_{C_2}, r_{C_k}), \ldots, (r_{C_{k-1}}, r_{C_k})$.*
(3) *Every edge of $\mathcal{R}$ is given by either (1) or (2), or it is an edge of the form $(r_C, r_{C'})$, where C, C' are two internal components that are connected in $G \setminus F$.*

Then $\mathcal{R}$ is called a connectivity graph of the internal components *of $T \setminus F$. The edges of (1) and (2) are called Type-1 and Type-2, respectively.*

The following lemma shows that a connectivity graph captures the connectivity relationship of the internal components of $T \setminus F$ in $G \setminus F$.

▶ **Lemma 10** ([6]). *Let $\mathcal{R}$ be a connectivity graph of the internal components of $T \setminus F$. Then, two internal components C, C' of $T \setminus F$ are connected in $G \setminus F$ if and only if $r_C, r_{C'}$ are connected in $\mathcal{R}$.*

3.4 Handling the updates: construction of a connectivity graph for the internal components of $T \setminus F$

Given a set of failed vertices F, with $|F| = d \leq d_\star$, we will show how we can construct a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$, using $O(d^4)$ calls to 2D-range-emptiness queries. Recall that $V(\mathcal{R})$ is the set of the roots of the internal components of $T \setminus F$.

Algorithm 1 shows how we can find all Type-1 edges of $\mathcal{R}$. The idea is basically to perform 2D-range-emptiness queries for every pair of internal components, in order to determine the existence of a back-edge that connects them. More precisely, we work as follows. Let C be an internal component of $T \setminus F$. Then it is sufficient to check every ancestor component C' of C, in order to determine whether there is a back-edge from C to C' (see Corollary 7). Let $f_1, \ldots, f_k$ be the boundary vertices of C, sorted in increasing order. Let also f' be the boundary vertex of C' that is an ancestor of C, and let $I = [r_{C'}, p(f')]$. Then we perform 2D-range-emptiness queries for the existence of a back-edge on the rectangles $[r_C, f_1 - 1] \times I, [f_1 + ND(f_1), f_2 - 1] \times I, \ldots, [f_k + ND(f_k), r_C + ND(r_C) - 1] \times I$. We know that there is a back-edge connecting C and C' if and only if at least one of those queries is positive (see Lemma 4(3) and Lemma 8). If that is the case, then we add the edge $(r_C, r_{C'})$ to $\mathcal{R}$.

Observe that the total number of 2D-range-emptiness queries that we perform is $O(d^2)$, because every one of them corresponds to a triple (C, f, C'), where C, C' are internal components, C' is an ancestor of C, and f is a boundary vertex of C, or r_C. And if $C_1, \ldots, C_k$ are all the internal components of $T \setminus F$, then the number of those triples is bounded by $(|\partial(C_1)| + 1) \cdot d + \cdots + (|\partial(C_k)| + 1) \cdot d = (|\partial(C_1)| + \cdots + |\partial(C_k)| + k) \cdot d \leq (d + k) \cdot d \leq (d + d) \cdot d = O(d^2)$.

▶ **Proposition 11** ([6]). *Algorithm 1 correctly computes all Type-1 edges to construct a connectivity graph for the internal components of $T \setminus F$. The running time of this algorithm is $O(d^2 \log n)$.*

Algorithm 1 Compute all Type-1 edges to construct a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$.

```
1  foreach internal component C of T \ F do
2      let f_1, ..., f_k be the boundary vertices of C, sorted in increasing order
3      // process every internal component C' that is an ancestor of C
4      set f' ← p(r_C)
5      while f' ≠ ⊥ do
6          if p(f') ≠ parent_F(f') then
7              let C' be the internal component of T \ F with f' ∈ ∂(C')
8              set I ← [r_{C'}, p(f')]
9              if at least one of the following queries is positive:
10             2D_range([r_C, f_1 − 1] × I)
11             2D_range([f_1 + ND(f_1), f_2 − 1] × I)
12             ...
13             2D_range([f_{k−1} + ND(f_{k−1}), f_k − 1] × I)
14             2D_range([f_k + ND(f_k), r_C + ND(r_C) − 1] × I) then
15                 add the Type-1 edge (r_C, r_{C'}) to R
16             end
17         end
18         f' ← parent_F(f')
19     end
20 end
```

The construction of Type-2 edges is not so straightforward. For every failed vertex f, and every two internal components C and C', such that C is an ancestor of f and C' is an ancestor of C, we would like to know whether there is a hanging subtree of f, from which stem a back-edge e with an endpoint in C and a back-edge e' with an endpoint in C'. The straightforward way to determine this is the following. Let b (resp., b') be the boundary vertex of C (resp., C') that is an ancestor of f. Then, for every hanging subtree of f with root c, we perform 2D-range-emptiness queries on the rectangles $[c, c + ND(c) - 1] \times [r_C, p(b)]$ and $[c, c + ND(c) - 1] \times [r_{C'}, p(b')]$. If both queries are positive, then we know that C and C' are connected in $G \setminus F$ through the hanging subtree with root c.

Obviously, this method is not efficient in general, because the number of hanging subtrees of f can be very close to n. However, it is the basis for our more efficient method. The idea is to perform a lot of those queries at once, for large batches of hanging subtrees. More specifically, we perform the queries on *consecutive* hanging subtrees of f (i.e., their roots are consecutive children of f), for which we know that the answer is positive on C' (i.e., for every one of those subtrees, there certainly exists a back-edge that connects it with C'). In order for this idea to work, we have to rearrange properly the lists of children of all vertices. (Otherwise, the hanging subtrees of f that are connected with C' through a back-edge may not be consecutive in the list of children of f.) In effect, we maintain several DFS trees (specifically: $d_\star$), and several 2D-range-emptiness data structures, one for every different arrangement of the children lists.

Let us elaborate on this idea. Let H be a hanging subtree of f that connects some internal components, and let C' be the lowest one among them (i.e., the one that is an ancestor of all the others). Then we have that the lower ends of all back-edges that stem from H and end in ancestors of C' are failed vertices that are ancestors of C'. Thus, since there are at

most d failed vertices in total, we have that at least one among $low_1(r_H), \ldots, low_d(r_H)$ is in C'. In other words, r_H is one of the children of f whose low_i point is in C', for some $i \in \{1, \ldots, d\}$. Now, assume that for every $i \in \{1, \ldots, d_\star\}$, we have a copy of the list of the children of f sorted in increasing order w.r.t. the low_i point; let us call this list $L_i(f)$, and let it be stored in way that allows for binary search w.r.t. the low_i point. Then, for every internal component C that is an ancestor of f, we can find the segment $S_i(C)$ of $L_i(f)$ that consists of the children of f whose low_i point lies in C, by searching for the leftmost and the rightmost child in $L_i(f)$ whose low_i point lies in $[r_C, p(b)]$, where b is the boundary vertex of C that is an ancestor of f.

Now let $i \in \{1, \ldots, d\}$ be such that $low_i(r_H) \in C'$. Then we have that $r_H \in S_i(C')$. Furthermore, we have that every child of f that lies in $S_i(C')$ and is the root of a hanging subtree H' of f has the property that H' is also connected with C' through a back-edge. Thus, we would like to be able to perform 2D-range-emptiness queries as above on the subset S of $S_i(C')$ that consists of roots of hanging subtrees, in order to determine the connectivity (in $G \setminus F$) of C' with all internal components C that are ancestors of f and descendants of C'. We could do this efficiently if we had the guarantee that S consists of large segments of consecutive children of f. We can accommodate for that during the preprocessing phase: for every $i \in \{1, \ldots, d_\star\}$, we perform a DFS of T, starting from r, where the adjacency list of every vertex v is given by $L_i(v)$.[5] Let T_i be the resulting DFS tree, and let DFS_i be the corresponding DFS numbering. Then, with the DFS numbering of T_i, we initialize a data structure *2D_range_i*, for answering 2D-range-emptiness queries for back-edges w.r.t. T_i in subrectangles of $[1, n] \times [1, n]$.

Now let us see how everything is put together. Let H be a hanging subtree of f that connects two internal components C_1 and C_2, and let b_1 and b_2 be the boundary vertices of C_1 and C_2, respectively, that are ancestors of f. Let C' be the lowest internal component that is connected through a back-edge with H. Then there is an $i \in \{1, \ldots, d\}$ such that $low_i(r_H) \in C'$. Let S be the maximal segment of $S_i(C')$ that contains r_H and consists of roots of hanging subtrees, let L be the minimum of S and let R be the maximum of S.[6] Then the 2D-range-emptiness queries on $[L, R + ND(R) - 1]_i \times [r_{C_1}, p(b_1)]_i$ and $[L, R + ND(R) - 1]_i \times [r_{C_2}, p(b_2)]_i$ with *2D_range_i* are both positive, and so we will add the edges $(r_{C_1}, r_{C'})$ and $(r_{C_2}, r_{C'})$ to $\mathcal{R}$. This will maintain in $\mathcal{R}$ the information that C', C_1 and C_2, are connected with the same hanging subtree of f.

The algorithm that constructs enough Type-2 edges to make $\mathcal{R}$ a connectivity graph of the internal components of $T \setminus F$ is given in Algorithm 2. Our result is stated in Proposition 12.

▶ **Proposition 12** ([6]). *Algorithm 2 computes enough Type-2 edges to construct a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$ (supposing that $\mathcal{R}$ contains all Type-1 edges). The running time of this algorithm is $O(d^4 \log n)$.*

3.5 Answering the queries

Assume that we have constructed a connectivity graph $\mathcal{R}$ for the internal components of $T \setminus F$, and that we have computed its connected components. Thus, given two internal components C and C', we can determine in constant time whether C and C' are connected in $G \setminus F$, by simply checking whether r_C and $r_{C'}$ are in the same connected component of $\mathcal{R}$ (see Lemma 10).

[5] I.e., it is necessary that the vertices in the adjacency list of v appear in the same order as in $L_i(v)$.
[6] Notice that, due to the construction of T_i, we have that $DFS_i(L)$ and $DFS_i(R)$ are also the minimum and the maximum, respectively, of $DFS_i(S)$.

Algorithm 2 Compute enough Type-2 edges to construct a connectivity graph for the internal components of $T \setminus F$.

```
1  foreach failed vertex f do
2      // process all pairs of internal components that are ancestors of f
3      set f' ← parent_F(f)
4      while f' ≠ ⊥ do
5          let C' be the internal component with f' ∈ ∂(C')
6          // skip the following if C' does not exist, and go immediately
               to Line 26
7          foreach i ∈ {1, ..., d} do
8              let S_i be the collection of all maximal segments of L_i(f) that consist of
                roots of hanging subtrees with their low_i point in C'
9          end
10         // process all internal components C that are ancestors of f
               and descendants of C'
11         set f'' ← f
12         while f'' ≠ f' do
13             let C be the internal component with f'' ∈ ∂(C)
14             // skip the following if C does not exist, and go
                   immediately to Line 24
15             // check if C is connected with C' through at least one
                   hanging subtree of f
16             foreach i ∈ {1, ..., d} do
17                 foreach S ∈ S_i do
18                     let L ← min(S) and R ← max(S)
19                     if 2D_range_i([L, R + ND(R) − 1]_i × [r_C, p(f'')]_i) = true then
20                         add the Type-2 edge (r_C, r_C') to R
21                     end
22                 end
23             end
24             f'' ← parent_F(f'')
25         end
26         f' ← parent_F(f')
27     end
28 end
```

Now let x, y be two vertices in $V(G) \setminus F$. In order to determine whether x, y are connected in $G \setminus F$, we try to substitute x, y with roots of internal components of $T \setminus F$, and then we reduce the query to those roots. Specifically, if x (resp., y) belongs to an internal component C of $T \setminus F$, then the connectivity between x and y is the same as that between r_C and y (resp., x and r_C). Otherwise, if x (resp., y) belongs to a hanging subtree H of $T \setminus F$, then we try to find an internal component that is connected with H through a back-edge. If such an internal component C exists, then we can substitute x (resp., y) with r_C. Otherwise, x, y are connected in $G \setminus F$ if and only if they belong to the same hanging subtree of $T \setminus F$. This idea is shown in Algorithm 3.

▶ **Proposition 13** ([6]). *Given two vertices x, y in $V(G) \setminus F$, Algorithm 3 correctly determines whether x, y are connected in $G \setminus F$. The running time of Algorithm 3 is $O(d)$.*

Algorithm 3 query(x, y).

```
1  if x lies in an internal component C and y lies in an internal component C' then
2      if r_C is connected with r_C' in R then return true
3      return false
4  end
5  // at least one of x, y lies in a hanging subtree
6  if x lies in a hanging subtree H then
7      // check whether H is connected with an internal component through
           a back-edge
8      for i ∈ {1, ..., d} do
9          if low_i(r_H) ≠ ⊥ and low_i(r_H) ∉ F then
10             return query(low_i(r_H), y)
11         end
12     end
13     // there is no internal component that is connected with H in G \ F
14     if y lies in H then return true
15     return false
16 end
17 return query(y, x)
```

References

1. Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.
2. Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry,*, pages 1–10, 2011. doi:10.1145/1998196.1998198.
3. Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008. URL: https://www.worldcat.org/oclc/227584184.
4. Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.
5. Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. *SIAM J. Comput.*, 49(6):1363–1396, 2020. doi:10.1137/17M1146610.
6. Evangelos Kosinas. Connectivity queries under vertex failures: Not optimal, but practical. *arXiv version*, 2023. arXiv:2305.01756.
7. Yaowei Long and Thatchaphol Saranurak. Near-optimal deterministic vertex-failure connectivity oracles. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 1002–1010, 2022. doi:10.1109/FOCS54457.2022.00098.
8. Hiroshi Nagamochi and Toshihide 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. doi:10.1007/BF01758778.
9. Michal Pilipczuk, Nicole Schirrmacher, Sebastian Siebertz, Szymon Torunczyk, and Alexandre Vigny. Algorithms and data structures for first-order logic with connectivity under vertex failures. In *49th International Colloquium on Automata, Languages, and Programming, ICALP*, volume 229 of *LIPIcs*, pages 102:1–102:18, 2022. doi:10.4230/LIPIcs.ICALP.2022.102.
10. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.