An Optimal 3-Fault-Tolerant Connectivity Oracle

Evangelos Kosinas
University of Ioannina, Greece
Archimedes, Athena RC, Greece

Abstract

We present an optimal oracle for answering connectivity queries in undirected graphs in the presence of at most three vertex failures. Specifically, we show that we can process a graph G in $O(n+m)$ time, in order to build a data structure that occupies $O(n)$ space, which can be used in order to answer queries of the form *"given a set F of at most three vertices, and two vertices x and y not in F, are x and y connected in $G \setminus F$?"* in constant time, where n and m denote the number of vertices and edges, respectively, of G. The idea is to rely on the DFS-based framework introduced by Kosinas [ESA'23], for handling connectivity queries in the presence of multiple vertex failures. Our technical contribution is to show how to appropriately extend the toolkit of the DFS-based parameters, in order to optimally handle up to three vertex failures. Our approach has the interesting property that it does not rely on a compact representation of vertex cuts, and has the potential to provide optimal solutions for more vertex failures. Furthermore, we show that the DFS-based framework can be easily extended in order to answer vertex-cut queries, and the number of connected components in the presence of multiple vertex failures. In the case of three vertex failures, we can answer such queries in $O(\log n)$ time.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; 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.ICALP.2025.110

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2504.17937

Funding This work has been partially supported by project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the European Union under the NextGenerationEU Program.

1 Introduction

1.1 Problem definition

This paper is motivated by the following problem. Let G be an undirected graph, and let $d_\star$ be a positive integer. The goal is to efficiently build a data structure, that occupies as little space as possible, so that we can efficiently answer queries of the following form:

"*Given a set of vertices F, with $|F| \leq d_\star$, that have failed to work, and two vertices x and y not in F, are x and y connected in $G \setminus F$?*".

This problem has received significant attention in the last few years [11, 40, 31, 28, 32], and it can be characterized as an instance of *"emergency planning"* [37], or as a data structure problem in the *fault-tolerant*, or *sensitivity* setting. The potential usefulness of such data structures in real-world applications is obvious: since networks in real life are prone to failures, it may be worthwhile to spend some time in order to preprocess the graph, so that we can deal efficiently with (temporary) malfunctions.

What makes this particular problem so interesting, and justifies the variety of the approaches that have been taken so far, is the difficulty in simultaneously optimizing all the parameters of efficiency: i.e., the preprocessing time, the space usage, the time to answer the queries, and (in some cases) the time to separately handle the set F of failed vertices. Thus, the solutions that exist in the literature provide various trade-offs, and none of them outcompetes the others in every measure of efficiency (see Table 1).

However, for the cases where $d_\star \in \{1, 2\}$, one can provide an optimal solution by relying on some older works. Specifically, for $d_\star = 1$, one can use a simple DFS-based approach [41], and for $d_\star = 2$ one can use the SPQR trees [2, 21, 18], that represent compactly all 2-vertex cuts of a biconnected graph. These cases admit of an *optimal* solution in the sense that the data structures can be constructed in $O(n + m)$ time, they occupy $O(n)$ space, and they can answer any connectivity query in $O(1)$ time, where n and m denote the number of vertices and edges, respectively, of G. That these preprocessing and query times are optimal is obvious. The claim of the optimality of the $O(n)$ space usage is justified by the $\Omega(\min\{m, d_\star n\})$ lower bound on the bits of space usage that are provably necessary for some classes of graphs [11].

Two related kinds of queries, that do not involve a particular pair of vertices, but enquire for the global impact of the failed vertices on the connectivity of G, are:

1. *"Is $G \setminus F$ connected?"* (vertex-cut query)
2. *"What is the number of connected components of $G \setminus F$?"*

A conditional lower bound for vertex-cut queries was given in [31], and only very recently [24] have provided an efficient solution for them. The optimal oracles mentioned above for $d_\star \in \{1, 2\}$ can also answer queries 1 and 2. Specifically, a DFS-based approach can easily answer both such queries in constant time when $|F| = 1$, and an $O(n)$-time preprocessing on the SPQR-based data structure can answer at least the first kind of queries in $O(\log n)$ time (as mentioned in [24]).

In this paper, it is sufficient to assume that the input graph G is connected. This is because, if one has an oracle (for any of the problems that we have mentioned), that works for connected graphs, then one can simply initialize an instance of the oracle on every connected component of the graph, and thus have in total an oracle providing the same bounds as the original. Furthermore, we can use the sparsifier of Nagamochi and Ibaraki [33], in order to replace every "m" in the bounds that we state with "$\min\{m, d_\star n\}$" (with the exception of the preprocessing time, in which "m" must necessarily appear as an additive factor).

1.2 Our contribution

Our main contribution is an optimal oracle for the case $d_\star = 3$. This can be stated as the following:

▶ **Theorem 1.** *We can process an undirected graph G with n vertices and m edges in $O(n+m)$ time, in order to build a data structure with $O(n)$ size, which can answer queries of the form "given a set F of at most three vertices, and two vertices x and y not in F, are x and y connected in $G \setminus F$?" in constant time.*

We note that Theorem 1 constitutes a genuine theoretical contribution, since the existence of an optimal oracle for $d_\star > 2$ was an open problem prior to our work (see our discussion in Section 1.3). Furthermore, the solution that we provide has the following notable characteristics:

(1) It encompasses the cases $d_\star \in \{1, 2\}$, and has the potential to be extended to an optimal solution for $d_\star > 3$.

(2) The underlying data structure does not rely on a computation or compact representation of vertex cuts. Instead, we work directly on a DFS tree, and this enables us to report efficiently the number of connected components upon removal of a set of vertices.
(3) The most complicated data structures that we use are: the optimal DSU data structure of Gabow and Tarjan [15] (that works on a predetermined tree of unions) on the RAM model of computation, an optimal level-ancestors query data structure [3], an optimal RMQ data structure (e.g., [13]), and an optimal oracle for NCA queries (e.g., [4]). Thus, in practice one could use alternative (but still very efficient) implementations of those data structures, in order to have a working oracle operating in near-optimal efficiency.

In order to establish Theorem 1, we rely on the DFS-based framework introduced by Kosinas [28], for designing an oracle for any fixed $d_\star$. The general idea in [28] is to use a DFS tree T of the graph, and determine the connectivity relation in $G \setminus F$ between some specific subtrees of $T \setminus F$, which are called *internal components* (see Section 2.2). This connectivity relation is sufficient in order to be able to answer efficiently any connectivity query. The input $d_\star$ for constructing the oracle is vital in order to compute a set of DFS-based parameters that are used in order to handle up to $d_\star$ failures. What differs for us in the case $d_\star = 3$, is that we choose an alternative set of DFS-based parameters (coupled with, and motivated by, a much more involved case analysis for the internal components), which can be computed in linear time, occupy $O(n)$ space, and, if handled carefully, can be used in order to establish the connectivity between the internal components in constant time. (After that, every connectivity query on $G \setminus F$ can be handled precisely as in [28].)

One of the main attributes of the oracle from [28], that makes it particularly attractive for our purposes, is its simplicity, both in itself, and in relation to all the other approaches that have been taken so far for general $d_\star$. In fact, we show that we can easily extend the underlying data structure, so that it can answer efficiently both vertex-cut queries, and queries for the number of connected components of $G \setminus F$. Specifically, we have the following (see Section 2.3):

▶ **Theorem 2.** *Let G be an undirected graph with n vertices and m edges, and let $d_\star$ be a positive integer. We can process G in $O(d_\star m \log n)$ time, in order to build a data structure with size $O(d_\star m \log n)$, which can answer queries of the form "given a set F of at most $d_\star$ vertices, what is the number of the connected components of $G \setminus F$?", in $O((d^4 + 2^d d^2) \log n)$ time, where $d = |F|$.*

Notice that the oracle in Theorem 2 can also answer the corresponding vertex-cut query within the same time bound, because, if we are given a set of vertices F, then $G \setminus F$ is connected if and only if $G \setminus F$ consists of only one connected component. (Recall our convention that G is connected.)

Our method for augmenting the data structure from [28] so that we can provide Theorem 2, implies the following improved bound in the case where G is $d_\star$-connected (i.e., in the case where G is so "well-connected", that we have to remove at least $d_\star$ vertices in order to disconnect it; see Section 2.3):

▶ **Corollary 3.** *Suppose that G is $d_\star$-connected. Then, the oracle described in Theorem 2 can answer queries of the form "given a set F with $d_\star$ vertices, what is the number of the connected components of $G \setminus F$?", in $O(d_\star^4 \log n)$ time.*

Again, notice that the oracle in Corollary 3 can also answer the corresponding vertex-cut query within the same time bound. We note that the problem of efficiently answering $d_\star$-vertex-cut queries for $d_\star$-connected graphs was left as an open problem in [39], and it was only very recently addressed in [24]. (For a comparison between our bounds and that of [24], see Table 3 and the discussion below the table.) Furthermore, in the case where G is κ-connected, for some $\kappa \leq d_\star$, we can still have an improved time bound for answering the queries, but this is a little bit more involved:

▶ **Corollary 4.** *Suppose that G is κ-connected. Then, the oracle described in Theorem 2 can answer queries of the form "given a set F of vertices, with $|F| = d$ and $\kappa \leq d \leq d_\star$, what is the number of connected components of $G \setminus F$?", in $O(d^4 \log n + (d-\kappa+1)(\binom{d-1}{\kappa-1}(\kappa-1) + \cdots + \binom{d-1}{d-1}(d-1)) \log n)$ time.*

For a proof of Corollary 4, see Section 2.3.2. To appreciate this time bound, let us suppose, for example, that G is κ-connected (for some $\kappa < d_\star$), and that F is a set of $\kappa+1$ vertices. Then, we can determine the number of connected components of $G \setminus F$ in $O(\kappa^4 \log n)$ time.

Our technique for answering the queries for the number of connected components can be adapted to the case where $d_\star = 3$, so that we have the following:

▶ **Corollary 5.** *The data structure described in Theorem 1 can answer queries of the form "given a set F of at most three vertices, what is the number of connected components of $G \setminus F$?" in $O(\log n)$ time.*

1.3 Previous work

In Table 1 we see the bounds provided by the best known solutions for general $d_\star$, and in Table 2 we see how they compare with our optimal oracle for the case $d_\star = 3$. (In Table 2 some entries are removed, because they do not provide any advantage in their bounds over the rest when $d_\star$ is a fixed constant.)

Table 1 The best known bounds for a deterministic oracle for answering connectivity queries under up to $d_\star$ vertex failures. Notice that there are various trade-offs that make every one of those solutions have an advantage over the rest in some respects. The $\widetilde{O}$ symbol hides polylogarithmic factors, and $\widehat{O}$ hides subpolynomial factors that are worse than polylogarithmic. The function $\log^* n$ that appears in the space usage of the Long and Saranurak oracle is described in their paper as one that "can be substituted with any slowly growing function". All these oracles, except the one by Pilipczuk et al., support an update phase, in which the set F of failed vertices is processed, so that all connectivity queries under the failures from F can be answered in $O(d)$ time, where $d = |F|$. Thus, in order to answer the first connectivity query, at least *Update* time must have been expended.

	Preprocessing	Space	Update	Query
Duan and Pettie [11]	$O(mn \log n)$	$O(d_\star m \log n)$	$O(d^3 \log^3 n)$	$O(d)$
Long and Saranurak [31]	$\hat{O}(m) + \tilde{O}(d_\star m)$	$O(m \log^* n)$	$\hat{O}(d^2)$	$O(d)$
Pilipczuk et al. [40]	$O(2^{2^{O(d_\star)}} mn^2)$	$O(2^{2^{O(d_\star)}} m)$	–	$O(2^{2^{O(d_\star)}})$
Kosinas [28]	$O(d_\star m \log n)$	$O(d_\star m \log n)$	$O(d^4 \log n)$	$O(d)$
Long and Wang [32]	$\hat{O}(m) + O(d_\star m \log^3 n)$	$O(m \log^3 n)$	$O(d^2 \log^7 n)$	$O(d)$

We must note that some authors (e.g., [11, 31]) claim that a data structure from [26] implies a near-optimal oracle for $d_\star = 3$. (Specifically, that the oracle occupies space $O(n)$, can answer queries in $O(1)$ time, and can be constructed in near-linear time.) However, no attempt was ever made (as far as we know) to substantiate this claim, and therefore we cannot take it for granted. The data structure from [26] was designed in order to solve the problem

Table 2 The best known bounds when $d_\star$ is a fixed constant. We get the first three entries precisely from Table 1, after removing "$d_\star$" and "d" from the bounds, and then deleting the entries that do not have any advantange (in the asymptotic complexity) over the rest. For the case $d_\star = 3$ in particular, one can substitute "m" with "n" everywhere, except in the preprocessing time, where "m" must appear as an additive factor. (Because we can consider as part of the preprocessing the sparsification of Nagamochi and Ibaraki [33], that works in time $O(m+n)$, and produces a graph with n vertices and $O(d_\star n)$ edges, which maintains the connectivity relation up to $d_\star$ failures.)

	Preprocessing	Space	Update	Query
Long and Saranurak [31]	$\hat{O}(m) + \tilde{O}(m)$	$O(m \log^* n)$	$\hat{O}(1)$	$O(1)$
Pilipczuk et al. [40]	$O(mn^2)$	$O(m)$	–	$O(1)$
Kosinas [28]	$O(m \log n)$	$O(m \log n)$	$O(\log n)$	$O(1)$
Block- and SPQR-trees [2] (for $d_\star \in \{1,2\}$)	$O(m+n)$	$O(n)$	–	$O(1)$
This paper (for $d_\star = 3$)	$O(m+n)$	$O(n)$	–	$O(1)$

of answering 4-connectivity queries. Thus, after a near-linear-time preprocessing, one can use this data structure in order to answer queries of the form *"are x and y 4-connected?"*. A 4-connectivity query for x and y asks whether x and y remain connected if *any* set of at most three vertices (and/or edges) is removed. Thus, we first notice that this solves a different problem than our own, and, in general, the two problems are relatively independent. This is because, if we know that x and y are *not* 4-connected, then this in itself tells us nothing about whether a particular set of at most three vertices disconnects x and y upon removal. So the question arises, how does the data structure from [26] determine the 4-connectivity? And the answer to that, according to [26], is that the data structure stores a collection of *some* 3-vertex cuts, which are sufficient in order to determine that relation. However, in order to solve our problem with such a data structure, one would need to have a representation of *all* 3-vertex cuts of the graph, and of how they separate the vertex set. Can the data structure from [26] support this functionality? This is not discussed in [26], and one should provide an explanation as to how this data structure can provide an oracle for answering connectivity queries under any set of at most three vertex failures. In any case, even if [26] can be made to do that, the construction time is not linear, and so our own oracle remains the first known optimal solution for $d_\star = 3$.

1.4 Concurrent work

The problem of designing an efficient oracle for vertex-cut queries was only very recently addressed in [24], although the question was posed (and conditional lower bounds were given) in [31], and a particular interest for the case of κ-connected graphs was expressed in [39]. In Table 3 we see the bounds provided by [24], and by our simple extension of the framework of [28]. We note that our own oracle answers a stronger type of queries: it reports the number of connected components after removing a set of vertices.

In order to get a better appreciation of our result, it is worth comparing it in some detail with that of [24][1]. First, as noted above, our oracle reports the number of connected components after removing a set of vertices F (and not just whether F is a vertex cut).

[1] Although the paper of [24] does not appear to have been peer-reviewed, the result is highly credible.

Table 3 Best known bounds for vertex-cut oracles. These take as input G and $d_\star$, and can answer, for every vertex set F with $|F| = d \leq d_\star$, whether $G \setminus F$ is connected. Our own oracles have the stronger property that they report the number of the connected components of $G \setminus F$. In the first line, the oracle of [24] demands that $d_\star = O(\log n)$ (and so $d_\star$ is absorbed by the $\widetilde{O}$ expression), whereas our result in the third line works for any $d_\star$. Being able to lift the restriction $d_\star = O(\log n)$ gives an advantage to us, as we discuss in the main text. The "$+\delta$" that appears in the exponent in the preprocessing time of the oracle of [24] can be any fixed $\delta > 0$, but the choice of δ influences the `polylogn` factors hidden behind the expressions for the space and the query time bounds. δ can also be replaced with $o(1)$, but this is going to introduce an $n^{o(1)}$ factor in both the space and the query time. Notice the better time bounds for the queries in both oracles when we are given as information that the graph is $d_\star$-connected. Also, notice that the time bounds for the queries depend on $|F|$, except when the graph is $d_\star$-connected, in which case the queries are non-trivial only for F with $|F| = d_\star$. Using the sparsifier of Nagamochi and Ibaraki [33], we can replace the "m" in our bounds with $\min\{m, d_\star n\}$.

	Preprocessing	Space	Query	Graph Type
Jiang et al. [24]	$O(m) + \widetilde{O}(n^{1+\delta})$	$\widetilde{O}(n)$	$\widetilde{O}(2^d)$	all
>>	$O(m) + \widetilde{O}(d_\star^2 n) + \widetilde{O}((d_\star n)^{1+\delta})$	$\widetilde{O}(d_\star n)$	$\widetilde{O}(d_\star^2)$	$d_\star$-conn
This paper	$O(d_\star m \log n)$	$O(d_\star m \log n)$	$O((d^4 + 2^d d^2) \log n)$	all
>>	$O(d_\star m \log n)$	$O(d_\star m \log n)$	$O(d_\star^4 \log n)$	$d_\star$-conn

As far as we know, we are the first to provide a non-trivial (and very efficient) oracle for this problem. Second, our bounds involve only one "$\log n$" factor, whereas the bounds from [24] hide many more. Third, although [24] provide a better query time (w.r.t. $d_\star$) for the case where the graph is $d_\star$-connected, we provide a better query time throughout the entire regime where the graph is κ-connected, for any $\kappa \leq d_\star$. (For the precise time bound, see Corollary 4.) This is a much stronger result than the one asked for by [39], and it has the interesting property that it does not rely on a computation or compact representation of vertex cuts, but it works directly on a DFS tree. Specifically, the information that the graph is κ-connected does not affect the preprocessing step (which is only determined by G and $d_\star$): it is sufficient to be given as input to the query, and it can speed up the query-answering procedure by allowing us to skip some unnecessary searches.

Finally, in both our oracle and that of [24], there appears a 2^d factor in the query time, where $d = |F|$, which seems to make both oracles useless when $d = \Omega(\log n)$. However, this is a worst-case bound for our own oracle. A single glance into the mechanics of our oracle (provided in Section 2.3) will reveal that the exponential time occurs only if there is a large subset of F that consists of vertices that are pairwise related as ancestor and descendant. Thus, if it happens that there are not large subsets of F of related vertices (w.r.t. the DFS tree), then our oracle is potentially much better than applying brute force (e.g., BFS), and thus it makes sense to initialize it for values of $d_\star$ which are larger than $\Omega(\log n)$.

1.5 Related work

1.5.1 Connectivity under edge failures

There is a similar problem for connectivity oracles under *edge failures*.[2] This problem was considered first by Patrascu and Thorup [37], where they provided an oracle with $O(m)$ space and near-optimal query time, but very high preprocessing time. Kosinas [29] provided

[2] In fact, this problem can be easily reduced to oracles for vertex failures, but such a reduction provides suboptimal solutions; by dealing directly with edge failures, one can provide more efficient oracles.

an optimal solution for up to four edge failures, using a DFS-based approach. Duan and Pettie [11] provide a very efficient oracle for any number of edge failures, with near-linear preprocessing time (in expectation). For more references on that problem, see [11].

1.5.2 Connectivity under mixed deletions/insertions

What happens if, instead of vertex failures only, we allow for intermixed failures and activations of vertices? This model was considered first by Henzinger and Neumann [20], where they provided an efficient reduction to any oracle for vertex failures. However, some data structures for vertex failures have the potential to be extended to the intermixed activations/deactivations version, so that they provide better bounds than a black-box reduction. This approach was taken by Long and Wang [32] (who extended the data structure of Long and Saranurak [31]), and by Bingbing et al. [22] (who extended the data structure of Kosinas [28]).

1.5.3 Dynamic subgraph connectivity

If the activations/deactivations of vertices are not restricted to batches of bounded size (which afterwards rebound back to the original state of the graph), but they are totally unrestricted, then we are dealing with the so-called *dynamic subgraph connectivity* problem. This model was introduced first by Frigioni and Italiano [14] in the context of planar graph, and was later considered in various works [10, 6, 12], for general graphs. The unrestricted nature of the updates forbids the existence of adequately efficient solutions, according to some conditional lower bounds [19, 25].

1.5.4 Labeling schemes

There is a very interesting line of work [9, 35, 23, 36, 30] that provides connectivity oracles for vertex and edge failures which can answer the queries given access only to some labels that have been assigned to the elements of interest (i.e., to the query vertices, and to the vertices or edges that have failed). Here the goal is to optimize the size of the labels that are attached to the elements of the graph, and the time to compute the labels (in a preprocessing phase). Very recently, such labeling schemes were presented for the first time for vertex cut queries, by Jiang, Parter, and Petruschka [24].

1.5.5 κ-connectivity oracles

There is a related problem of constructing an oracle that can efficiently answer κ-connectivity queries. I.e., given two vertices x and y, the question is "are x and y κ-connected"? The query can be extended as: "if x and y are not κ-connected, what is their connectivity"? or "what is a λ-vertex cut, with $\lambda < \kappa$, that separates x and y"? For $\kappa \leq 4$, optimal oracles exist, w.r.t. the space usage and query time [2, 26]. A data structure for the case of κ-connected graphs was first given by Cohen et al. [8], and a faster construction, informed by an elaborate analysis of the structure of minimum vertex cuts, was given by Pettie and Yin [39]. Various other solutions are known when there is no restriction on the connectivity of the graph (e.g., [34, 38, 27]).

1.5.6 Directed connectivity

There are similar problems for reachability under vertex failures in directed graphs. The most efficient oracle known, for general reachability, is given by van den Brand and Saranurak [42] (which provides the answer with high probability). For the case of single source reachability, dominator trees [4] and an oracle by Choudhary [7], provide optimal space and query time for one and two vertex failures, respectively. (However, the *construction time* of the oracle in [7] is probably not optimal.) For the case of strong connectivity, an optimal solution is given by Georgiadis et al. [16] for one vertex failure, which can also report the number of strongly connected components upon removal of a vertex, in constant time. For strong connectivity under multiple failures, an efficient oracle was given by Baswana et al. [1]. The complexity landscape for the case of more vertex failures is still under exploration, and some lower bounds were given by Chakraborty and Choudhary [5].

1.6 Organization

Throughout this paper we assume that we work on a connected undirected graph G, with n vertices and m edges. Since our approach is to build on the DFS-based framework of Kosinas [28], it is natural to spend some time in order to review the basics of that framework. Thus, we provide a brief (but self-contained) overview of this DFS-based approach in Section 2, and we show how it can be easily extended in order to answer queries about the number of connected components upon removal of a set of vertices (in Section 2.3). In Section 3.1 we introduce some of the DFS-based parameters that we will need for the oracle that handles up to three vertex failures. Then, as a warm-up, we show how we can easily handle the case of two vertex failures (in Section 3.2), by relying on some of those DFS-based parameters (and by completely avoiding the use of block- and SPQR-trees). In Section 3.3, we sketch the case analysis and the techniques that provide the oracle for $d_\star = 3$. Due to the space constraints, the details that establish our result are given in the full version of the paper.

2 The DFS-based framework for connectivity oracles

2.1 Basic definitions and notation

Let T be a DFS tree of G, with start vertex r [41]. 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, or the set of vertices on that path. For any 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 $T(v) = \{v, v+1, \ldots, v + ND(v) - 1\}$, and therefore we can check the ancestry relation in constant time. If c is a child of a vertex v, we define *the next sibling* of c as the lowest child of v that is greater than c (w.r.t. the DFS numbering).

A DFS tree T has the following very convenient property that makes it particularly suitable for solving various connectivity problems: the endpoints of every non-tree edge of G are related as ancestor and descendant on T [41], 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 [41]). Whenever (x, y) denotes a back-edge, we make the convention that $x > y$. We call x and y the higher and the lower, respectively, endpoint of (x, y).

Given a DFS tree T of G (together with a DFS numbering), we get another DFS tree T' of G (providing a different DFS numbering) if we rearrange the children list of every vertex. If T' is derived from T in this way, we call it *a permutation of* T. A recent work by Kosinas [28] has demonstrated the usefulness in considering a collection of permutations of a base DFS tree, in order to provide an efficient (and efficiently constructed) oracle for connectivity queries under vertex failures. Notice that, although the DFS numbering may change, the ancestry relation is an invariant across all permutations of a base DFS tree. The usefulness in considering permutations of a base DFS tree will become apparent in Section 3.2. (In the sequel, we will actually need only two permutations of the base DFS tree T, which we denote as T_{lowInc} and $T_{highDec}$, and we define in Section 3.1.)

For a vertex v, we will use $depth(v)$ to denote the depth of v on the DFS tree. This is defined recursively as $depth(r) := 0$, and $depth(v) := depth(p(v)) + 1$, for any vertex $v \neq r$. (Notice that the *depth* of vertices is an invariant across all permutations of a base DFS tree.) We will use the *depth* of vertices in order to initialize a level-ancestor oracle [3] on the DFS tree, which answers queries of the following form: $\texttt{QueryLA}(v, d) \equiv$ "return the ancestor of v which has depth d". The oracle in [3] can be initialized in $O(n)$ time, and can answer every level-ancestor query in constant time. We need those queries in order to be able to find in constant time the child of v in the direction of u, for any two vertices v and u such that v is a proper ancestor of u. This child is given precisely by the answer to $\texttt{QueryLA}(u, level(v) + 1)$.

2.2 Internal components and hanging subtrees

Given a set $F = \{f_1, \ldots, f_k\}$ of vertices of G that have failed to work, we have that $T \setminus F$ is possibly split into several connected components. Since every connected component C of $T \setminus F$ is a subtree of T, it makes sense to speak of the root r_C of C. Now, following the terminology from [28], we distinguish two types of connected components of $T \setminus F$: internal components, and hanging subtrees. (See Figure 1.) If C is a connected component of $T \setminus F$ such that r_C is an ancestor of at least one vertex from F, then C is called an *internal component*. Otherwise, C is called *a hanging subtree*. Notice that the root of a hanging subtree H is a child of a failed vertex f, and so we say that H is a hanging subtree of f. The important observation from [28] is that, although there may exist $\Omega(n)$ hanging subtrees (irrespective of k), the number of internal components is at most k. Thus, in the case of three vertex-failures, we have at most three internal components.

The main challenge in determining the connectivity relation in $G \setminus F$ is to determine the connection between the internal components of $T \setminus F$. (For more on that, see Section 2.4.) This is because, for a hanging subtree with root c, it is sufficient to know at least k back-edges of the form (x, y) with $x \in T(c)$ and $y < p(c)$, with pairwise distinct lower endpoints. We call such a set of back-edges *the surviving back-edges of* $T(c)$. Then, we have that $T(c)$ is connected with an internal component of $T \setminus F$ in $G \setminus F$, if and only if there is at least one surviving back-edge of $T(c)$ whose lower endpoint is not in F. Furthermore, in order for $T(c)$ to be connected with another hanging subtree $T(c')$ in $G \setminus F$, it is necessary that both of them remain connected with an internal component. Thus, by establishing the connection in $G \setminus F$ between the internal components of $T \setminus F$, and by having stored, for every vertex c, a set of (candidate) surviving back-edges for $T(c)$, we can now easily determine in $O(k)$ time whether two vertices remain connected in $G \setminus F$. (For the details, we refer to [28].)

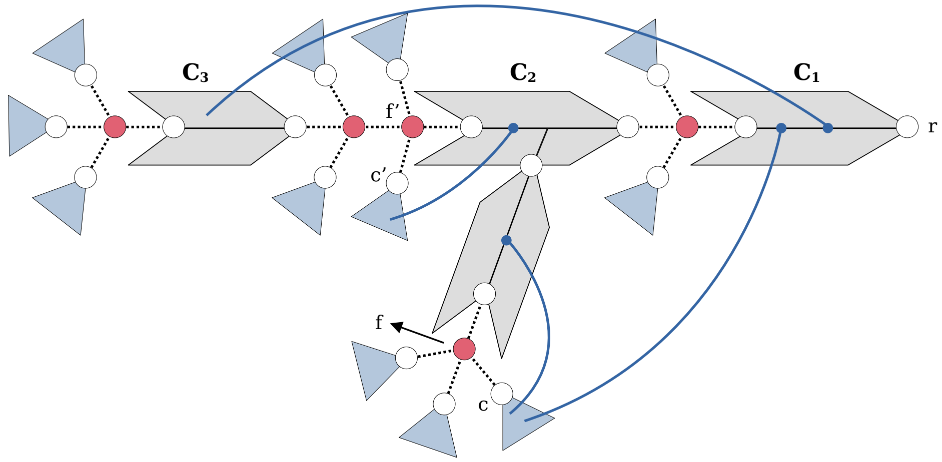

Figure 1 An example of internal components (in gray) and hanging subtrees (in blue) of a DFS tree rooted at r, given a set of vertices that have failed to work. (The failed vertices are coloured red.) We have that $T(c)$ is a hanging subtree of the failed vertex f, and $T(c')$ is a hanging subtree of f'. Notice that there two kinds of back-edges that remain in the graph: those that connect internal components, and those that connect hanging subtrees and internal components. (There is no direct connection between two hanging subtrees with a back-edge.) We have that C_3 remains connected with C_1 directly with a back-edge, and C_1 remains connected with C_2 through the mediation of the hanging subtree $T(c)$.

2.3 Computing the number of connected components of $G \setminus F$

The distinction into internal components and hanging subtrees of $T \setminus F$ is the key to determine the number of connected components of $G \setminus F$. As we will discuss in Section 2.4, there is a graph $\mathcal{R}$, called the connectivity graph, whose nodes correspond to the internal components of $T \setminus F$, and whose connected components capture the connectivity relation between the internal components of $T \setminus F$ in $G \setminus F$. More precisely, two internal components are connected in $G \setminus F$ if and only if their corresponding nodes are connected in $\mathcal{R}$. Now, if we consider a hanging subtree of $T \setminus F$, there are two possibilities: either it is isolated in $G \setminus F$ (i.e., a connected component of $G \setminus F$), or it is connected with an internal component through a surviving back-edge. Thus, the number of connected components of $G \setminus F$ is: #[connected components of $\mathcal{R}$]+#[isolated hanging subtrees].

The data structure from [28] can build the connectivity graph $\mathcal{R}$ in $O(d^4 \log n)$ time, where $d = |F|$. Thus, it remains to compute the number of isolated hanging subtrees of $T \setminus F$ in $G \setminus F$. To do this, we process the failed vertices from F (in any order), and for each of them we determine the number of its children that induce isolated hanging subtrees. For every vertex $v \in F$, and every child c of v, we have that $T(c)$ induces an isolated hanging subtree if and only if: (1) no vertex from F is a descendant of c, and (2) every surviving back-edge (x, y) of $T(c)$ has $y \in F$. (See Figure 2.) Notice that (1) is necessary in order to ensure that $T(c)$ is indeed a hanging subtree, and then (2) ensures that $T(c)$ is isolated.

To facilitate the search for the number of children of a vertex $v \in F$ that induce isolated hanging subtrees, we need to enrich the data structure from [28] as follows. Recall that, for every vertex c, the data structure stores $d_\star$ surviving back-edges for $T(c)$. What we really care about are the lower endpoints of those back-edges. Thus, for every vertex c, we store

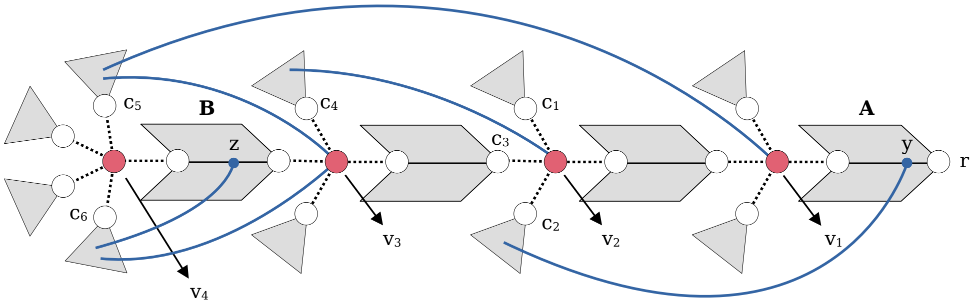

Figure 2 The vertices $\{v_1, v_2, v_3, v_4\}$ have failed to work. $T(c_1)$ is an isolated hanging subtree of v_2, because it has no surviving back-edges. On the other hand, $T(c_2)$ is not an isolated hanging subtree of v_2, because it has a surviving back-edge whose lower endpoint is not a failed vertex. $T(c_3)$ is not a hanging subtree of v_2, because there are descendants of c_3 that are failed vertices. $T(c_4)$ is an isolated hanging subtree of v_3, because the lower endpoint of its only surviving back-edge is v_2. $T(c_5)$ is an isolated hanging subtree of v_4, because the lower endpoints of its two surviving back-edges are v_1 and v_3. $T(c_6)$ is a non-isolated hanging subtree of v_4, because it has a surviving back-edge whose lower endpoint is not a failed vertex. Following the notation in the main text, and assuming that we store up to four surviving back-edges, we have the sorted lists $\mathcal{L}(c_1) = \{\perp, \perp, \perp, \perp\}$, $\mathcal{L}(c_2) = \{y, \perp, \perp, \perp\}$, $\mathcal{L}(c_3) = \{v_1, \perp, \perp, \perp\}$, $\mathcal{L}(c_4) = \{v_2, \perp, \perp, \perp, \}$, $\mathcal{L}(c_5) = \{v_1, v_3, \perp, \perp\}$, and $\mathcal{L}(c_6) = \{v_3, z, \perp, \perp\}$.

the lower endpoints of the $d_\star$ surviving back-edges of $T(c)$ in a list $\mathcal{L}(c)$, sorted in increasing order (w.r.t. the DFS numbering). (Notice that some of the elements in $\mathcal{L}(c)$ may be $\perp$, and we make the convention that $\perp$ is an element greater than every vertex; see Figure 2 for an illustration.) Then, we sort the children list of every vertex in increasing order w.r.t. the $\mathcal{L}$ lists of its children, where the comparison between the $\mathcal{L}$ lists is done lexicographically. We can use bucket-sort in order to have all children lists sorted thus in $O(d_\star n)$ time in total. So, for a vertex v, let $C(v)$ be its sorted children list. (Notice that the asymptotic complexity of the construction time and the space usage of the data structure stays the same.)

Now, given a vertex $v \in F$, we will show how to count the number of children of v that induce isolated hanging subtrees. (Then, the total sum of the number of those children, for all vertices $v \in F$, gives the number of the isolated hanging subtrees of $T \setminus F$ in $G \setminus F$.) Let $v_1, \dots, v_k$ be the proper ancestors of v in F, sorted in increasing order. Now let c be a child of v that induces a hanging subtree. Then, $T(c)$ is isolated if and only if $\mathcal{L}(c) \subseteq \{v_1, \dots, v_k\}$ (where we ignore all "$\perp$" entries in $\mathcal{L}(c)$). Thus, for every subset F' of $\{v_1, \dots, v_k\}$ (including the empty set), we search for (the endpoints of) the segment $\mathcal{S}$ of $C(v)$ that consists of all children c of v with $\mathcal{L}(c) = F'$. Notice that $\mathcal{S}$ is indeed a segment (i.e., consists of consecutive entries) of $C(v)$, and can be found in $O(|F'| \log n)$ time using binary search on $C(v)$. (This is because the comparisons are done lexicographically, but we only need to access the first $|F'|$ entries of the $\mathcal{L}$ lists of the vertices in $C(v)$.) Since $|F'| \leq k < |F| = d$, and this is done for every subset of $\{v_1, \dots, v_k\}$, we have that all these segments $\mathcal{S}$ can be found in $O(2^d d \log n)$ time in total for v, and thus in $O(2^d d^2 \log n)$ time in total for all vertices in F. (This establishes the time bound for the query for the number of connected components of $G \setminus F$.) It is important to notice that, two *distinct* subsets F_1 and F_2 of $\{v_1, \dots, v_k\}$, provide *disjoint* segments $\mathcal{S}_1$ and $\mathcal{S}_2$.

Now we are almost done. Since we know the endpoints of $\mathcal{S}$, we also know its size, but we have to subtract from $|\mathcal{S}|$ the number of children of v in $\mathcal{S}$ that do not induce hanging subtrees. To do that, we process all vertices v' from F, and for every v' that is a proper descendant of v, we ask for the child c of v that is an ancestor of v'. (We note that c can be found in constant time using a level-ancestor query, as explained in Section 2.1; we assume that we have initialized an oracle for such queries on the DFS tree that is given by the children lists $C(v)$.) Then, we can check in constant time whether $c \in \mathcal{S}$. If that is the case, then we mark c as a child that does not induce a hanging subtree. After the processing of all those v', we can subtract from $|\mathcal{S}|$ the number of vertices in $\mathcal{S}$ that do not induce hanging subtrees of v (which are precisely all the vertices that we have marked). This discussion establishes Theorem 2.

2.3.1 The case where G is $d_\star$-connected

The computation of the number of connected components of $G \setminus F$ can be sped up if we know that G is κ-connected, for some $\kappa \leq d_\star$. (Of course, if $\kappa > d_\star$, then $G \setminus F$ is connected, for any set of vertices F with $|F| \leq d_\star$.) As a warm-up, let us consider the case where $\kappa = d_\star$. Then, $G \setminus F$ may be disconnected only if $|F| = d_\star$. (In every other case, i.e., when $|F| < d_\star$, we can immediately report that $G \setminus F$ consists of a single connected component.)

So let F be a vertex set with $|F| = d_\star$. Notice that every vertex v in F, except possibly one, has the property that there are less than $d_\star$ vertices in F that are ancestors of v. Then, for such a vertex v, we have that no child c of v induces an isolated hanging subtree. This is precisely due to the fact that G is $d_\star$-connected: otherwise, i.e., if there was a child c of v such that $T(c)$ induces an isolated hanging subtree, then the set of the ancestors of v in F would constitute a set with less than $d_\star$ vertices whose removal would disconnect $T(c)$ from the rest of the graph. Thus, if no vertex in F has $d_\star$ ancestors from F, then $G \setminus F$ is connected. Otherwise, let $\tilde{v}$ be the unique vertex in F with the property that all vertices from F are ancestors of it. Then, only $\tilde{v}$ may have isolated hanging subtrees. But it is easy to find them with a single binary search in $C(\tilde{v})$: it is sufficient to search for those children c of $\tilde{v}$ in $C(\tilde{v})$ with $\mathcal{L}(c) = F \setminus \{\tilde{v}\}$. (Because, since G is $d_\star$-connected, every child c of $\tilde{v}$ has at least $d_\star - 1$ non-$\perp$ entries in $\mathcal{L}(c)$.) This binary search (for the endpoints of the corresponding segment of $C(\tilde{v})$) takes time $O(d_\star \log n)$ (because the comparisons in this binary search are done lexicographically on the $\mathcal{L}$ lists, which have $d_\star$ entries). Thus, we can find the number of connected components of $G \setminus F$ in $O(d_\star^4 \log n)$ time (i.e., the query time is dominated by the computation of the connectivity graph $\mathcal{R}$). This establishes Corollary 3.

2.3.2 The case where G is κ-connected (for $\kappa \leq d_\star$)

For the general case where G is κ-connected, with $\kappa \leq d_\star$, the query for the number of the connected components of $G \setminus F$ is non-trivial only if $\kappa \leq d \leq d_\star$, where $d = |F|$. Here we are guided by the same intuition as in the previous paragraph (for the case where G is $d_\star$-connected).

So let F be a set of failed vertices with $|F| = d$ and $\kappa \leq d \leq d_\star$, and let F' be the subset of F that consists of those vertices that have at least κ ancestors from F. Then, it should be clear that only vertices from F' can provide isolated hanging subtrees. To see this, consider a vertex $v \in F$ that has less than κ ancestors from F (where v is included as an ancestor of itself from F). Then, every child c of v has less than κ ancestors from F, and therefore, by removing all of them, we have that $T(c)$ is still connected with the remaining graph (since the graph is κ-connected). This implies the existence of a surviving back-edge for $T(c)$ – which still exists even if we remove all of F from G, and demonstrates that $T(c)$ remains connected with at least one internal component of $T \setminus F$ in $G \setminus F$.

Now, for every $v \in F'$, we let $Anc(v)$ denote the set of its proper ancestors from F, and let $N(v) = |Anc(v)|$. Then, for every $v \in F'$, and for every subset A of $Anc(v)$ with at least $\kappa - 1$ elements, we must perform a binary search, as explained in Section 2.3, for the segment of $C(v)$ that consists of those vertices c with $\mathcal{L}(c) = A$. Thus, we perform $\binom{N(v)}{\kappa-1} + \cdots + \binom{N(v)}{N(v)}$ binary searches, which, in total, take time $O((\binom{N(v)}{\kappa-1}(\kappa-1) + \cdots + \binom{N(v)}{N(v)}N(v)) \log n)$.

Thus, we can provide the following upper bound for the total time for finding the number of isolated hanging subtrees. Notice that F' consists of at most $d - \kappa + 1$ vertices, and that $N(v) \leq d - 1$ for every $v \in F'$. Thus, the most pessimistic upper bound that we can get is $O((d - \kappa + 1)(\binom{d-1}{\kappa-1}(\kappa-1) + \cdots + \binom{d-1}{d-1}(d-1)) \log n)$. Notice that this bound dominates the time that it takes to find, for every $v \in F'$, for every subset A of $Anc(v)$, the elements in the segment of $C(v)$, that corresponds to A, that are ancestors of vertices from F (since v has at most $d - \kappa + 1$ descendants from F).

To appreciate this time bound, let us suppose, for example, that G is κ-connected, and that F is a set of $\kappa + 1$ vertices. Then, we can determine the number of isolated hanging subtrees of $T \setminus F$ in $O(\kappa^2 \log n)$ time, and thus the number of connected components of $G \setminus F$ in time $O(\kappa^4 \log n)$. (Thus, the query time is still dominated by the time to compute the connectivity graph $\mathcal{R}$.)

2.4 Determining the connection between the internal components

Upon receiving the set F of failed vertices, the main problem is to determine the connectivity relation in $G \setminus F$ between the internal components of $T \setminus F$. Here it helps to distinguish two basic types of connection between internal components: either (1) directly with a back-edge, or (2) through the mediation of a hanging subtree. These two connections can only exist for two internal components that are related as ancestor and descendant. So let C and C' be two distinct internal components such that r_C is a descendant of $r_{C'}$. Now, if (1) there is a back-edge (x, y) such that $x \in C$ and $y \in C'$, then we say that C and C' are connected *directly with a back-edge*. And if (2) there is a hanging subtree H for which there exist a back-edge (x, y) with $x \in H$ and $y \in C$, and a back-edge (x', y') with $x' \in H$ and $y' \in C'$, then we say that C and C' are connected *through the mediation of H*. (Notice that, in this case, we have that r_H must be a descendant of r_C.) It is not difficult to verify the following:

▶ **Lemma 6** (Implicit in [28]). *Let F be a set of failed vertices, and let C and C' be two internal components of $T \setminus F$. Then C and C' are connected in $G \setminus F$ if and only if: there is a sequence of internal components $C_1, \ldots, C_t$, with $C_1 = C$ and $C_t = C'$, such that C_i and C_{i+1} are connected either directly with a back-edge or through the mediation of a hanging subtree, for every $i \in \{1, \ldots, t-1\}$.*

Lemma 6 implies that, in order to determine the connectivity relation between the internal components, it is sufficient to check, for every pair of internal components C and C' (that are related as ancestor and descendant), whether they are connected directly with a back-edge, or through the mediation of a hanging subtree. If this is the case, then we add an artificial edge between (some representative nodes for) C and C'. Thus, we have built a graph $\mathcal{R}$, which in [28] was called *the connectivity graph*, whose nodes are (representatives of) the internal components, and whose edges represent the existence of a connection either with a back-edge or through a hanging subtree. Then, it is sufficient to compute the connected components of $\mathcal{R}$, in order to get the connectivity relation between all internal components.

So the problem we have to solve is the following: given two internal components C and C' (that are related as ancestor and descendant), how can we determine efficiently whether C and C' are connected directly with a back-edge, or through the mediation of a hanging subtree? To determine the existence of a back-edge that directly connects C and C' will usually be a very straightforward task for us, with the exception of some cases where we will have to use an indirect, counting argument. (Recall that we will only have to deal with at most three internal components, but still our task is challenging.) The problem of establishing a connection through the mediation of a hanging subtree is, in general, much more difficult, because the number of hanging subtrees can be $\Omega(n)$, and thus it is forbidding to check for each of them, explicitly, whether it provides some desired back-edges. Thus, we resort to somehow considering large batches of hanging subtrees at once, and this is why we work with specific permutations of a base DFS tree. The idea is to rearrange the children lists, so that children with "similar properties" (which capture properties of their induced subtrees) appear as consecutive siblings in the children lists, and so we can process large segments of those lists at once. We will provide a concrete instance of this idea in the following section.

3 Designing an oracle for $d_\star = 3$

Let F be a set of at most three vertices that have failed to work. In order to determine the connectivity between two vertices x and y on $G \setminus F$, we use the DFS-based framework outlined in Section 2. Thus, it is sufficient to establish the connectivity between the (at most three) internal components of $T \setminus F$, where T is a DFS tree of G.

In order to reduce the number of cases that may appear, it is very convenient to make the following two assumptions.

▶ **Assumption 7.** *The root of T is not a failed vertex, and no two failed vertices are related as parent and child.*

▶ **Assumption 8.** *Every back-edge has the property that its higher endpoint is a leaf of T, which is not a failed vertex.*

These two assumptions follow from the observation, that splitting the edges of the graph does not change the connectivity structure under vertex failures. (For more details, we refer to the full version of the paper.)

Now, let us first note that the case where F consists of a single vertex is trivial. Assumption 7 implies that there is a single internal component in $T \setminus F$, and thus there is nothing to do with it: every connectivity query for two vertices x and y in $G \setminus F$ is answered with the help of the surviving back-edges (if needed), as explained in Section 2.2.

In the following section we introduce some elementary DFS-based concepts. These will help us in order to handle easily the case where $|F| = 2$ (in Section 3.2). This case provides an opportunity to demonstrate some of the techniques that will also be useful for handling the case where $|F| = 3$. Finally, in Section 3.3, we outline an analysis into cases, for determining the connection of the internal components of $T \setminus F$ when $|F| = 3$.

3.1 Some concepts defined on a DFS tree

Here we introduce some DFS-based concepts that will help the reader follow our discussion in the next two subsections. These are defined for all vertices (although they may be $\perp$ for some of them), and can be computed in linear time in total.

The most fundamental concept that we rely on is that of *the leaping back-edges over the parent of a vertex*. Specifically, for every vertex v, we let $B_p(v)$ denote the set of all back-edges (x, y) such that $x \in T(v)$ and $y < p(v)$. The sets B_p prove to be very useful for vertex-connectivity purposes, but we cannot afford to compute all of them explicitly, since there are graphs with $O(n)$ edges for which the total size of all B_p sets (for some DFS trees) is $\Omega(n^2)$. Instead, we will rely on parameters that capture some properties of the back-edges in those sets that will be sufficient for our purposes.

First, we introduce some concepts that are defined w.r.t. the lower endpoints of the back-edges in the B_p sets. For a vertex v, we define $low(v) := \min\{y \mid \exists(x,y) \in B_p(v)\}$ and $high_p(v) := \max\{y \mid \exists(x,y) \in B_p(v)\}$. The *low* points were used by Tarjan [41] in order to solve various algorithmic problems of small connectivity, and the $high_p$ points were introduced by Georgiadis and Kosinas [17] in order to solve some algorithmic problems that relate to vertex-edge cuts.

Next, we introduce some concepts that are defined w.r.t. the higher endpoints of the back-edges in the B_p sets. For a vertex v, we define $L_p(v) := \min\{x \mid \exists(x,y) \in B_p(v)\}$ and $R_p(v) := \max\{x \mid \exists(x,y) \in B_p(v)\}$. These are called the *leftmost* and the *rightmost* point, respectively, of v. Furthermore, we also define $M_p(v) := nca\{L_p(v), R_p(v)\}$. Notice that $M_p(v)$ (as a vertex) is an invariant across all permutations of a base DFS tree, although $L_p(v)$ and $R_p(v)$ may vary.

We define the permutations T_{lowInc} and $T_{highDec}$ of the base DFS tree T. The children lists in T_{lowInc} are sorted in increasing order w.r.t. the *low* points (where we consider $\perp$ to be the largest element), and the children lists in $T_{highDec}$ are sorted in decreasing order w.r.t. the $high_p$ points (where we consider $\perp$ to be the smallest element). Once the *low* and $high_p$ points of all vertices are computed, we can easily construct T_{lowInc} and $T_{highDec}$ in $O(n)$ time with bucket-sort.

3.2 The case $|F| = 2$

Let $F = \{u, v\}$. Then there are two possibilities: either u and v are not related as ancestor and descendant, or one of them is an ancestor of the other. In the first case, Assumption 7 implies that there is a single internal component in $T \setminus F$, and therefore this case is trivial.

So let us consider the second case, and let us assume w.l.o.g. that u is an ancestor of v. Then, by Assumption 7, we have the situation depicted in Figure 3. Thus, there are two internal components, A and B, and the goal is to determine whether they are connected, either directly with a back-edge, or through a hanging subtree of v. For this, we can use Lemma 9, which provides a very simple constant-time testable criterion for the connectivity of A and B in $G \setminus F$. This lemma also provides a good example of the techniques and arguments that we will employ in the most demanding cases as well.

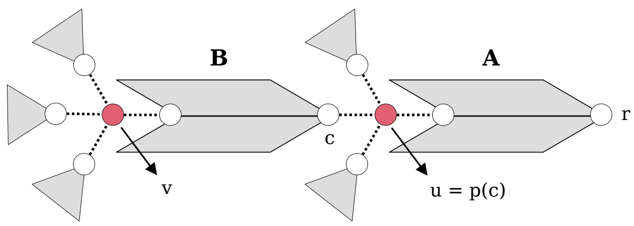

Figure 3 An illustration of the situation analyzed in Lemma 9. The set of failed vertices is $\{v, p(c)\}$, and the goal is to check whether the parts A and B remain connected. To do this, we rely on the leftmost and the rightmost points, $L_p(c)$ and $R_p(c)$, respectively, that provide back-edges from $T(c)$ to $T[p(p(c)), r]$.

▶ **Lemma 9.** *Let c and v be two vertices such that c is a proper ancestor of v, and $p(c) \neq r$. Let A be the set of vertices that are not descendants of $p(c)$, and let B be the set of vertices that are descendants of c, but not of v. (See Figure 3.) Let $L_p(c)$ and $R_p(c)$ be the leftmost and the rightmost points of c on $T_{highDec}$. Then, after removing v and $p(c)$ from the graph, we have that A is connected with B if and only if one of the following is true:*

(1) *One of $L_p(c)$ and $R_p(c)$ is on B.*

(2) *Both $L_p(c)$ and $R_p(c)$ are proper descendants of v, and $high_p(d) \in B$, where d is the child of v that is an ancestor of $L_p(c)$.*

▶ Remark 10. The idea behind the "⇒" direction is the following. If (1) is not true, then B and A are not connected directly with a back-edge. Therefore, there must exist a hanging subtree of v that connects B and A. Then, since we are working on $T_{highDec}$, the child d of v that is an ancestor of $L_p(c)$ must necessarily induce such a subtree. The "⇐" direction is immediate.

3.3 The case $|F| = 3$

In the case $|F| = 3$, we initially determine the ancestry relation between the vertices from F. Thus, we have the following cases (see also Figure 4):

(1) No two vertices from F are related as ancestor and descendant.

(2) Two vertices from F are related as ancestor and descendant, but the third is not related in such a way with the first two.

(3) One vertex from F is an ancestor of the other two, but the other two are not related as ancestor and descendant.

(4) Any two vertices from F are related as ancestor and descendant.

Notice that cases (1)-(4) are mutually exclusive, and exhaust all possibilities for the ancestry relation between the vertices from F.

In case (1) we have a single internal component in $T \setminus F$, and therefore this case is trivial. In case (2) we have two internal components, and this case is handled precisely as the case where $|F| = 2$ (i.e., we can determine the connectivity between the two internal components, using the parameters described in Lemma 9, which are associated with the child of the failed vertex that is an ancestor of the other failed vertex).

Case (3) is a little bit more involved, but it is handled with similar techniques as case (2). Let $F = \{u, v, w\}$, and let us assume w.l.o.g. that u is an ancestor of v and w, but v and w are not related as ancestor and descendant. Here we distinguish between the case where v and w are descendants of different children of u, and the case where v and w are descendants of the same child of u. The first case is handled precisely with the same technique as case (2). For the second case we have to be a little bit more careful, but the same technique essentially applies here too.

Case (4) is much more involved. Let $F = \{u, v, w\}$, and let us assume w.l.o.g. that u is an ancestor of v, and v is an ancestor of w. Let also c be the child of u in the direction of v, and let d be the child of v in the direction of w, and let the three internal components A, B and C, of $T \setminus F$ be as depicted in Figure 4. Here we found it convenient to distinguish all different cases w.r.t. the location of $M_p(c)$. Thus, we have the six cases: (i) $M_p(c) = \perp$, (ii) $M_p(c) \in B$, (iii) $M_p(c) = v$, (iv) $M_p(c)$ is a descendant of a child d' of w, (v) $M_p(c) = w$, and (vi) $M_p(c) \in C$. (Notice that these cases are mutually exclusive, and exhaust all possibilities for $M_p(c)$.) Before proceeding further, we would encourage the reader to try to solve some of those cases independently, before studying our own solutions.

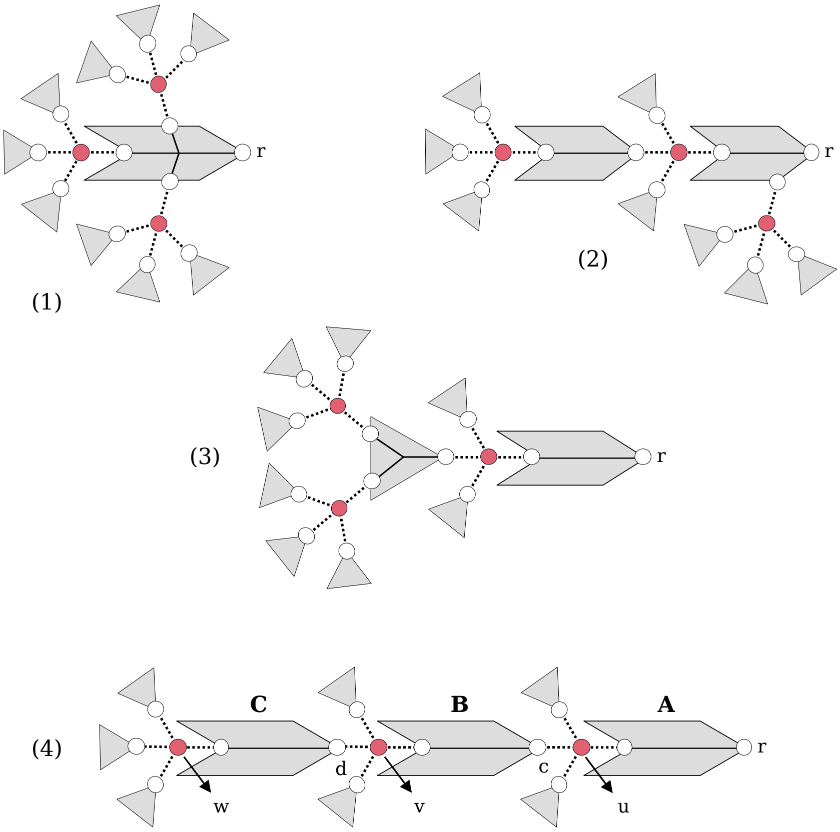

Figure 4 The four possibilities for the ancestry relation between the vertices from F, when $|F| = 3$.

In case (i) we have that A is isolated from B and C, and it remains to determine whether B remains connected with C (either directly with a back-edge, or through a hanging subtree of w). We found that this case becomes easily manageable if we distinguish the four different cases for the location of $M_p(d)$: i.e., either $M_p(d) = \perp$, or $M_p(d)$ is a descendant of a child of w, or $M_p(d) = w$, or $M_p(d) \in C$. This case is the simplest one, and very instructive for the more demanding cases.

In case (ii) we have that B is connected with A directly with a back-edge, and it remains to determine whether C is connected with either B or A (either directly with a back-edge, or through a hanging subtree of w). This is done similarly as in case (i).

In case (iii), there is a possibility that a child of u induces a hanging subtree that connects A and B. This is easy to check, and, if true, we can handle the rest as in case (ii). Otherwise, we have that only $T(d)$ has the potential to provide back-edges that establish the connectivity of A and B. Here we distinguish all the different cases for $high_p(d)$ and $low(d)$, and the most difficult case appears when $high_p(d) \in B$ and $low(d) \in A$. However, in this case d is uniquely determined by c, and so we can gather enough information during the preprocessing phase in order to accommodate for this case.

In case (iv), notice that $T(d')$ is the only hanging subtree that may connect A with either B or C. Here we found it very convenient to distinguish between the different cases for $M_p(d)$, as in case (i). Then, if $M_p(d)$ is a descendant of a child of w, it must necessarily be a descendant of d'. Thus, this case is particularly easy. If $M_p(d) \in C$, then we can use either $L_p(d)$ and $R_p(d)$ in order to establish the connectivity between C and B, or the leftmost and rightmost points of d that reach the segment $T[p(v), c]$. Thus, this case is still not difficult to manage. The case $M_p(d) = w$ presents a singular difficulty, and we have to resort to a counting argument to determine the existence of some back-edges.

In case (v) we distinguish the two cases for $M_p(d)$: either $M_p(d) = w$, or $M_p(d) \in C$. In the first case, we can appropriately use the leftmost and the rightmost points of d that reach the segment $T[p(v), c]$. In the second case, we follow essentially the same approach as for the case $M_p(d) \in C$ when $M_p(c)$ is a descendant of a child of w.

Finally, in case (vi), we have $M_p(d) \in C$ (as a consequence of $M_p(c) \in C$). An easy case appears if $M_p(d) = M_p(c)$, because then we can use the leftmost and rightmost points of d that reach the segment $T[p(v), c]$. Otherwise, (after sorting out some easier cases), there appears again a singular difficulty, for which we have to resort to a counting argument to determine the existence of some back-edges.

References

1. Surender Baswana, Keerti Choudhary, and Liam Roditty. An efficient strongly connected components algorithm in the fault tolerant model. *Algorithmica*, 81(3):967–985, 2019. doi:10.1007/S00453-018-0452-3.
2. Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 15(4):302–318, 1996. doi:10.1007/BF01961541.
3. Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/J.TCS.2003.05.002.
4. Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008. doi:10.1137/070693217.
5. Diptarka Chakraborty and Keerti Choudhary. New extremal bounds for reachability and strong-connectivity preservers under failures. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020*, volume 168 of *LIPIcs*, pages 25:1–25:20, 2020. doi:10.4230/LIPICS.ICALP.2020.25.
6. Timothy M. Chan, Mihai Pătraşcu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM J. Comput.*, 40(2):333–349, 2011. doi:10.1137/090751670.
7. Keerti Choudhary. An optimal dual fault tolerant reachability oracle. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, volume 55 of *LIPIcs*, pages 130:1–130:13, 2016. doi:10.4230/LIPICS.ICALP.2016.130.
8. Robert F. Cohen, Giuseppe Di Battista, Arkady Kanevsky, and Roberto Tamassia. Reinventing the wheel: an optimal data structure for connectivity queries. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, STOC 1993*, pages 194–200. ACM, 1993. doi:10.1145/167088.167152.
9. Michal Dory and Merav Parter. Fault-tolerant labeling and compact routing schemes. In *PODC '21: ACM Symposium on Principles of Distributed Computing, 2021*, pages 445–455. ACM, 2021. doi:10.1145/3465084.3467929.
10. Ran Duan. New data structures for subgraph connectivity. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010*, volume 6198 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 2010. doi:10.1007/978-3-642-14165-2_18.
11. 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.
12. Ran Duan and Le Zhang. Faster randomized worst-case update time for dynamic subgraph connectivity. In *Algorithms and Data Structures - 15th International Symposium, WADS*

2017, volume 10389 of *Lecture Notes in Computer Science*, pages 337–348. Springer, 2017. doi:10.1007/978-3-319-62127-2_29.

13 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.

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

15 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

16 Loukas Georgiadis, Giuseppe F. Italiano, and Nikos Parotsidis. Strong connectivity in directed graphs under failures, with applications. *SIAM J. Comput.*, 49(5):865–926, 2020. doi:10.1137/19M1258530.

17 Loukas Georgiadis and Evangelos Kosinas. Linear-time algorithms for computing twinless strong articulation points and related problems. In *31st International Symposium on Algorithms and Computation, ISAAC 2020*, volume 181 of *LIPIcs*, pages 38:1–38:16, 2020. doi:10.4230/LIPICS.ISAAC.2020.38.

18 Carsten Gutwenger and Petra Mutzel. A linear time implementation of spqr-trees. In *Graph Drawing, 8th International Symposium, GD 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer, 2000. doi:10.1007/3-540-44541-2_8.

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

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

21 John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973. doi:10.1137/0202012.

22 Bingbing Hu, Evangelos Kosinas, and Adam Polak. Connectivity oracles for predictable vertex failures. In *32nd Annual European Symposium on Algorithms, ESA 2024*, volume 308 of *LIPIcs*, pages 72:1–72:16, 2024. doi:10.4230/LIPICS.ESA.2024.72.

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

24 Yonggang Jiang, Merav Parter, and Asaf Petruschka. New oracles and labeling schemes for vertex cut queries. *CoRR*, abs/2501.13596, 2025. doi:10.48550/arXiv.2501.13596.

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

26 Arkady Kanevsky, Roberto Tamassia, Giuseppe Di Battista, and Jianer Chen. On-line maintenance of the four-connected components of a graph (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, FOCS 1991*, pages 793–801. IEEE Computer Society, 1991. doi:10.1109/SFCS.1991.185451.

27 Tuukka Korhonen. Linear-time algorithms for k-edge-connected components, k-lean tree decompositions, and more. *CoRR*, abs/2411.02658, 2024. doi:10.48550/arXiv.2411.02658.

28 Evangelos Kosinas. Connectivity queries under vertex failures: Not optimal, but practical. In *31st Annual European Symposium on Algorithms, ESA 2023*, volume 274 of *LIPIcs*, pages 75:1–75:13, 2023. doi:10.4230/LIPICS.ESA.2023.75.

29 Evangelos Kosinas. Computing the 5-edge-connected components in linear time. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024*, pages 1887–2119. SIAM, 2024. doi:10.1137/1.9781611977912.76.

30 Yaowei Long, Seth Pettie, and Thatchaphol Saranurak. Connectivity labeling schemes for edge and vertex faults via expander hierarchies. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025*, pages 1–47. SIAM, 2025. doi:10.1137/1.9781611978322.1.

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

32 Yaowei Long and Yunfan Wang. Better decremental and fully dynamic sensitivity oracles for subgraph connectivity. In *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024*, volume 297 of *LIPIcs*, pages 109:1–109:20, 2024. doi:10.4230/LIPICS.ICALP.2024.109.

33 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.

34 Zeev Nutov. Data structures for node connectivity queries. In *30th Annual European Symposium on Algorithms, ESA 2022*, volume 244 of *LIPIcs*, pages 82:1–82:12, 2022. doi:10.4230/LIPICS.ESA.2022.82.

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

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

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

38 Seth Pettie, Thatchaphol Saranurak, and Longhui Yin. Optimal vertex connectivity oracles. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, 2022*, pages 151–161. ACM, 2022. doi:10.1145/3519935.3519945.

39 Seth Pettie and Longhui Yin. The structure of minimum vertex cuts. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 105:1–105:20, 2021. doi:10.4230/LIPICS.ICALP.2021.105.

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

41 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

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