Planar Reachability Under Single Vertex or Edge Failures

Giuseppe F. Italiano* Adam Karczmarz† Nikos Parotsidis‡

Abstract

In this paper we present an efficient reachability oracle under single-edge or single-vertex failures for planar directed graphs. Specifically, we show that a planar digraph G can be preprocessed in $O(n \log^2 n / \log\log n)$ time, producing an $O(n \log n)$-space data structure that can answer in $O(\log n)$ time whether u can reach v in G if the vertex x (the edge f) is removed from G, for any query vertices u, v and failed vertex x (failed edge f). To the best of our knowledge, this is the first data structure for planar directed graphs with nearly optimal preprocessing time that answers all-pairs queries under any kind of failures in polylogarithmic time.

We also consider 2-reachability problems, where we are given a planar digraph G and we wish to determine if there are two vertex-disjoint (edge-disjoint) paths from u to v, for query vertices u, v. In this setting we provide a nearly optimal 2-reachability oracle, which is the existential variant of the reachability oracle under single failures, with the following bounds. We can construct in $O(n \operatorname{polylog} n)$ time an $O(n \log^{3+o(1)} n)$-space data structure that can check in $O(\log^{2+o(1)} n)$ time for any query vertices u, v whether v is 2-reachable from u, or otherwise find some separating vertex (edge) x lying on all paths from u to v in G.

To obtain our results, we follow the general recursive approach of Thorup for reachability in planar graphs [J. ACM '04] and we present new data structures which generalize dominator trees and previous data structures for strong-connectivity under failures [Georgiadis et al., SODA '17]. Our new data structures work also for general digraphs and may be of independent interest.

1 Introduction

Computing reachability is perhaps one of the most fundamental problems in directed graphs. Let $G = (V, E)$ be a directed graph with n vertices and m edges. The transitive closure (i.e., all-pairs reachability) problem consists of computing whether there is a directed path from u to v, for all pairs of vertices $u, v \in V$. The single-source reachability variant asks for each $v \in V$ whether there exists a path from s to v, where $s \in V$ is fixed. While single-source reachability can be solved in optimal $O(m)$ time, the fastest algorithm for computing transitive closure runs in $\widetilde{O}(\min(n^\omega, nm))$ time, where $\omega < 2.38$ is the matrix multiplication exponent. Notice that for solving all-pairs reachability one needs $O(n^2)$ space to store the information for all pairs of vertices.

In the oracle variant of all-pairs reachability, we wish to preprocess the input graph and build a data structure that can answer reachability queries between any pair of vertices, while trying to minimize the query time, the preprocessing time, as well as the size of the data structure. Henzinger et al. [32] gave conditional lower bounds for "combinatorial"[1] constructions for this problem. Specifically, they showed that there is no all-pairs reachability oracle that simultaneously requires $O(n^{3-\epsilon})$ time preprocessing and supports queries in $O(n^{2-\epsilon})$ time (for all m), unless there is a truly "combinatorial" algorithm that can multiply two $n \times n$ boolean matrices in $O(n^{3-\epsilon})$ time.

However, non-trivial reachability oracles are known for a few important graph classes. Most notably, for planar digraphs, for which $m = O(n)$, the first reachability oracle with near-linear preprocessing and polylogarithmic query time was obtained by Thorup [50], whereas a decade later Holm et al. [33] presented an asymptotically optimal oracle, with $O(n)$ space and preprocessing time and constant query time. For graph classes admitting balanced separators of size $s(n)$ (which include graphs with treewidth $O(s(n))$, and minor-free graphs for $s(n) = O(\sqrt{n})$), an $\widetilde{O}(n \cdot s(n))$-space reachability oracle with query time $\widetilde{O}(s(n))$ exists[2].

Real-world networks undoubtedly experience link or node failures. This has motivated the research community to develop graph algorithms and data structures that can efficiently deal with failures. A notable example is the notion of dominators in digraphs with respect

*LUISS University, Rome, Italy, gitaliano@luiss.it. Giuseppe F. Italiano is partially supported by MUR, the Italian Ministry for University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).

†Institute of Informatics, University of Warsaw, Poland, a.karczmarz@mimuw.edu.pl. Supported by ERC Consolidator Grant 772346 TUgbOAT, the Polish National Science Centre grant 2017/24/T/ST6/00036, and by the Foundation for Polish Science (FNP) via the START programme.

‡Google Research, nikosp@google.com. This work was partially done while the author was employed at the University of Copenhagen supported by the Grant Number 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM Foundation.

[1] That is, not relying on fast matrix multiplication algorithms, which are often considered impractical.

[2] To obtain such an oracle simply precompute single-source reachability from/to all the $O(s(n))$ vertices of the separator and recurse on the components of G after removing the separator.

to a source vertex s. We say that a vertex x dominates a vertex v if all paths from s to v contain x. The dominance relation from s is transitive and can be represented via a tree called the *dominator tree* from s. The dominator tree from s allows one to answer several reachability under failure queries, such as "are there two edge- (or vertex-) disjoint paths from s to v?" and "is there a path from s to v avoiding a vertex x (or an edge e)?", in asymptotically optimal time. The notion of dominators has been widely used in domains like circuit testing [7], theoretical biology [5], memory profiling [42], constraint programming [44], connectivity [27], just to state some. Due to their numerous applications, dominators have been extensively studied for over four decades [4, 35, 39, 48] and several linear-time algorithms for computing dominator trees are known [6, 13, 14, 30]. While extremely useful, dominator trees are restricted to answering queries only from a single source s.

Oracles that answer queries in the presence of failures are often called *f-sensitivity oracles*[3], where f refers to the upper bound on the number of failures allowed. If not explicitly mentioned, in this paper when we refer to sensitivity oracles we refer to 1-sensitivity oracles that allow failures of either one edge or one vertex. In what follows we denote by $G-F$ the graph obtained from G by deleting the set of vertices or edges F. If F is a single vertex or edge x, we write $G-x$.

In this paper, we study 1-sensitivity oracles for the all-pairs reachability problem in planar digraphs. Specifically, we wish to preprocess a planar graph efficiently and build a possibly small (in terms of space) data structure that can efficiently answer queries of the form "is there a path from u to v avoiding x?", for query vertices u, v and vertex (or edge) x. Moreover, we study *2-reachability problems*, where, given a directed graph G, we wish to determine if there are two vertex-disjoint (resp., edge-disjoint) paths from u to v, for query vertices u, v. In particular, we consider *2-reachability oracles*, which are the existential variant of 1-sensitivity reachability oracles. Here, the desired data structure should, for an arbitrary pair of query vertices (u, v), efficiently find a vertex $x \notin \{u, v\}$ (resp., an edg e) whose failure destroys all $u \to v$ paths in the graph, or declare there is none. Note that in the latter case, vertex v is 2-reachable from vertex u[4].

Our data structures support the aforementioned queries answered with dominator trees, but we allow a source s to be a query parameter as well, as opposed to a dominator tree which assumes a fixed source. We focus on planar graphs not only because they are one of the most studied non-general classes of graphs, but also because dominator trees have been used in the past for solving problems on planar graphs (i.e., circuit testing [7]), and hence our result could potentially motivate further similar applications as an efficient tool that can answer all-pairs dominance queries.

Notice that a simple-minded solution to both 1-sensitivity reachability oracle and 2-reachability oracle problems with $O(n^2)$ space and preprocessing time and $O(1)$ query time is to compute the dominator tree from each source vertex s. In general directed graphs, the all-pairs version of a dominator tree cannot be computed faster than matrix-multiplication or be stored in subquadratic space [26], which can be prohibitive in applications that require the processing of data of even moderate size. In this paper we show how to achieve significantly better bounds for both these problems when the input digraph is planar.

Related work. There has been an extensive study of sensitivity oracles in directed graphs, with the initial studies dating several decades back. Sensitivity oracles for single-source reachability have been studied widely under the name *dominator trees* since the seventies (see e.g., [39]). Choudhary [19] considered the problem of computing 2-sensitivity oracles for single-source reachability. In particular, she showed how to construct a data structure of size $O(n)$ that can answer in constant time reachability queries from a source s to any vertex v in $G-\{x, y\}$, for query vertices v, x, y. While the preprocessing time is not specified, a simple initialization of her data structure requires $O(mn^2)$ time. Baswana et al. [8], considered the version of this problem with multiple failures. Specifically, they presented an f-sensitivity oracle, with size $O(2^f n)$ and preprocessing time $O(mn)$, that computes the set or reachable vertices from the source vertex s in $G-F$ in $O(2^f n)$ time, where F is the set of failed vertices or edges, with $|F| \leq f$.

King and Sagert [37] were the first to study 1-sensitivity oracles for all-pairs reachability under single edge failures. In particular, they gave an algorithm that can answer queries in constant time in directed acyclic graphs (DAGs), after $O(n^3)$ time preprocessing. For general directed graphs Georgiadis et al. [26] showed a near-optimal 1-sensitivity oracle for all-pairs reachability, with $O(n^2)$ space and $O(\min\{mn, n^\omega \log n\})$ preprocessing time, that can answer in constant time queries of the form "is there a path from u to v in $G-x$", for query vertices u, v and failing vertex or edge x. Their approach first produces dominator trees from all sources, which are then used to answer the queries. Sensitivity oracles

[3] We adopt the use of the term from [32]. We note that other terms have also been used in the literature, such as "fault-tolerant" oracles or oracles "for failure prone graphs".

[4] The name 2-reachability comes from the fact that, by Menger's theorem, there exist two internally vertex-disjoint $u \to v$ paths iff no single failing vertex can make v unreachable from u.

for reachability problems admit a trivial lower bound: they cannot be built faster than the time it takes to compute the corresponding (single-source or all-pairs) reachability problem in the static case (i.e., without failures). Very recently, van den Brand and Saranurak [51] presented an f-sensitivity oracle for all-pairs reachability with $O(n^2 \log n)$ size and $O(f^\omega)$ query time, and $O(n^\omega)$ time preprocessing. Their f-sensitivity oracle is nearly optimal for $f \in O(1)$ and is obtained by adopting an improved f-sensitivity oracle for the All-Pairs Shortest Paths problem. Therefore, the problem of constructing an f-sensitivity all-pairs reachability oracle in general digraphs is well understood for small values of f.

As already mentioned, the 2-reachability problem asks to build a data structure that can efficiently report for a pair of query vertices (u, v) a single vertex (resp., edge) that appears in all paths from u to v, or determine that there is no such vertex (resp., edge). Georgiadis et al. [26] show how to precompute the answers to all possible 2-reachability queries in $O(\min\{n^\omega \log n, mn\})$ time. The notion of 2-reachability naturally generalizes to k-reachability where the query asks for a set of at most $(k-1)$ vertices (resp., edges) whose removal leaves v unreachable from u, or determine that there is no such set of vertices (resp., edges). For the case of k-reachability with respect to edge-disjoint paths, Abboud et al. [1] show how to precompute the answer for all pairs of vertices in $O(\min\{n^\omega, mn\})$ when $k = O(1)$, but only in the case of DAGs. For the case of (non-necessarily acyclic) planar graphs, Łącki et al. [41] showed an algorithm with $\widetilde{O}(n^{5/2} + n^2 k)$ running time. Hence, there are no non-trivial results on the k-reachability problem in general directed graphs.

Abboud et al. [1] also considered a weaker version of k-reachability in which they only distinguish whether there are k disjoint paths, or less (without reporting a set of at most $k-1$ vertices/edges that destroy all paths from u to v, if such a set exists). They show how to precompute all such answers, in the case of vertex-disjoint paths, in $O((nk)^\omega)$ time. This weaker version of the problem can also be solved with respect to edge-disjoint paths by computing the value of all-pairs min-cut in $O(m^\omega)$ time [18] for general graphs and in $\widetilde{O}(n^2)$ time for planar graphs [41].

The related problem of sensitivity oracles for strongly connected components (SCCs) was considered by Georgiadis et al. [29]. Specifically they presented a 1-sensitivity oracle with $O(m)$ size and preprocessing time, that answers various SCC queries under the presence of single edge or vertex failures in asymptotically optimal time. For instance they can test whether two vertices u, v are in the same SCC in $G - x$ in constant time, for query vertices u, v and failed vertex or edge x, or they can report the SCCs of $G - x$ in $O(n)$ time. Baswana et al. [9], showed an f-sensitivity oracle with $O(2^f n^2)$ space, and $O(mn^2)$ preprocessing time, that can report the SCCs of $G - F$ in $O(2^f n \operatorname{polylog} n)$ time, where F is a set of failed vertices or edges, for $|F| \leq f$.

Reachability queries under edge or vertex failures can be also answered using more powerful sensitivity oracles for shortest paths or approximate shortest paths. For planar directed graphs there is no known $o(n^2)$ space all-pairs distance sensitivity oracle with $O(\operatorname{polylog} n)$ query time. Baswana et al. [10] presented a single-source reachability oracle under single edge or vertex failures with $O(n \operatorname{polylog} n)$ space and construction time, that can report the length of the shortest path from s to v in $G - x$ in $O(\log n)$ time, for query vertex v and a failed vertex or edge x. They extend their construction to work for the all-pairs variant of the problem in $O(n^{3/2} \operatorname{polylog} n)$ preprocessing time and size of the oracle, and answer queries in $O(\sqrt{n} \operatorname{polylog} n)$ time. Later on Charalampopoulos et al. [17] presented improved sensitivity oracles for all-pairs shortest paths on planar graphs. Their sensitivity oracle also handles multiple failures at the expense of a worse trade-off between size and query time. For the all-pairs version of the problem, their oracles have significantly worse bounds compared to the best known exact distance oracles (without failures) for planar graphs. The best known exact distance oracle for planar graphs with $O(\operatorname{polylog} n)$ query time was presented recently by Charalampopoulos et al. [16]; it uses $O(n^{1+\epsilon})$ space and has $O(n^{(3+\epsilon)/2})$ construction time. However, we note that distance sensitivity oracles cannot answer 2-reachability queries.

Finally, we could in principle handle f-sensitivity queries with a fully dynamic reachability or shortest-paths oracle with good worst-case update and query bounds [21, 45]. However, not very surprisingly, this approach rarely yields better bounds than f-sensitivity solutions tailored to handle only batches of failures.

In summary, there exist efficient fault-tolerant reachability oracles for general digraphs with preprocessing time, size, and query time comparable to the fastest known static reachability oracles (without failures). This is the case, e.g., for the fault-tolerant reachability oracle and 2-reachability oracle of [26], which almost matches the $O(\min\{mn, n^\omega\})$ bound for computing reachability without failures, and for the data structure for SCCs under failures of [29], which has linear construction time and space, and it is capable of answering queries in asymptotically optimal time. However, and somehow surprisingly, such efficient fault-tolerant reachability oracles and 2-reachability oracles, i.e., oracles with preprocessing time, size, and query time com-

parable to the fastest known static oracles (without failures) are not known in the case of any basic problem on *planar directed graphs*. Such oracles would implement some of the most important functionalities of dominator trees, but for all possible sources at once. Since dominator trees have several applications, including applications in planar digraphs, it seems quite natural to ask whether such oracles exist.

An additional motivating factor for studying our problem is the large gap in known and possible bounds between undirected and directed graphs for related problems. For the case of undirected graphs, there exist nearly optimal f-sensitivity oracles for answering connectivity queries under edge and vertex failures. Duan and Pettie [23] presented a near-optimal preprocessing $O(n \log n)$-space f-sensitivity oracle that, for any set F of up to f edge-failures their oracle, spends $O(f \log f \log \log n)$ time to process the failed edges and then can answer connectivity queries in $G \setminus F$ in time $O(\log \log n)$ per query. This result is nearly optimal also for the case of planar undirected graphs. In the same paper, the authors also present near-optimal bounds for the case of vertex-failures. For general directed graphs, it is clear that sensitivity oracles for all-pairs reachability cannot achieve bounds anywhere close to the known bounds for sensitivity oracles for connectivity in undirected graphs. While answering connectivity queries in undirected graphs is a much simpler task than answering reachability queries in digraphs, an intriguing question is whether there exists a general family of directed graphs that admits fault-tolerant reachability oracles with bounds close to the known results for sensitivity connectivity oracles in undirected graphs, even for the case of a single failure.

Our results. We answer the questions posed above affirmatively by presenting the first *near-optimal* – in terms of both time and space – oracle handling *all-pair-type* queries for *directed* planar graphs and supporting any *single vertex or single edge failure*. Specifically, we prove the following.

THEOREM 1.1. *Let G be a planar digraph. There exists an $O(n \log n)$-space data structure answering queries of the form "is there a $u \to v$ path in $G - x$", where $u, v, x \in V$, in $O(\log n)$ time. The data structure can be constructed in $O(n \log^2 n / \log \log n)$ time.*

We remark that previous data structures handling failures in $O(\text{polylog}\, n)$ time either work only for the single-source version of the problem (see dominator trees, or [19] for two failures), or work only on undirected graphs (see, e.g., [22, 23] for oracles for general graphs, and [2, 12] for planar graphs), or achieve nearly linear space only for dense graphs [51]. It is worth noting that for planar digraphs vertex failures are generally more challenging than edge failures, since, whereas one can easily reduce edge failures to vertex failures, the standard opposite reduction of splitting a vertex into an in- and an out-vertex does not preserve planarity.

In order to achieve our 1-sensitivity oracle, we develop two new data structures that also work on general digraphs and can be of independent interest:

- Given a digraph G and a directed path P of G, we present a linear-space data structure that, after preprocessing G in $O(n + m \log m / \log \log m)$ time, can answer whether there exists a path from u to v in $G - x$ passing through a vertex of P, for any query vertices $u, v \in V$ and $x \in V \setminus V(P)$. In a sense, this result generalizes the dominator tree (a tree T rooted at a source-vertex s such that a vertex t is reachable from s in $G - x$ if and only if it is reachable in $T - x$) in the following way. Note that a pair of dominator trees from and to s can be used to support queries of the form "is u reachable from v through a "hub" s in $G - x$?", where $u, v, x \neq s$ are all query vertices and s is fixed. Our data structure allows to replace the single hub s with any number of hubs that form a directed path, provided that these hubs cannot fail. Since dominator trees have numerous applications, as discussed before, we believe that our generalization can find other applications as well.
- We show that given a digraph G and an assignment of real-valued labels to the vertices of G, in $O(m + n(\log n \log \log n)^{2/3})$ time one can construct a linear-space data structure that supports $O(1)$-time queries of the form "what is the largest/smallest label in the strongly connected component of u in $G - x$?", for any pair of query vertices $u, x \in V$.

By suitably extending our 1-sensitivity oracle, we obtain a nearly optimal 2-reachability oracle for planar digraphs, summarized as follows.

THEOREM 1.2. *In $O(n \log^{6+o(1)} n)$ time one can construct an $O(n \log^{3+o(1)} n)$-space data structure supporting the following queries in $O(\log^{2+o(1)} n)$ time. For $u, v \in V(G)$, either find some separating vertex $x \notin \{u, v\}$ lying on all $u \to v$ paths in G, or declare v 2-reachable from u.*

Our 1-sensitivity and 2-reachability oracles, combined, extend several supported operations of dominator trees to the all-pairs version of the problem. For example, for any u, v, our data structures can identify the set X of all vertices (or edges) that appear

in all paths from u to v in $O(|X|\log^{2+o(1)} n)$ time by executing $O(|X|)$ 2-reachability queries. That is, a 2-reachability query returns a vertex x that appears in all paths from u to v, and all other vertices in $X \setminus x$ (if any) appear either in all paths from u to x or in all paths from x to v, which we can identify by recursively executing 2-reachability queries from u to x and from x to v, and eventually reconstruct the set X.

Whereas we achieve $\widetilde{O}(n)$ preprocessing time for both our oracles, the main conceptual challenge lies in obtaining near-optimal space. However, efficient construction of the used data structures, especially our generalization of the dominator tree, proved to be a highly non-trivial task as well.

Overview of our 1-sensitivity oracle. As already discussed, it is sufficient to build a 1-sensitivity oracle for single vertex failures as the case of edge failures reduces to the case of vertex failures by applying edge-splitting on the edges of the graph (i.e., replacing each edge zw by a new vertex c, and two edges zc and cw). Since $m = O(n)$ in planar graphs, this process does not increase the number of vertices or edges significantly. The initial step of our approach is the use of the basic[5] hierarchical decomposition approach introduced by Thorup [50]. For the problem of constructing a reachability oracle (with no failures) this initial phase allows one to focus on the following problem, at the expense of an increase by a factor $O(\log n)$ in the preprocessing time, the size, and the query time of the constructed oracle. Given a graph G and a directed path P of G, construct a data structure that answers efficiently whether there is a path from u to v containing any vertex of P, for any two vertices u and v. It is rather easy to obtain such a data structure with linear space and preprocessing time that can answer the required queries in constant time.

Although we use the decomposition phase of [50] as an initial step in our approach, the main difficulty in our problem is to build a data structure that can efficiently answer whether there exists a path from u to v in $G-x$ that uses a vertex of a path P. In the presence of failed vertices this becomes much more challenging, compared to reachability queries with no failures, as the set of vertices of P that are reachable from (or can reached by) a vertex w, might be different under failures of different vertices. Additionally, the case where the failed vertex appears on P disconnects the path into two subpaths which we need to query. We cannot afford to simply preprocess all such subpaths, as there can be as many as $|P| = \Theta(n)$ of those for all possible failures of vertices on the path P. We overcome these problems by using new insights, developing new supporting data structures and further exploiting planarity.

We distinguish two cases depending on whether the failed vertex appears on P or not. For each path P, we preprocess the graph to handle each case separately.

To deal with the case where the failed vertex lies outside of P, we identify, for the query vertices u, v and failed vertex x, the earliest (resp., latest) vertex on P that u can reach (resp., that can reach v) in $G - x$. Call this vertex $first^P_{G-x}(u)$ (resp., $last^P_{G-x}(v)$). Given these vertices it suffices to test whether $first^P_{G-x}(u)$ appears no later than $last^P_{G-x}(v)$ on P. Recall from our previous discussion that both these vertices depend on the failed vertex x. A useful notion throughout the paper is the following. A path Q between any two vertices w, z is called a *satellite* path (with respect to P) if no vertex of Q other than w, z is a vertex of P, i.e., if $V(Q) \cap V(P) \subseteq \{w, z\}$. On a very high level, we first develop a near-optimal data structure that can identify in constant time for each vertex v the latest vertex $v' \in V(P)$ that has a satellite path to v in $G-x$. The performance of the data structure relies on the efficient constructions of dominator trees [6, 13, 24, 25] and their properties, as well as dynamic orthogonal range-searching data structures [15]. Given v', we then show that $last^P_{G-x}(u)$ is the latest vertex on P that is in the same SCC as v' in $G - x$. We generalize the problem of computing such a vertex to the mentioned problem of efficiently finding a maximum-labeled vertex in the SCC of v in $G - x$, where $v, x \in V$ are query parameters. For this problem we develop a near-optimal data structure with $O(1)$ query time. Finally, we proceed with computing $first^P_{G-x}(u)$ analogously. The query time in this case is $O(1)$.

In order to handle the case when the failed vertex x is on P we further exploit planarity. We observe that by modifying the basic recursive decomposition of Thorup to use fundamental cycle separators instead of root path separators (this modification was previously used in e.g. [2, 43]), we can assume that the endpoints of P in fact lie on a single face of G. This additional assumption enables us to achieve two important things. First we show that after linear preprocessing, in $O(\log n)$ time we can in fact compute the earliest (latest) vertex of *any subpath of* P reachable from (that can reach) a query vertex $v \in V$ by a satellite path. Here the subpath of interest is also a query parameter. Moreover, we

[5] Thorup [50] also presented a more involved data structure that allowed him to reduce query time to $O(1)$ while maintaining the preprocessing time $O(n \log n)$. However, this data structure significantly differs from his basic $O(\log n)$-query data structure in the fact that one needs to represent reachability through separating directed paths P "globally" in the entire graph G, as opposed to only representing reachability through P "locally" in the subgraph $H \subseteq G$ we recurse on. It is not clear if this more sophisticated approach can be extended to handle vertex failures.

introduce a concept of a detour of x to be a path that starts earlier and ends later than x on P. A minimal detour of x is a detour that does not simultaneously start earlier (on P) and end later (on P) that any other detour of x. We use planarity to show that there can be at most two significantly different minimal detours of any vertex $x \in V(P)$. Consequently, we show a linear time algorithm for finding the two minimal detours for each vertex $x \in V(P)$. Finally, we consider several possible scenarios of how the requested $u \to v$ path in $G - x$ can interact with P and x. In all of these cases we show that there exists a certain canonical path consisting of $O(1)$ subpaths that are either satellite paths, minimal detours, or subpaths of P. This allows us to test for existence of a $u \to v$ path in $G - x$ with only $O(1)$ queries to the obtained data structures.

Even though we are not able to reduce the query time in the case when x lies on the path P to constant, this turns out not to be a problem. This is because in order to answer a "global" query, we need only one such query and $O(\log n)$ constant-time queries to the data structures when x is out of the path. So, the query time of the whole reachability data structure is $O(\log n)$.

2-reachability oracle. The 2-reachability oracle is obtained by both extending and reusing the 1-sensitivity oracle. It is known that if the graph is strongly connected, then checking whether vertex v is 2-reachable from u can be reduced to testing whether v is reachable from u under only $O(1)$ single-vertex failures, which are easily computable from the dominator tree from an arbitrary vertex of the graph [26]. This observation alone would imply a 2-reachability oracle for strongly connected planar digraphs within the time/space bounds of our 1-sensitivity oracle.

However, for graphs with k strongly connected components, a generalization of this seems to require information from as many as $\Theta(k)$ dominator trees to cover all possible (u, v) query pairs.

Nevertheless, we manage to overcome this problem by using the same recursive approach, and carefully developing the "existential" analog of the 1-sensitivity data structures handling failures either outside the separating path P, or on the separating path P when P's endpoints lie on a single face of the graph. There is a subtle difference though; in the 2-reachability oracle the recursive call is made only when P contains no vertex that lies on a $u \to v$ path; once we find a path P containing a vertex that lies on any $u \to v$ path we make no further recursive calls.

In the case of failures outside the path P, we prove that $O(1)$ single-failure queries are sufficient to decide whether there exists $x \notin V(P)$ that destroys all $u \to v$ paths in G. Even though we use $\Theta(n)$ dominator trees to encode the information about which single-failure queries we should issue to the 1-sensitivity oracle (for any pair u, v of query vertices), our construction ensures that the total size of these dominator trees is $O(n)$.

When searching for vertices $x \in V(P)$ that lie on all $u \to v$ paths in G, the reduction to asking few 1-sensitivity queries does not to work. Instead, we take a substantially different approach. Roughly speaking, we simulate the single-failure query procedure developed in the 1-sensitivity oracle for all failing $x \in V(P)$ at once. We prove that deciding if for any such x the query to the 1-sensitivity oracle would return false can be reduced to a generalization of a 4-dimensional orthogonal range reporting problem, where the topology of one of the dimensions is a tree as opposed to a line. A simple application of heavy-path decomposition [46] allows us to reduce this problem to the standard 4-d orthogonal range reporting problem [36] at the cost of $O(\log n)$-factor slowdown in the query time compared to the standard case. This turns out to be the decisive factor in the $O(\log^{2+o(1)} n)$ query time and $O(n \log^{3+o(1)} n)$ space usage of our 2-reachability oracle.

Organization of the paper. In Section 2 we fix the notation and recall some important properties of planar graphs. In Section 3 we give a quite detailed overview of Thorup's construction and explain how we modify it to suit our needs. In Section 4 we show how to make the reachability data structure from Section 3 to support vertex failures. Apart from that, in Section 4 we also state and explain the usage of our main technical contributions – Theorems 4.1, 4.2 and 4.3 – and give a more detailed overview of how they are achieved. The detailed proofs of Theorems 4.1 and 4.2 can be found in Sections 6 and 7, respectively. In Section 5 we review some useful properties of dominator trees. Finally, we briefly discuss the 2-reachability data structure in Section 8. Due to lack of space, some technical details and proofs are omitted from this extended abstract and are deferred to the full-version of this paper.

2 Preliminaries

In this paper we deal with *directed* simple graphs (*digraphs*). We often deal with multiple different graphs at once. For a graph G, we let $V(G)$ and $E(G)$ denote the vertex and edge set of G, respectively. If G_1, G_2 are two graphs, then $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$ and $G_1 \cap G_2 = (V(G_1) \cap V(G_2), E(G_1) \cap E(G_2))$. Even though we work with digraphs, some notions that we use, such as connected components, or spanning trees, are only defined for undirected graphs. Whenever we use these notions with respect to a digraph, we ignore the directions of the edges.

Let $G = (V, E)$ be a digraph. We denote by $uv \in E$

the edge from u to v in G. A graph G' is called a subgraph of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. For $S \subseteq V(G)$, we denote by $G[S]$ the *induced subgraph* $(S, \{uv : uv \in E(G), \{u,v\} \subseteq S\})$. Given a digraph G, we denote by G^R the digraph with the same set of vertices as G and with all of the edges reversed compared to the orientation of the corresponding edges in G. That is, if G contains an edge uv, G^R contains an edge vu and vice versa. We say that G^R is the *reverse graph* of G. For any $X \subseteq V$ we define $G - X = G[V \setminus X]$. For $x \in V$, we write $G - x$ instead of $G - \{x\}$.

A *path* $P \subseteq G$ is a subgraph whose edges $E(P)$ can be ordered $e_1, \ldots, e_k$ such that if $e_i = u_i v_i$, then for $i = 2, \ldots, k$ we have $u_i = v_{i-1}$. Such P is also called a $u_1 \to v_k$ path. We also sometimes write $P = u_1 u_2 \ldots u_k$. A path P is *simple* if $u_i \neq u_j$ for $i \neq j$. For a simple path $P = u_1 \ldots u_k$ we define an order $\prec_P$ on the vertices of P. We write $v \prec_P w$ for $v = u_i$ and $w = u_j$ if $i < j$. If $u \prec_P v$, we also say that u is *earlier* on P than v, whereas v is *later* than u on P. We denote by $P[u_i, u_j]$ the unique subpath of P from u_i to u_j. Similarly, let $P(u_i, u_j)$ be the subpath of P from the vertex following u_i to the vertex preceding u_j on P. For $u, v \in V$, we say that v is *reachable* from u if there exists a $u \to v$ path in G. We call v *2-reachable* from u if there exist two internally vertex-disjoint $u \to v$ paths in G. u and v are *strongly connected* if there exist both paths $u \to v$ and $v \to u$ in G. A *non-oriented path* $P' \subseteq G$ is a subgraph that would become a path if we changed the directions of some of its edges. If $P_1 = u \to v$ is a (potentially non-oriented) path and $P_2 = v \to w$ is a (non-oriented) path, their concatenation $P_1 P_2 = u \to w$ is also a (non-oriented) path.

We sometimes use trees, which can be rooted or unrooted. If a tree T is rooted, we denote by $T[v]$ the *subtree* of T rooted in one of its vertices v. We denote by $T[u,v]$ the path between u and v on T, by $T(u,v]$ (resp., $T[u,v)$) the path between u and v on T, excluding u (resp., excluding v). Analogously, we use $T(u,v)$ to denote the path between u and v on T, excluding u and v. For any tree T, we use the notation $t(v)$ to refer to the parent of node v in T. If v is the root of the tree, then $t(v) = v$. To avoid cumbersome notation we sometimes write $w \in T[v]$ when we formally mean $w \in V(T[v])$, $w \in T[u,v]$ when $w \in V(T[u,v])$ and so on. If $T \subseteq G$ is a spanning tree of a connected graph G, then for any $uv \in E(G) \setminus E(T)$ the *fundamental cycle* of uv wrt. T is a subgraph of G that consists of the unique non-oriented simple $u \to v$ path in T and the edge uv.

Plane graphs. A *plane embedding* of a graph is a mapping of its vertices to distinct points and of its edges to non-crossing curves in the plane. We say that G is *plane* if some embedding of G is assumed. A *face* of a connected plane G is a maximal open connected set of points that are not in the image of any vertex or edge in the embedding of G. There is exactly one *unbounded* face. The *bounding cycle* of a bounded (unbounded, respectively) face f is a sequence of edges bounding f in clockwise (counterclockwise, respectively) order. Here, we ignore the directions of edges. An edge can appear in a bounding cycle at most twice. An embedding of a planar graph (along with the bounding cycles of all faces) can be found in linear time [34]. A plane graph G is *triangulated* if all its faces' bounding cycles consist of 3 edges. Given the bounding cycles of all faces, a plane graph can be triangulated by adding edges inside its faces in linear time.

A graph G' is called a *minor* of G if it can be obtained from G by performing a sequence of edge deletions, edge contractions, and vertex deletions. If G is planar then G' is planar as well. By the Jordan Curve Theorem, a simple closed curve C partitions $\mathbb{R}^2 \setminus C$ into two connected regions, a bounded one B and an unbounded one U. We say that a set of points P is *strictly inside* (*strictly outside*) C if and only if $P \subseteq B$ ($P \subseteq U$, respectively). P is *weakly inside* (*weakly outside*) iff $P \subseteq B \cup C$ ($P \subseteq U \cup C$, respectively). If G is plane then the fundamental cycle of uv corresponds to a simple closed curve in the plane. We often identify the fundamental cycle with this curve.

LEMMA 2.1. (E.G., [38]) *Let $G = (V, E)$ be a connected triangulated plane graph with n vertices. Let T be a spanning tree of G. Let $w : V \to \mathbb{R}_{\geq 0}$ be some assignment of weights to vertices of G. Set $W := \sum_{v \in V} w(v)$. Suppose that for each $v \in V$ we have $w(v) \leq \frac{1}{4}W$.*

There exists such $uv \in E \setminus E(T)$ that the total weight of vertices of G lying strictly on one side of the fundamental cycle of uv wrt. T is at most $\frac{3}{4}W$. The edge uv can be found in linear time.

3 The Reachability Oracle by Thorup

In this section we describe the basic reachability oracle of Thorup [50] that we will subsequently extend to support single vertex failures. His result can be summarized as follows.

THEOREM 3.1. ([50]) *Let G be a directed planar graph. One can preprocess G in $O(n \log n)$ time so that arbitrary reachability queries are supported in $O(\log n)$ time.*

DEFINITION 3.1. ([50]) *A 2-layered spanning tree T of a digraph H is a rooted spanning tree such that any non-oriented path in T from the root is a concatenation of at most two directed paths in H.*

We will operate on graphs with some *suppressed vertices*. Those suppressed vertices will guarantee cer-

tain useful topological properties of our plane graphs, but will otherwise be forbidden to be used from the point of view of reachability. In other words, when answering reachability queries we will only care about directed paths that do not go through suppressed vertices.

REMARK 3.1. *Our description differs from that of Thorup in the fact that we allow $O(1)$ suppressed vertices (where Thorup needed only one that was additionally always the root of the spanning tree, and no balancing of suppressed vertices was needed), and we insist on using simple cycle separators (whereas Thorup's separators consisted of two root paths). Whereas using cycle separators does not make any difference for reachability, we rely on them when we handle single vertex failures.*

LEMMA 3.1. ([50]) *Let $G = (V, E)$ be a connected digraph. In linear time we can construct digraphs $G_0, \ldots, G_{k-1}$, where $G_i = (V_i, E_i)$, and $A_i \subseteq V_i$ is a set of suppressed vertices, $|A_i| \leq 1$, their respective spanning trees T_i, and a function $\iota : V \to \{0, \ldots, k-1\}$ such that:*

1. *The total number of edges and vertices in all G_i is linear in $|V|+|E|$.*
2. *Each G_i is a minor of G, and $G_i - A_i \subseteq G$.*
3. *For any $u, v \in V$ and any directed path $P = u \to v$, $P \subseteq G$ if and only if $P \subseteq G_{\iota(u)} - A_{\iota(u)}$ or $P \subseteq G_{\iota(u)-1} - A_{\iota(u)-1}$.*
4. *Each spanning tree T_i is 2-layered.*

The basic reachability data structure of Thorup [50] can be described as follows. Observe that by Lemma 3.1 v is reachable from u in G (where $u, v \in V$) if and only if it is reachable from u in either $G_{\iota(u)} - A_{\iota(u)}$ or $G_{\iota(u)-1} - A_{\iota(u)-1}$. Moreover, all the graphs G_i in Lemma 3.1, being minors of G, are planar as well. Therefore, by applying Lemma 3.1, the problem is reduced to the case when (1) a planar graph $G = (V, E)$ has a 2-layered spanning tree T, (2) we are only interested in reachability without going through some set of $O(1)$ (in fact, at most 5, as we will see) suppressed vertices A of G. Under these assumptions, the problem is solved recursively as follows.

If G has constant size, or has only suppressed vertices, we compute its transitive closure so that queries are answered in $O(1)$ time. Otherwise, we apply recursion. We first temporarily triangulate G by adding edges and obtain graph G^{Δ}. Note that T is a 2-layered spanning tree of G^{Δ} as well.

FACT 3.1. *Let $G = (V, E)$ be a connected plane digraph. Let T be a 2-layered spanning tree of G rooted at r. Let $uv \in E \setminus E(T)$. Let V_1 (resp., V_2) be the subset of V strictly inside (resp., strictly outside) the fundamental cycle C_{uv} of uv wrt. T. Let S_{uv} be the non-oriented path $C_{uv} - uv$.*

Let G' (T') be obtained from G (T) by contracting S_{uv} into a single vertex r'. Then for $i = 1, 2$, $T'[V_i \cup \{r'\}]$ is a 2-layered spanning tree of $G'[V_i \cup \{r'\}]$.

Using Lemma 2.1, we compute in linear time a balanced fundamental cycle separator C_{ab}, where $ab \in E(G^{\Delta}) \setminus E(T)$. Let A be the set of suppressed vertices of G. The weights assigned to vertices depend on the size of A: if $|A| \leq 4$ then we assign weights 1 uniformly to all vertices of G, and otherwise we assign unit weights to the vertices of A only (the remaining vertices V get weight 0). Let the non-oriented path $S_{ab} = C_{ab} - ab$ be called *the separator*. Let V_1, V_2, G', T' and r' be defined as in Fact 3.1.

Note that for any $u, v \in V$, a $u \to v$ path in $G - A$ can either go through a vertex of S_{ab}, or is entirely contained in exactly one $G[V_i] - A$, for which $\{u, v\} \subseteq V_i$ holds. We deal with these two cases separately. Since $G[V_i] \subseteq G'[V_i \cup \{r'\}]$, queries about a $u \to v$ path not going through S_{ab} can be delegated to the data structures built recursively on each $G'[V_i \cup \{r'\}]$ with suppressed set $A_i = (A \cap V_i) \cup \{r'\}$. By Fact 3.1, $G'[V_i \cup \{r'\}]$ has a 2-layered spanning tree $T'[V_i \cup \{r'\}]$ rooted in r'. Moreover, a path $u \to v$ exists in $G[V_i] - A$ if and only if a path $u \to v$ not going through a suppressed set A_i exists in $G'[V_i \cup \{r'\}]$. Note that since S_{ab} does not necessarily go through any of the vertices of A, A_i might be larger than A by a single element – this is why balancing of suppressed vertices is needed. Hence, indeed recursion can be applied in this case.

Paths $u \to v$ going through $V(S_{ab})$ in $G - A$ are in turn handled as follows. Since T is 2-layered, S_{ab} can be decomposed into at most 4 edge-disjoint directed paths in G. Consequently, $S_{ab} - A$ can be split into at most $4 + |A|$ edge-disjoint directed paths in $G - A$. Next, we take advantage of the following lemma.

LEMMA 3.2. ([47, 50]) *Let H be a directed graph and let $P \subseteq H$ be a simple directed path. Then, in linear time we can build a data structure that supports constant-time queries about the existence of a $u \to v$ path that necessarily goes through $V(P)$.*

The above lemma is based on the following simple fact that will prove useful later on.

FACT 3.2. ([47, 50]) *Let H be a digraph. Let $P \subseteq H$ be a simple directed path. Denote by $first_H^P(u)$ the earliest vertex of P* reachable from *u in H. Denote by $last_H^P(v)$ the latest vertex of P that* can reach *v in H. Then there exists a $u \to v$ path in H going through $V(P)$ iff $first_H^P(u) \preceq_P last_H^P(v)$.*

Consequently, by building at most $4 + |A|$ data structures of Lemma 3.2, we can check whether v is reachable from u through $V(S_{ab})$ in $G - A$ in $O(1 + |A|)$ time.

LEMMA 3.3. *At each recursive call, the size of the suppressed set A is at most 5.*

LEMMA 3.4. *The depth of the recursion is $O(\log n)$.*

At each recursive call of the data structure's construction procedure we use only linear preprocessing time. Observe that at each recursive level the total number of vertices in all graphs of that level is linear: there are at most n vertices that are not suppressed and each of these vertices resides in a unique graph of that level. Since each graph has to have at least one non-suppressed vertex, the number of graphs on that level is also at most n. Therefore, given that $|A| = O(1)$ in every recursive call, the sum of sizes of the graphs on that level is $O(n)$. By Lemma 3.4, we conclude that the total time spent in preprocessing is $O(n \log n)$.

In order to answer a query whether an $u \to v$ path exists, we only need to query $O(\log n)$ data structures of Lemma 3.2 handling queries about reachability through a directed path.

4 Reachability Under Failures

In this section we explain how to modify the data structure discussed in Section 3 to support queries of the form "is v reachable from u in $G - x$?", where $u, v, x \in V$ are distinct query parameters.

First recall that, by Lemma 3.1, each path $P = u \to v$ exists in G if and only if it exists in either $G_{\iota(u)} - A_{\iota(u)}$ or $G_{\iota(u)-1} - A_{\iota(u)-1}$. This, in particular, applies to paths P avoiding x. As a result, v is reachable from u in $G - x$ if and only if v is reachable from u in either $G_{\iota(u)} - A_{\iota(u)} - x$ or $G_{\iota(u)-1} - A_{\iota(u)-1} - x$. That being said, we can again concentrate on the case when G has a 2-layered spanning tree and we only care about reachability in $G - A$, where $A \subseteq V$ has size $O(1)$.

We follow the recursive approach of Section 3. The only difference lies in handling paths in $G - A$ going through the separator S_{ab}. Ideally, we would like to generalize the data structure of Lemma 3.2 so that single-vertex failures are supported. However, it is not clear how to do it in full generality. Instead, we show two separate data structures, which, when combined, are powerful enough to handle paths going through a *cycle* separator.

Recall that $S_{ab} - A$ can be decomposed into $O(1)$ simple directed paths $P_1, \ldots, P_k$ in $G - A$ that can only share endpoints. Denote by $D \subseteq V(S_{ab})$ the set of endpoints of these paths. Suppose we want to compute whether there exists a $u \to v$ path Q in $G - A - x$ that additionally goes through some vertex of S_{ab}. We distinguish several cases.

1. Suppose that $x \in D$. Recall that there exist only $O(1)$ such vertices x. Moreover, $S_{ab} - A - x$ can be decomposed into $O(1)$ simple paths in $G - A - x$. Hence, for each such x, we build $O(1)$ data structures of Lemma 3.2 to handle reachability queries through S_{ab} in $G - A - x$ exactly as was done in Section 3. The preprocessing is clearly linear and the query time is $O(1)$ in this case.

2. Now suppose $x \notin D$ and there exists such P_i that $V(Q) \cap V(P_i) \neq \emptyset$ and $x \notin V(P_i)$. Paths of this kind are handled using the following theorem (for $G := G - A$, $P := P_i$) proved in Section 6.

THEOREM 4.1. *Let G be a digraph and let $P \subseteq G$ be a simple directed path. In $O\left(m \frac{\log n}{\log \log n}\right)$ time one can build a linear-space data structure that can decide, for any $u, v \in V$, and $x \notin V(P)$, if there exists a $u \to v$ path going through $V(P)$ in $G - x$. Such queries are answered in $O(1)$ time.*

3. The last remaining case is when none of the above cases apply. This means that $x \notin D$ and for all P_i either Q does not go through $V(P_i)$ or $x \in V(P_i)$. Since $V(P_i) \cap V(P_j) \subseteq D$ for all $j \neq i$, there can be at most one such i that $x \in V(P_i)$. For all $j \neq i$, Q does not go through $V(P_j)$. On the other hand, since Q goes through S_{ab}, it in fact has to go through $V(P_i)$.

LEMMA 4.1. *Let $\bar{V_i} = V \setminus V(S_{ab}) \cup V(P_i)$. Then the endpoints of P_i lie on a single face of the component of $G[\bar{V_i}] - A$ that contains P_i.*

By Lemma 4.1, the last case can be handled using the following theorem proved in Section 7.

THEOREM 4.2. *Let G be a plane digraph and let $P \subseteq G$ be a simple path whose endpoints lie on a single face of G. In linear time one can build a data structure that can compute in $O(\log n)$ time whether there exists a $u \to v$ path going through $V(P)$ in $G - x$, where $u, v \in V$ and $x \in V(P)$.*

In order to prove Theorem 1.1, first note that the preprocessing time of our data structure is $O(n \log^2 n / \log \log n)$, since all the data structures of Lemma 3.2, and Theorems 4.1 and 4.2 can be constructed in $O(n \log n / \log \log n)$ time for each input graph of the recursion. Since all these data structures use only linear space, the total space used is $O(n \log n)$. To analyze the query time, observe that similarly to Section 3, we query $O(\log n)$ "reachability through a path"

data structures. All of them, except of the data structure of Theorem 4.2, have constant query time. However, the data structure of Theorem 4.2 is only used to handle the case when $x \in V(S_{ab})$. Observe that for any $w \in V$ there is no more than one node in the recursive data structure such that w is a vertex of the respective separator S_{ab}. As a result, we only need to perform a single query to a data structure of Theorem 4.2.

Finding the maximum label in an SCC under failures. In order to achieve constant query time in Theorem 4.1, we make use of the following Theorem, proved in the full version, which we believe might be of independent interest.

THEOREM 4.3. *Given a digraph G and an assignment $f : V \to \mathbb{R}$ of labels to the vertices of G, we can preprocess G in $O(m+n(\log n \log\log n)^{2/3})$ time, so that the following queries are supported in $O(1)$ time. Given $x, v \in V(G)$, find a vertex with the maximum label in the SCC of v in $G - x$.*

In order to obtain the general data structure of Theorem 4.3, we exploit the framework developed in [29]. We extend this framework to identify the minimum/maximum label in any single SCC in $G - x$, except of the one SCC that contains an arbitrary but fixed vertex s. To deal with this case, we show how we can precompute the maximum value in the SCC of s in $G - x$, for all possible failures x, via a reduction to a special case of offline two-dimensional range minimum queries, which can be further reduced to decremental one-dimensional range minimum queries [52].

5 Dominators in Directed Graphs

In this section we review the dominator trees and their properties, which we exploit extensively later on.

A *flow graph* is a directed graph with a start vertex s, where all vertices are reachable from s. Let G_s be a flow graph with start vertex s. A vertex u is a *dominator* of a vertex v (u *dominates* v) if every path from s to v in G_s contains u; u is a *proper dominator* of v if u dominates v and $u \neq v$. Let $dom(v)$ be the set of dominators of v. Clearly, $dom(s) = \{s\}$ and for any $v \neq s$ we have that $\{s, v\} \subseteq dom(v)$: we say that s and v are the *trivial dominators* of v in the flow graph G_s. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree [3, 40], known as the *dominator tree* D: u dominates v if and only if u is an ancestor of v in D. If $v \neq s$, the parent of v in D, denoted by $d(v)$, is the *immediate dominator* of v: it is the unique proper dominator of v that is dominated by all proper dominators of v. Similarly, we can define the dominator relation in the flow graph G_s^R, and let D^R denote the dominator tree of G_s^R. We also denote the immediate dominator of v in G_s^R by $d^R(v)$. Lengauer and Tarjan [39] presented an algorithm for computing dominators in $O(m\alpha(m, n))$ time for a flow graph with n vertices and m edges, where α is a functional inverse of Ackermann's function [49]. Subsequently, several linear-time algorithms were discovered [6, 13, 24, 25]. We apply the tree notation introduced earlier on when referring to subtrees and paths of dominator trees.

LEMMA 5.1. ([28]) *Let G be a flow graph with start vertex s and let $v \neq s$. Let w be any vertex that is not a descendant of v in D. All simple paths in G from w to any descendant of v in D must contain v.*

LEMMA 5.2. ([31]) *For each $zw \in E(G_s)$, where $z \neq s$, z is a descendant of $d(w)$ in D.*

LEMMA 5.3. *Let G be a flow graph with start vertex s and let $v \neq s$ be a non-leaf in the dominator tree D of G. There exists a simple path in $G[D[v]]$ from v to any $w \in D[v]$.*

6 Proof of Theorem 4.1

In this section we prove the following theorem.

THEOREM 4.1. *Let G be a digraph and let $P \subseteq G$ be a simple directed path. In $O\left(m\frac{\log n}{\log\log n}\right)$ time one can build a linear-space data structure that can decide, for any $u, v \in V$, and $x \notin V(P)$, if there exists a $u \to v$ path going through $V(P)$ in $G - x$. Such queries are answered in $O(1)$ time.*

Here, we do not assume that the underlying graph G is planar. Let $n = |V(G)|$ and $m = |E(G)|$. Wlog. assume $n \leq m$. Let $P = p_1 p_2 \dots p_\ell$ be the directed path we consider. To answer our queries at hand for a failing vertex $x \notin V(P)$, we use the same approach as in Lemma 3.2: observe that since $x \notin V(P)$, by Fact 3.2, u can reach v through P in $G - x$ if and only if $first^P_{G-x}(u) \preceq_P last^P_{G-x}(v)$. In what follows we only show how to compute $last^P_{G-x}(v)$ efficiently, since $first^P_{G-x}(u) = last^{P^{\mathrm{R}}}_{G^{\mathrm{R}}-x}(u)$ and thus it can be computed by proceeding identically on the reverse graph G^R. For brevity, in the remaining part of this section we omit the superscript/subscript P and write $last_{G-x}$ instead $last^P_{G-x}$, $last^*_G$ instead $last^{*P}_G$, $\prec$ instead of $\prec_P$, etc.

DEFINITION 6.1. *We call a simple directed path $Q = e \to f$ of G* satellite *if it does not go through $V(P)$ as intermediate vertices, i.e., $V(Q) \cap V(P) \subseteq \{e, f\}$.*

For any $v \in V \setminus V(P)$, we also denote by $last^*_{G-x}(v)$ the latest vertex of P that can reach v in $G - x$ by a

satellite path, if such a vertex exists. For $v \in V(P)$, set $last^*_{G-x}(v) = v$. Having computed $last^*_{G-x}(v)$, we then compute $last^P_{G-x}(v)$ using the following lemma.

LEMMA 6.1. *$last_{G-x}(v)$ is the latest vertex of P in the SCC of $last^*_{G-x}(v)$ in $G - x$.*

Proof. Since $last^*_{G-x}(v)$ can reach v in $G - x$, we have that $last^*_{G-x}(v) \preceq last_{G-x}(v)$. As $x \notin V(P)$, there exists a path $last^*_{G-x}(v) \to last_{G-x}(v)$ in $G - x$ following P.

If $v \in V(P)$, then, by the definition, $last_{G-x}(v)$ can reach $v = last^*_{G-x}(v)$. Suppose $v \notin V(P)$ in $G-x$. Then take any simple path $Q = last_{G-x}(v) \to v$ in $G - x$. Let r be the last vertex on Q such that $r \in V(P)$. The subpath $r \to v$ of Q is a satellite path, so $r \preceq last^*_{G-x}(v)$ and thus there exists a $r \to last^*_{G-x}(v)$ path in $G - x$. Since r is reachable from $last_{G-x}(v)$ in $G - x$, we conclude that there exists a $last_{G-x}(v) \to last^*_{G-x}(v)$ path in $G - x$. Hence, $last_{G-x}(v)$ and $last^*_{G-x}(v)$ are strongly connected in $G - x$.

Clearly, all vertices that are strongly connected with $last^*_{G-x}(v)$ can reach v in $G-x$. Therefore, $last_{G-x}(v)$ is the latest vertex of P in the strongly connected component of $last^*_{G-x}(v)$ in $G - x$. ∎

Let us now note the following simple lemma on strong connectivity between vertices of a path.

LEMMA 6.2. *Let H be a digraph and let $Q = q_1, \ldots, q_\ell$ be a directed path in H. Then for any $i = 1, \ldots, \ell$, there exist two indices $a \in \{1, \ldots, \ell\}$ and $b \in \{i, \ldots, \ell\}$ such that the only vertices of P that are strongly connected to q_i are $q_a, \ldots, q_b$.*

Proof. Assume by contradiction that there exist q_j and q_k such that q_k lies between q_i and q_j on P, and q_j is strongly connected to q_i, whereas q_k is not strongly connected to q_i. Without loss of generality suppose $i < j$ (the case $i > j$ is symmetric). We have $i < k < j$. Since q_i and q_j are strongly connected, there exists a path $q_j \to q_i$ in G. However, since q_k lies before q_j on P, there exists a path $q_k \to q_j \to q_i$ in G. But q_i lies before q_k on P, so there exists a $q_i \to q_k$ path in G. We conclude that q_i and q_k are strongly connected, a contradiction. ∎

Georgiadis et al. [29] showed the following theorem.

THEOREM 6.1. ([29]) *Let G be a digraph. In linear time one can construct a data structure supporting $O(1)$-time queries of the form "are u and v strongly connected in $G - x$?", where $u, v, x \in V(G)$.*

Hence, after linear preprocessing, by Lemma 6.2 applied to $H = G$ and $Q = P$, we could compute the vertex $last_{G-x}(v)$ (which, by Lemma 6.1, is the latest vertex of P in the SCC of $last^*_{G-x}(v) \in V(P)$) by using binary search. Each step of binary search would take a single query to the data structure of [29], so computing $last_{G-x}(v)$ out of $last^*_{G-x}(v)$ would take $O(\log n)$ time.

However, we can do better using Theorem 4.3. In order to compute $last_{G-x}(v)$ out of $last^*_{G-x}(v)$ in constant time using Theorem 4.3, we assign a label $f(v)$ to each vertex v as follows: for each vertex $p_i \in V(P)$ we set $f(p_i) = i$. For each vertex $w \notin V(P)$, we set $f(w) = 0$. The maximum labeled vertex in the SCC of $last^*_{G-x}(v)$ in $G - x$ is precisely $last_{G-x}(v)$. Our final task is to show how to compute $last^*_{G-x}(v)$ efficiently.

6.1 Computing $last^*_{G-x}(v)$.

For $i = \ell, \ldots, 1$, define the *layer* L_i to be the vertices of $V \setminus V(P)$ reachable from p_i *by a satellite path*, minus $\bigcup_{j=i+1}^{\ell} L_j$. In other words, L_ℓ contains vertices not on the path P that are reachable from p_ℓ by a satellite path, and each subsequent layer L_i contains vertices reachable from p_i by a satellite path that are not reachable from $p_{i+1}, \ldots, p_\ell$ by a satellite path. The layers $L_\ell, \ldots, L_1$ can be computed by performing ℓ graph searches with starting points $p_\ell, \ldots, p_1$. The graph search never enters the vertices of P (except the starting vertex) or the vertices of previous layers. This way, the total time needed to perform all ℓ graph searches is linear.

For each layer L_i we also compute the dominator tree D_i of $G[L_i \cup \{p_i\}]$ rooted at p_i. Denote by $d_i(w)$ the parent of $w \in L_i$ in D_i. Since we have $E(G[L_i \cup \{p_i\}]) \cap E(G[L_j \cup \{p_j\}]) = \emptyset$ for $i \neq j$, the dominator trees for all $i = 1, \ldots, \ell$ can be computed in linear time overall.

Before proceeding with the computation, we need a few more definitions. Let $v \in V \setminus V(P)$. We denote by $layer(w)$ the index in $j \in [\ell]$ for which $w \in L_j$, if it exists. Recall that $last^*_G(v)$ is the latest vertex on P with a satellite path to v in G.

Suppose $last^*_G(v)$ exists, since otherwise $last^*_{G-x}(v)$ does not exist either. We distinguish two cases: (i) when $layer(x)$ does not exist or $layer(x) \neq layer(v)$, and the more involved case (ii) when $layer(x) = layer(v)$. We first show that case (i) is actually very easy to handle.

LEMMA 6.3. *If $layer(x)$ does not exist or $layer(x) \neq layer(v)$, then $last^*_{G-x}(v) = last^*_G(v)$.*

Proof. Let $l_v = layer(v)$. By the definition of a layer, $last^*_G(v) = p_{l_v}$. There is a satellite path from p_{l_v} to v in $G[L_{l_v} \cup \{p_{l_v}\}]$, avoiding x, as $x \notin L_{l_v}$. Consequently, $last^*_G(v) \preceq last^*_{G-x}(v)$.

However, since $G - x$ is a strict subgraph of G, we also have $last^*_{G-x}(v) \preceq last^*_G(v)$. We conclude that indeed $last^*_{G-x}(v) = last^*_G(v)$. ∎

The most interesting case is when $layer(x) = layer(v)$, which we deal with as follows. Set $l = layer(x) = layer(v)$. We will exploit the dominator tree D_l. If x does not dominate v in D_l, then again one can show that $last^*_{G-x}(v) = last^*_G(v)$. Otherwise, we will show that it is enough to compute the latest vertex $p_q \in V(P)$, for which there is a path from p_q to some vertex of $D_l[x, v]$ in $G - x$. In order to efficiently compute the appropriate vertices p_q, we in turn show that it is sufficient to execute a precomputation phase during which we store for each vertex $w \in V \setminus V(P)$ only the latest vertex $undom(w) \in V(P)$ such that there is a satellite path from $undom(w)$ to w avoiding the parent of w in $D_{layer(w)}$. Finally, we explain how to compute the values $undom(w)$ for all w in $O(m \log m/\log\log m)$ time. Below we study the case $layer(x) = layer(v)$ in more detail.

LEMMA 6.4. *If $layer(x) = layer(v)$ and x is not an ancestor of v in $D_{layer(v)}$, then $last^*_{G-x}(v) = last^*_G(v)$.*

Proof. We already argued that $last^*_{G-x}(v) \preceq last^*_G(v)$. As x is not an ancestor of v in D_l, the dominator tree of $G[L_l \cup \{p_l\}]$, there is a path avoiding x from $last^*_G(v)$ to v in $G[L_l \cup \{p_l\}]$. Hence, $last^*_{G-x}(v) = last^*_G(v)$. ∎

Recall that we denote by $undom(w)$ the latest vertex on P that has a satellite path to w in $G - d_{layer(w)}(w)$ (that is, avoiding the parent of w in the dominator tree $D_{layer(w)}$). In other words, $undom(w) = last^*_{G-d_{layer(w)}(w)}(w)$. If no such vertex exists, $undom(w) = \perp$.

LEMMA 6.5. *Let Q be a satellite path from $p_k = undom(w)$ to w in G avoiding $d_l(w)$, where $l = layer(w)$. Then $k < l$ and $Q = Q_1Q_2Q_3$, where $undom(w) \to y = Q_1 \subseteq G[L_k \cup \{p_k\}]$, $Q_2 = yz \in V(L_k) \times V(L_l)$ (i.e., Q_2 is a single-edge path), and $z \to w = Q_3 \subseteq G[D_l[d_l(w)] \setminus \{d_l(w)\}]$.*

Proof. First observe that $k < l$ follows from the fact that w is not reachable from $p_{l+1}, \ldots, p_\ell$ at all in the corresponding layers (by the definition of L_l), and it is not reachable from p_l in $G[L_l \cup p_l] - d_l(w)$ by the definition of a dominator tree.

Moreover, by the definition of layers, there is no edge in G from a vertex of L_i to a vertex of L_j, where $i > j$. In particular, there is no edge from L_l to L_k.

Since the edges in G can only go from lower-numbered layers to the higher-numbered ones, and Q is a satellite path, Q cannot visit a lower-numbered layer after visiting a higher-numbered layer. Suppose that Q goes through a vertex of $a \in L_j$, where $k < j < l$. Let $R = a \to w$ be a subpath of Q. By the definition of Q, R does not go through $d_l(w)$. Moreover, a is reachable from p_j in $G - d_l(w)$ (since $a \in L_j$ and $d_l(w) \in L_l$) by a satellite path. Consequently, w is reachable from p_j in $G - d_l(w)$ by a satellite path, which contradicts the fact that $p_k = last^*_{G-d_l(w)}(w)$. We conclude that indeed $V(Q) \subseteq L_k \cup \{p_k\} \cup L_l$, and the vertices of L_l appear on Q only after the vertices of L_k. Hence Q can be expressed as $Q_1(yz)Q_3$, where $Q_1 \subseteq G[L_k \cup \{p_k\}]$ and $Q_3 \subseteq G[L_l]$.

It remains to prove that in fact we have $Q_3 \subseteq G[D_l[d_l(w)] \setminus \{d_l(w)\}]$. Clearly, $d_l(w) \notin V(Q_3)$ since Q avoids $d_l(w)$. Suppose a vertex $t \in V(Q_3) \cap (L_l \setminus D_l[d_l(w)])$ exists. Then, by Lemma 5.1 the subpath $t \to w$ of Q_3 has to go through $d_l(w)$, a contradiction. ∎

For convenience we identify $\perp$ with p_0 and extend the order $\prec$ so that $\perp \prec p_i$ for all $i = 1, \ldots, \ell$.

LEMMA 6.6. *If $layer(x) = layer(v)$ and x is an ancestor of v in $D_{layer(v)}$, then $last^*_{G-x}(v) = p_q$, where $q = \max\{t : p_t = undom(w), w \in D_{layer(v)}(x, v]\}$.*

Proof. Let $l = layer(v) = layer(x)$. We first show that for all $w \in D_l(x, v]$, if $undom(w) \neq \perp$ then the vertex $undom(w)$ has a satellite path to v avoiding x in G. By this, $p_q \preceq last^*_{G-x}(v)$ follows. Consider a satellite path Q from $undom(w)$ to w, avoiding $d_l(w)$. By Lemma 6.5, $V(Q) \cap L_l \subseteq D_l[d_l(w)] \setminus \{d_l(w)\}$. Since $D_l[d_l(w)] \setminus \{d_l(w)\} \subseteq D_l[x] \setminus \{x\}$, Q avoids x.

Now suppose that $p_q \prec last^*_{G-x}(v)$. Then, $undom(w) \prec last^*_{G-x}(v)$ for all $w \in D_l(x, v]$. Take any simple satellite path Q from $last^*_{G-x}(v)$ to v in $G - x$, and consider the earliest vertex $w \in D_l(x, v]$ that it contains. Then the $last^*_{G-x}(v) \to w$ subpath of Q avoids all vertices on $D_l(x, w)$ (including $d_l(w)$). This contradicts that $undom(w)$ is the latest vertex on P that has a satellite path to w in $G - d_l(w)$. Hence, $last^*_{G-x}(v) \preceq p_q$ and we obtain $last^*_{G-x}(v) = p_q$. ∎

Suppose we have the vertices $undom(w)$ computed for all $w \in V \setminus V(P)$. For any $i = 1, \ldots, \ell$, let D'_i be the tree D_i with labels on its edges added, such that the label of an edge $d_i(w)w$, where $w \in L_i$, is equal to $undom(w)$. Then, by Lemma 6.6, computing $last^*_{G-x}(v)$ when $l = layer(x) = layer(v)$ and x is an ancestor of v in D_l can be reduced to finding a maximum label on the x' to v path of D'_l, where x' is the child of x in D_l that $v \in D_l[x']$. Such queries can be answered in $O(1)$ time after linear preprocessing of D'_l: we can use the data structure of [11] for finding x', and that of [20] for finding the maximum label on a path.

6.2 Computing $undom(w)$, for all w, in $O(n \log n/\log\log n)$ time. Our final task is to compute

the vertices $undom(w)$ for all $w \in V \setminus V(P)$. Let initially $ud(w) := \perp$ for all $w \in V \setminus V(P)$. The goal is to eventually obtain $ud(w) = undom(w)$ for each w. The computation will be divided into ℓ phases numbered ℓ down to 1. During the phase i, we process all edges $xy \in (V \times L_i) \cap E(G)$. Recall that by the definition of the levels, there is no edge $e \in V(L_j) \times V(L_i)$ for $j > i$.

Each phase is subdivided into rounds, where each round processes a single edge from $E(G) \cap ((V \setminus L_i) \times L_i)$ and some edges from $E(G) \cap (L_i \times L_i)$. The edges originating in "later" layers are processed before the edges originating in "earlier" layers. Consider the round processing an edge $xy \in L_j \times L_i$, where $j < i$. We start by initializing a queue of edges Q, initially containing only xy. While Q is not empty, we extract an edge zw from Q. We test whether $ud(w) \neq \perp$, and if so, we set $ud(w) := p_j$, we remove from G all edges from $E(G) \cap (D_i[w] \times (L_i \setminus D_i[w]))$ and push them to Q. The round ends when Q becomes empty. Afterwards, we proceed with the next round. A phase ends when there are no unprocessed edges incident to a vertex of L_i.

The following lemma establishing the correctness of the above procedure is proved in the full-version.

LEMMA 6.7. *Fix some phase i. After all rounds processing edges from*

$$E(G) \cap ((L_j \cup L_{j+1} \cup \dots L_{i-1}) \times L_i),$$

we have $ud(w) = undom(w)$ for all $w \in L_i$ such that $p_j \preceq undom(w)$ and $ud(w) = \perp$ for all $w \in L_i$ such that $undom(w) \prec p_j$.

COROLLARY 6.1. *After the algorithm finishes, we have $ud(w) = undom(w)$ for all $w \in V \setminus V(P)$.*

Proof. We apply Lemma 6.7 for all i and $j = 1$. ∎

Finally, we now analyze the time complexity of the procedure that computes the vertices $undom(w)$ for all $w \in V \setminus V(P)$.

LEMMA 6.8. *The algorithm from this section can be implemented to run in time $O\left(m \frac{\log m}{\log \log m}\right)$.*

Proof. The time for computing the dominator trees of all layers L_i is $O(m)$ in total. Moreover, we can compute the correct order with which to process the incoming edges of each layer L_i, in $O(n+m)$ in total using radix-sort. Observe that the insertions and extraction to/from the maintained queue Q, through all layers L_i, take $O(m)$ time overall since each edge is inserted into Q at most once.

Now we bound the total time spent on reporting and deleting the edges from G. This can be done with the help of a dynamic two-dimensional range reporting data structure, as we next explain. We build such a data structure for each graph $G[L_i]$ separately – recall that in the i-th phase we only report/remove the edges of $E(G) \cap (L_i \times L_i) = E(G[L_i])$.

Let $n' = |L_i|$ and $m' = |E(G[L_i])|$. Let $ord : L_i \cup \{p_i\} \to [1..n'+1]$ be some preorder of the dominator tree D_i. Let $size(v) = |V(D_i[v])|$. Clearly, both ord and $size$ can be computed in linear time. Note that we have $u \in D_i[v]$ if and only if $ord(u) \in [ord(v), ord(v) + size(v) - 1]$.

We map each edge $xy \in E(G[L_i])$ to a point $(ord(x), ord(y))$ on the plane. Let A be the set of obtained points. We store the points A in a two-dimensional dynamic range reporting data structure of Chan and Tsakalidis [15].[6] This data structure supports insertions and deletions in $O(\log^{2/3+o(1)} b)$ time, and allows reporting all points in a query rectangle in $O\left(\frac{\log b}{\log \log b} + k\right)$ time, where k is the number of points reported and b is the maximum number of points present in the point set at any time. Hence, we can build the range reporting data structure in $O(m' \log^{2/3+o(1)} m') = O\left(m' \frac{\log m'}{\log \log m'}\right)$ time, by inserting all points from A.

Suppose we are required to remove and report the edges of $E(G[L_i]) \cap (D_i[v] \times (L_i \setminus D_i[v]))$. Then, it is easy to see that this can be done by querying the range reporting data structure for the edges corresponding to points in the set $[ord(u), ord(u) + size(u) - 1] \times ([1, ord(u) - 1] \cup [ord(u) + size(u), n'])$, which, clearly, consists of two orthogonal rectangles in the plane. This way, we get all the remaining edges xy of $G[L_i]$ such that $x \in D[v]$ and $y \notin D[v]$. Subsequently, (the corresponding points of) all the found edges are removed from the range reporting data structure. Since each edge of $G[L_i]$ is reported and removed only once, and we issue $O(m')$ queries to the range reporting data structure, the total time used to process any sequence of deletions is $O\left(m' \frac{\log m'}{\log \log m'}\right)$. Since the sum of m' over all layers is m, the total time to report all edges and removing them at all levels L_i is $O\left(m \frac{\log m}{\log \log m}\right)$. ∎

7 Proof of Theorem 4.2

In this section we prove the following theorem.

[6] In our application, any *decremental* two-dimensional range reporting data structure would be enough to obtain a nearly linear time algorithm. In fact, what we need is a range *extraction* data structure that removes all the reported points that remain in the query rectangle so that no point is reported twice. To the best of our knowledge, such a problem is not well-studied and this is why we use a seemingly much more general data structure of [15].

THEOREM 4.2. *Let G be a plane digraph and let $P \subseteq G$ be a simple path whose endpoints lie on a single face of G. In linear time one can build a data structure that can compute in $O(\log n)$ time whether there exists a $u \to v$ path going through $V(P)$ in $G - x$, where $u, v \in V$ and $x \in V(P)$.*

Let the subsequent vertices of P be $p_1, \ldots, p_\ell$, i.e., $P = p_1 \ldots p_\ell$. For simplicity, let us set $\prec := \prec_P$. Recall that the failed vertex x satisfies $x \in V(P)$.

Define G_1 and G_2 to be the two subgraphs of G lying weakly on the two sides of P. More formally, the bounding cycle of the infinite face (which, by our assumption, contains p_1 and p_ℓ) of G can be expressed as C_1C_2, where C_1 is a non-oriented $p_1 \to p_\ell$ path, and C_2 is a non-oriented $p_\ell \to p_1$ path. Then we define G_1 to be the subgraph of G weakly inside the cycle C_1P^R (where P^R is the non-oriented path P reversed), and G_2 to be the subgraph of G weakly inside the cycle C_2P. We have $G_1 \cap G_2 = P$. Note that P is a part of the infinite face's bounding cycle of each G_i.

Our strategy for finding a $u \to v$ path Q in $G - x$, $x \in V(P)$ that goes through $V(P)$ will be to consider a few cases based on how Q crosses P. In order to minimize the number of cases we will define a number of auxiliary notions in Section 7.1. Next, in Section 7.2 we will show how these notions can be efficiently computed. Subsequently, in Section 7.3 we will show a few technical lemmas stating that we only need to look for paths Q of a very special structure. Finally, in Section 7.4 we describe the query procedure.

7.1 Auxiliary notions We first extend the notation $first, last, first^*, last^*$ from Section 6 to subpaths of P. For any $H \subseteq G$ denote by $first_H(v, a, b)$ ($last_H(v, a, b)$) the earliest (latest resp.) vertex of $P[a, b]$ that v can reach (that can reach v, resp.) in H. In fact, as we will see, we will only need to compute $first_H(v, a, b)$ or $last_H(v, a, b)$ for $v \in V(P[a, b])$.

Similarly, for $v \in V \setminus V(P)$ denote by $first^*_H(v, a, b)$ ($last^*_H(v, a, b)$) the earliest (latest resp.) vertex of $P[a, b]$ that v can reach (that can reach v, resp.) *by a satellite path* in H. For $v \in V(P)$ we set $first^*_H(v, a, b) = last^*_H(v, a, b) = v$ if $a \preceq v \preceq b$. Note that in some cases we leave the values $first^*_H(v, a, b)$ and $last^*_H(v, a, b)$ undefined.

For brevity we sometimes omit the endpoints of the subpath if we care about some earliest/latest vertices of the whole path $P = P[p_1, p_\ell]$, and write (as in Section 6) e.g., $first^*_H(v)$ instead of $first^*_H(v, p_1, p_\ell)$ or $last_H(v)$ instead of $last_H(v, p_1, p_\ell)$.

We will also need the notions of earliest/latest jumps and detours that are new to this section.

DEFINITION 7.1. *Let $w \in V(P)$. Let H be a digraph such that $V(P) \subseteq V(H)$. We call the earliest vertex of P that w can reach using a satellite path in H the* earliest jump *of w in H and denote this vertex by $\alpha^-_H(w)$. Similarly, we call the latest vertex of P that w can reach using a satellite path in H the* latest jump *of w in H and denote this vertex by $\alpha^+_H(w)$.*

DEFINITION 7.2. *Let $x = p_k$. We define a* detour *of x to be any directed satellite path $D = p_i \to p_j$, where $i < k < j$. Detour D is called* minimal *if there is no detour $D' = p_{i'} \to p_{j'}$ of x such that $i \le i' \le j' \le j$ and $j' - i' < j - i$. A pair (p_i, p_j) is called a* minimal detour pair *of x if there exists a minimal detour of x of the form $p_i \to p_j$.*

The fact that p_1 and p_ℓ lie on a single face of G implies the following important property of minimal detours.

LEMMA 7.1. *For any $x \in V(P)$, there exist at most two minimal detour pairs of x.*

Proof. Let $x = p_k$. Assume the contrary and suppose there are at least three minimal detour pairs. Let some three minimal detours D_1, D_2, D_3 correspond to these pairs. Some two of D_1, D_2, D_3, say D_1, D_2, are both contained in some of G_1 and G_2 – say G_1.

Suppose $D_1 = p_i \to p_j$ and $D_2 = p_{i'} \to p_{j'}$. Assume w.l.o.g. that $i \le i'$. Then since (p_i, p_j) and $(p_{i'}, p_{j'})$ are distinct minimal detour pairs of x, we have $i < i' < k < j < j'$.

Recall that all vertices of P lie on the infinite face of G_1. Since $i < i' < j < j'$, any $p_i \to p_j$ path in G_1 crosses (i.e., has a common internal vertex with) any $p_{i'} \to p_{j'}$ path in G_1. Let $z \in (V(D_1) \cap V(D_2)) \setminus V(P)$. So D_1 can be represented as $D'_1D''_1$, where $D''_1 = z \to p_j$. Similarly, $D_2 = D'_2D''_2$, where $D'_2 = p_{i'} \to z$. Observe that $D'_2D''_1 = p_{i'} \to p_j$ is a detour of x. But $j - i' < j - i$, so this contradicts the minimality of D_1. ∎

7.2 Computing $first, last, first^*, last^*$, jumps and detours

LEMMA 7.2. *After linear preprocessing of G, for any $v \in V$ and $a, b \in V(P)$, where $a \preceq b$, the vertices $first^*_G(v, a, b)$ and $last^*_G(v, a, b)$ (if they exist) can be computed in $O(\log n)$ time.*

Proof. The case $v \in V(P)$ is trivial by the definition of $first^*_G(v, a, b)$ and $last^*_G(v, a, b)$: these vertices are defined and equal to v only for $a \preceq v \preceq b$. Hence, below we assume $v \in V \setminus V(P)$.

First observe that any satellite path is entirely contained in either G_1 or G_2. Hence, $first^*_G(v, a, b)$ is the

earlier of $first^*_{G_1}(v,a,b)$ and $first^*_{G_2}(v,a,b)$. In the following we focus on computing $first^*_{G_1}(v,a,b)$. Satellite paths in G_2 can be handled identically. Computing latest vertices can be done symmetrically.

Note that since $v \notin V(P)$, a satellite path $Q = v \to V(P[a,b])$ in G_1 does not use the outgoing edges of the vertices of P. Let us thus remove the outgoing edges of all $w \in V(P)$ from G_1 and this way obtain G'_1. Note that all the remaining paths in G'_1 are satellite now and all the satellite paths from v in G_1 are preserved in G'_1.

For simplicity let us assume that ℓ is a power of 2. This is without loss of generality, since otherwise we could extend P by adding less than ℓ vertices "inside" the last edge of P so that the length of P becomes a power of two. Recall that all the outgoing edges of $V(P)$ are removed in G'_1 anyway so this extension does not influence reachability or the answers to the considered queries.

Let $\mathcal{I}$ be the set of *elementary* intervals defined as follows: for each $f \in \{0,\ldots,\log \ell\}$ the intervals $[(k-1)\cdot 2^f + 1, k\cdot 2^f]$, for all $k \in [\frac{\ell}{2^f}]$, is included in $\mathcal{I}$. Observe that $\mathcal{I}$ has $\ell - 1$ intervals and these intervals can be conveniently arranged into a full binary tree.

We further extend G'_1 by adding an auxiliary vertex $p_{[e,f]}$ for each $[e,f] \in \mathcal{I}$ such that $e < f$. We also identify each vertex $p_i \in V(P)$ with $p_{[i,i]}$. For each $[e,f] \in \mathcal{I}$ with $e < f$ we add to G'_1 two edges $p_{[e,m]}p_{[e,f]}$ and $p_{[m+1,f]}p_{[e,f]}$, where $m = \left\lfloor \frac{e+f}{2} \right\rfloor$.

Observe that for any $[e,f] \in \mathcal{I}$, and $v \in V \setminus V(P)$ a path from v to any of $p_e,\ldots,p_f$ exists in G'_1 if and only if a $v \to p_{[e,f]}$ path exists in G'_1.

Since the vertices $p_1,\ldots,p_\ell$ lie on a single face of G_1 in this order, after the extension the graph G'_1 remains planar (see Figure 1). We can thus build a reachability oracle of Holm et al. [33] for G'_1 in linear time so that we can answer reachability queries in G'_1 in constant time.

Note that we would be able to compute $first^*_{G_1}(v,a,b)$ in $O(\log(b-a+1))$ time via binary search if only we could query in $O(1)$ whether a path from either of $p_x,\ldots,p_y$ to v exists for an arbitrary interval $[x,y]$. However, in $O(1)$ time we can only handle such queries for $[x,y] \in \mathcal{I}$. It is well-known that one can decompose an arbitrary $[x,y]$ into $O(\log n)$ elementary intervals: this way, we could implement a single step of binary search in $O(\log n)$ time and thus computing $first^*_{G_1}(v,a,b)$ would take $O(\log^2 n)$ time.

In order to compute $first^*_{G_1}(v,a,b)$ faster, we use a slightly more complicated binary-search-like recursive procedure as follows. The recursive procedure `first` is passed a single parameter $[e,f] \in \mathcal{I}$ and returns the earliest vertex from $p_{\max(a,e)},\ldots,p_{\min(b,f)}$ reachable from v in G'_1, or **nil** if such a vertex does not exits.

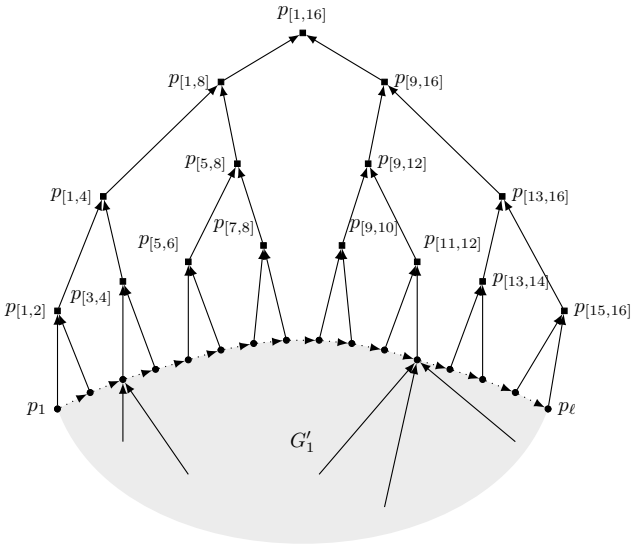

Figure 1: Auxiliary vertices corresponding to elementary intervals.

This way, in order to find $first^*_{G_1}(v,a,b)$ we compute `first`$([1,\ell])$.

The procedure `first`$([e,f])$ works as follows.

1. If $[a,b] \cap [e,f] = \emptyset$ then we return **nil**.
2. Otherwise, if $[e,f] \subseteq [a,b]$ and there is no $v \to p_{[e,f]}$ path in G'_1 (which can be decided in $O(1)$ time with a single reachability query on G'_1), we return **nil**.
3. Otherwise if $e = f$ we return p_e.
4. Finally, if none of the above cases apply, let $m = \left\lfloor \frac{e+f}{2} \right\rfloor$. If `first`$([e,m]) = v$, we return v. Otherwise (i.e., if `first`$([e,m])$ returned **nil**), we compute and return `first`$([m+1,f])$.

It is easy to see that the procedure is correct. Let us now analyze its running time. We say that a call to `first`$([e,f])$ is *non-leaf* if it invokes the procedure recursively. Note that the query algorithm runs in time linear in the number of non-leaf calls `first`$([e,f])$. We now show that the number of non-leaf calls is $O(\log n)$.

First observe that for each non-leaf call we have $[e,f] \cap [a,b] \neq \emptyset$. Let us first count non-leaf calls such that $[e,f] \not\subseteq [a,b]$. Suppose for some k there are at least 3 such (pairwise disjoint) intervals $[e_i,f_i]$, for $i = 1,2,3$ with $f_i - e_i + 1 = 2^k$. Let $e_1 < e_2 < e_3$. Since $[a,b]$ intersects both $[e_1,f_1]$ and $[e_2,f_2]$ but does not contain any of them $e_2 \in [a,b]$. Similarly, since $[a,b]$ intersects both $[e_2,f_2]$ and $[e_3,f_3]$ but does not contain any of them $f_2 \in [a,b]$. Hence, $[e_2,f_2] \subseteq [a,b]$,

a contradiction. So there are at most 2 such intervals per $k \in \{0, \ldots, \log_2 \ell\}$, and conclude that there are $O(\log n)$ non-leaf calls such that $[e, f] \not\subseteq [a, b]$.

Now suppose $\texttt{first}([e, f])$ is a non-leaf call and $[e, f] \subseteq [a, b]$. Since this is a non-leaf call, there is a path from v to some of p_j, where $j \in [e, f]$ in G'_1, and the call will return p_j. Consequently, the root call will also return p_j. We conclude that all such non-leaf calls return the same p_j. But there are only $O(\log n)$ intervals $[e, f]$ such that $p_j \in [e, f]$. ∎

LEMMA 7.3. *After linear preprocessing, we can compute $first_{G-x}(v, a, b)$ ($last_{G-x}(v, a, b)$) for all $v, a, b, x \in V(P)$ such that $a \preceq v \preceq b \prec x$ or $x \prec a \preceq v \preceq b$ in $O(\log n)$ time.*

Proof. We build the data structure of Theorem 6.1 for graph G in linear time. We only show how to compute $first_{G-x}(v, a, b)$, as $last_{G-x}(v, a, b)$ can be computed completely analogously.

Let $v' = first_{G-x}(v, a, b)$. Observe that $v' \preceq v$ since a path $v \to v$ exists in $G - x$. Note that $P[a, b]$ is a path in $G - x$. Since a path $v' \to v$ exists in $G - x$ (since $P[v', v]$ is a subpath of $P[a, b]$), v and v' are in fact strongly connected in $G - x$.

By Lemma 6.2, if v' is strongly connected to v in $G - x$, then all $y \in V(P)$, $v' \preceq y \preceq v$, are also strongly connected to v in $G - x$. Consequently, we can find v' by binary searching on the subpath $P[a, v]$ for the last vertex not strongly connected to y in $G - x$. A single step of binary search is executed in $O(1)$ time by Theorem 6.1. ∎

LEMMA 7.4. *For a digraph H such that $V(P) \subseteq V(H)$, the earliest and latest jumps of all $w \in V(P)$ can be computed in linear time.*

Proof. We only show how to compute earliest jumps, as latest jump can be computed analogously.

We first replace each $p_i \in V(P)$ in H with two vertices p_i^{in} and p_i^{out}. Next we change each original edge $p_i v$ of H into $p_i^{\text{out}} v$, and subsequently turn each edge up_i of H into up_i^{in}. Observe that every p_i^{out} has only outgoing edges, whereas every p_i^{in} only incoming.

Observe that there is a 1-1 correspondence between $p_i^{\text{out}} \to p_j^{\text{in}}$ paths in H and $p_i \to p_j$ *satellite* paths in H before the transformation since no path in the current H can go through either p_l^{in} or p_l^{out} as an intermediate vertex. Therefore, the task of computing the earliest jumps can be reduced to computing, for each $i \in [1, \ell]$, the minimum j such that p_j^{in} is reachable from p_i^{out} in the transformed H. This, in turn, can be done as follows, in an essentially the same way as we computed the layers in Section 6.1.

For each $i = 1, \ldots, \ell$, we run a reverse graph search from p_i^{in} in H that only enters vertices that have not been visited so far. If $w \in V(H)$ gets visited in phase i, we set $\alpha_H^-(w) = i$. Observe that if a vertex w is visited in phase i, this means that there exists a path $w \to p_i^{\text{in}}$ in H and for any $j < i$ there is no path $w \to p_j^{\text{in}}$ in H. This in particular applies to vertices w of the form p_l^{out}.

Finally, observe that the reverse graph search takes linear total time through all phases. ∎

LEMMA 7.5. *The minimal detour pairs of all $x \in V(P)$ can be computed in linear time.*

Proof. From the proof of Lemma 7.1 it follows that in fact there is at most one minimal detour pair for each $x \in V(P)$ in each G_i, $i = 1, 2$. We thus compute this pair (if it exists) separately for G_1 and G_2. Below we focus on G_1 since G_2 can be handled identically.

Let $x = p_k$. Suppose a minimal detour pair (p_i, p_j) of x in G_1 exists. Then the latest jump of p_i in G_1 also exists and $x \prec p_j \preceq \alpha^+_{G_1}(p_i)$. Let $a \in V(P)$ be the latest vertex $a \prec x$ such that $x \prec \alpha^+_{G_1}(a)$. We conclude that $p_i \preceq a$. Similarly, the earliest (with respect to the original order of P) jump of p_j in ${G_1}^{\text{R}}$ also exists and $\alpha^-_{{G_1}^{\text{R}}}(p_j) \preceq p_i \prec x$. Let $b \in V(P)$ be the earliest vertex satisfying $x \prec b$ such that $\alpha^-_{{G_1}^{\text{R}}}(b) \prec x$. Observe that we have $b \preceq p_j$, and thus $p_i \preceq a \prec x \prec b \preceq p_j$.

We now prove that in fact $p_i = a$ and $p_j = b$. First observe that since there exist detours $D_b = \alpha^-_{{G_1}^{\text{R}}}(b) \to b$ and $D_a = a \to \alpha^+_{G_1}(a)$ of x in G_1, by the definition of a we have $\alpha^-_{{G_1}^{\text{R}}}(b) \preceq a$ and $b \preceq \alpha^+_{G_1}(a)$. So, $\alpha^-_{{G_1}^{\text{R}}}(b) \preceq a \prec x \prec b \preceq \alpha^+_{G_1}(a)$. But $\alpha^-_{{G_1}^{\text{R}}}(b), a, b, \alpha^+_{G_1}(a)$ all lie on a single face of G_1 (in that order), so D_a and D_b have a common vertex. Similarly as in the proof of Lemma 7.1, we can conclude that in fact there exists a $a \to b$ detour of x in G_1. Hence, indeed $p_i = a$ and $p_j = b$.

It remains to show how to compute the vertices a, b, as defined above, for all $x \in V(P)$. Let us focus on computing the values a; the values b can be computed by proceeding symmetrically. We first compute the values $\alpha^+_{G_1}(w)$ for all $w \in V(P)$ in linear time using Lemma 7.4.

Now, the problem can be rephrased in a more abstract way as follows: given an array $t[1..\ell]$, compute for each $i = 1, \ldots, \ell$ the value $h[i] = \max\{j : j < i \wedge t[j] > i\}$. We solve this problem by processing t left-to-right and maintaining a certain subset S of indices $j < i$ that surely contains all the values $h[i], \ldots, h[\ell]$ that are less than i. The indices of S are stored in a stack sorted bottom-to-top. Additionally we maintain the invariant that for $S = \{i_1, \ldots, i_s\}$, where $i_1 < \ldots < i_s$, we have $t[i_1] > \ldots > t[i_s]$.

Initially, S is empty. Suppose we process some i. By the invariants posed on S we know that $h[i] \in S$. Let i_s be the top element of the stack. If $t[i_s] > i$ then $h[i] = i_s$ since $i_s = \max S$. Otherwise $t[i_s] \leq i$. Hence, we also have $t[i_s] \leq j$ for all $j \geq i$, so $h[j] \neq i_s$. Thus, we can remove i_s from S (by popping it from the top of the stack) without breaking the invariants posed on S. We repeat the process until S is empty or $t[i_s] > i$. If S becomes empty, we set $h[i] = -\infty$, otherwise $h[i] = i_s$. At this point, S contains $h[j]$ for all $j > i$ with $h[j] < i$.

In order to move to the next i, we need S to contain values $h[j]$ for all $j > i$ such that $h[j] \leq i$ and the stack storing S has to be sorted top to bottom according to the t-values. If $t[i_s] > t[i]$, we push i to S and finish. Suppose $t[i_s] \leq t[i]$. We show that $h[j] \neq t[i_s]$ for $j > i$ and thus we can safely remove i_s from S. Assume the contrary, i.e., $h[j] = i_s$ for some $j > i$. But then $t[i] \geq t[h[j]] > j$ and $h[j] = \max\{j' : j' < j \wedge t[j'] > i\} \geq i > h[j]$, a contradiction.

The running time of this algorithms is clearly linear in ℓ plus the number of stack operations. The total number of stack operations is linear in the number of indices pushed to the stack, i.e., no more than ℓ. ∎

7.3 Simplifying paths In this section we devise a few lemmas that will allow the query procedure to consider only paths of a very special form.

Let R be any path. Let $enter(R)$ be the first vertex of R that additionally lies on P. Similarly, let $leave(R)$ be the last vertex of R that additionally lies on P. Let $min(R)$ ($max(R)$) be the earliest (latest) vertex of $V(R) \cap V(P)$ wrt. to $\preceq_P$.

LEMMA 7.6. *Let a, b be some two vertices of P such that $x \notin V(P[a,b])$. Let $Q = LR$ be any $u \to v$ path in $G - x$ satisfying $enter(Q) \in V(P[a,b])$ and such that L is a satellite $u \to enter(Q)$ path. Let $u' = first_{G-x}(first^*_G(u,a,b), a, b)$.*

Then Q does not go through the vertices of $P[a,b]$ earlier than u'. Moreover, there exists a $u \to v$ path $Q' = L'R$ in $G - x$ such that $enter(Q') \in V(P[a,b])$, $u' \in V(L')$ and $leave(Q') = leave(Q)$.

Proof. First note that by $enter(Q) \in V(P[a,b])$, $first^*_G(u,a,b)$ is in fact defined. Hence $u' \in V(P[a,b])$ also exists and $u' \preceq first^*_G(v,a,b)$.

Suppose Q goes through a vertex u'' of $P[a,b]$ earlier than u'. Then there exists an $enter(Q) \to u''$ path in $G - x$. We have $first^*_G(u,a,b) \preceq enter(Q)$. Since $x \notin V(P[a,b])$, there exists a $first^*_G(u,a,b) \to enter(Q)$ path (e.g., $P[first^*_G(u,a,b), enter(Q)]$) in $G - x$. It follows that there exists a $first^*_G(u,a,b) \to u''$ path in $G - x$, which contradicts the definition of u'.

To finish the proof, note that it is sufficient to set L' to be a concatenation of a satellite $u \to first^*_G(u,a,b)$ path, any $first^*_G(u,a,b) \to u'$ path in $G - x$, and $P[u', enter(Q)]$. L' is not necessarily simple. ∎

LEMMA 7.7. *Let $Q = LR$ be any $u \to v$ path in $G - x$ satisfying $enter(Q) \prec x$ and such that L is a $u \to enter(Q)$ satellite path. Let $u' = first_{G-x}(first^*_G(u))$. Then $u' \preceq min(Q)$. Moreover, there exists a $u \to v$ path $Q' = L'R$ in $G - x$ such that $enter(Q') \prec x$, $min(L') = min(Q') = u'$, and $leave(Q') = leave(Q)$.*

Proof. It is enough to apply Lemma 7.6 to $a = p_1$ and b equal to the vertex preceding x on P. ∎

LEMMA 7.8. *Let $Q = LR$ be any $u \to v$ path in $G - x$ satisfying $x \prec leave(Q)$ and such that R is a $leave(Q) \to v$ satellite path. Let $v' = last_{G-x}(last^*_G(v))$. Then $max(Q) \preceq v'$. Moreover, there exists a $u \to v$ path $Q' = LR'$ in $G - x$ such that $x \prec leave(Q')$, $max(R') = max(Q') = v'$, $enter(Q') = enter(Q)$.*

Proof. This lemma is symmetric to Lemma 7.7 and therefore can be proved analogously. ∎

LEMMA 7.9. *Suppose a directed path $Q = u \to v$ in $G - x$ such that $min(Q) \prec x \prec max(Q)$ can be represented as Q_1RQ_2, where R is a $min(Q) \to max(Q)$ path. Then there is a $u \to v$ path Q' in $G - x$ that can be represented as $Q_1P_1P_2P_3Q_2$, where P_1 and P_3 are (possibly empty) subpaths of P and P_2 is a minimal detour of x.*

Proof. Let $R = R_1R_2$, where R_2 starts at y, $min(Q) \preceq y \prec x$ and R_2 does not go through any other vertices of $P[min(Q), x]$. Such an y exists since $min(Q) \prec x \prec max(Q)$. Observe that R_2 starts with some detour $D = y \to z$ of x, where $z \in P[x, max(Q)]$. Hence R can be rewritten as R_1DR_3, where $R_3 = z \to max(Q)$ is possibly an empty path. If D is not minimal, let $D' = y' \to z'$ be a minimal detour of x such that $y \preceq y' \prec x \prec z' \preceq z$. Note that since $min(Q) \preceq y \preceq y' \prec x$ and $x \prec z' \preceq z \preceq max(Q)$, $P[min(Q), y']D'P[z', max(Q)]$ is a $min(Q) \to max(Q)$ path in $G - x$. ∎

7.4 The query algorithm Finally we are ready to describe our query algorithm. Recall that we want to efficiently check whether there exists a $u \to v$ path Q in $G - x$ that goes through $V(P)$, for $x \in V(P)$. We consider cases based on the configuration of $enter(Q)$, $leave(Q)$ and x on P.

Case 1. Suppose there exists such a path Q that $enter(Q) \prec x$ and $x \prec leave(Q)$. Let $u' = first_{G-x}(first^*_G(u))$ as in Lemma 7.7. Since

$first^*_G(u) \prec x$, u' can be computed in $O(\log n)$ time by first using Lemma 7.2 to compute $first^*_G(u)$ and then using Lemma 7.3.

By Lemma 7.7, there exists a $u \to v$ path $Q' = L'R$ in $G-x$ such that $enter(Q') \prec x$, $min(L') = min(Q') = u'$ and $x \prec leave(Q')$. Let $v' = last_{G-x}(v, last^*_G(v))$. Similarly, since $x \prec leave(Q') \preceq last^*_G(v)$, v' can be computed in $O(\log n)$ time by Lemmas 7.2 and 7.3. By Lemma 7.8, there exists a $u \to v$ path $Q'' = L'R'$ in $G - x$ such that $enter(Q'') \prec x$, $x \prec leave(Q'')$, $max(Q'') = v'$. Since $min(L') = u'$ and Q'' cannot go through a vertex of P earlier then u' (by Lemma 7.7), we also have $min(Q'') = u'$.

As we have $u' = min(Q'') \prec x \prec max(Q'') = v'$, we can apply Lemma 7.9. In order to find a $u \to v$ path in $G - x$ such that $enter(Q) \prec x$ and $x \prec leave(Q)$, we only need to check whether there exits a path $P_1P_2P_3 = u' \to v'$ such that P_1 and P_3 are subpaths of P and P_2 is a minimal detour of x. By Lemma 7.1, there are only two minimal detour pairs (e, f) of x. Assuming $P_2 = e \to f$, P_1 exists if and only if $u' \preceq e$. Similarly P_3 exists if and only if $f \preceq v'$. Hence, each of at most two minimal detour pairs can be checked in constant time.

Case 2. Suppose that $enter(Q) \prec x$ and $leave(Q) \prec x$. Similarly as in the previous case, by Lemma 7.7 (and putting $Q := Q'$) we can assume that the sought path Q goes through $u' = first_{G-x}(first^*_G(u))$ and $min(Q) = u'$. As before, $u' \prec x$ can be reached from u in $G-x$. Now, since $u' = min(Q) \preceq leave(Q) \prec x$, we can limit our attention to such paths Q that the subpath from u' to $leave(Q)$ of Q is actually a subpath of P. Hence, in order to check if such Q exists, we only need to check whether there is a satellite path $z \to v$, where $u' \preceq z \prec x$. In other words we can check whether $last^*_G(v, u', x')$, where x' is a predecessor of x on P, exists. This can be done in $O(\log n)$ time using Lemma 7.2.

Case 3. Suppose that $x \prec enter(Q)$ and $x \prec leave(Q)$. This case is symmetric to the preceding case: we analogously check whether u can reach v' by a satellite path and a subpath of P.

Case 4. Finally, suppose $x \prec enter(Q)$ and $leave(Q) \prec x$. Let y be the vertex following x on P. Let us now apply Lemma 7.6 for $a = y$ and $b = p_\ell$, and let $u' = first_{G-x}(first^*_G(u, y, p_\ell), y, p_\ell)$ be as in Lemma 7.6. u' can be again computed in $O(\log n)$ time by first using Lemma 7.2 to obtain $first^*_G(u, y, p_\ell)$ and then applying Lemma 7.3. We conclude that there exists a $u \to v$ path Q' in $G - x$ that goes through u', does not go through vertices of P between x and u', and satisfies $leave(Q') \prec x$. Note that since Q' last departs from $V(P)$ at a vertex earlier than x, and simultaneously goes through $u' \in V(P[y, p_\ell])$, Q' can be written as $Q' = YRST$, where $R = u' \to s$, $S = s \to t$, $s \in V(P[y, p_\ell])$, $t \in V(P)$, $t \prec x$, and S is satellite.

Let t' be the earliest vertex of P such that t' can be reached from u' by a path of the form $R'S'$, where R' is a subpath of P and S' is a satellite path. Observe that we have $t' \preceq t$, since the path R does not go through any vertex between x and u' on P and thus t can be reached from u' by a path $P[u', s]S$. Note also that there exists a $u \to v$ path Q'' avoiding x such that $Q'' = YR'S'T'$: if we set $T' = P[t', t]T$, clearly all Y, R', S', T' avoid x.

Furthermore, note that we could replace path T' with any $t' \to v$ path T'' in $G-x$ satisfying $leave(T'') \prec x$ and would still obtain a $u \to v$ path in $G - x$. Note that $enter(T'') = t'$, so clearly $enter(T'') \prec x$. Checking if any such path $T'' = t' \to v$ exists can be performed as in case 2.

It remains to show how to compute the earliest possible vertex t' given u'. Observe that we want to pick such $s \in P[u', p_\ell]$ that can reach the earliest vertex of P using a satellite path. To this end we need to find the earliest out of the earliest backwards jumps of all vertices of $P[u', p_\ell]$. This can be computed in $O(1)$ time, after preprocessing the earliest backwards jumps for all suffixes of P in linear time.

Summary. The query procedure tries to find a $u \to v$ path Q in $G - x$ for each of the above four configurations of $enter(Q)$, $leave(Q)$ and x. Each of the cases is handled in $O(\log n)$ time.

8 2-Reachability Queries

To answer 2-reachability queries we reuse the recursive approach and extend the data structure of Section 4. Suppose we want to know whether there exist two vertex-disjoint paths from u to v in some graph G with suppressed set A and separator S_{ab} that arises in the recursive decomposition. Recall that $S_{ab} - A$ can be decomposed into $O(1)$ simple paths $P_1, \ldots, P_k$ in $G-A$.

First suppose that there is no P_i such that some $u \to v$ path goes through $V(P_i)$ in $G - A$. Then either no $u \to v$ path in $G - A$ exists at all, or it is contained in at most one child subgraph $G[V_i] - A$ ($i \in \{1, 2\}$), where V_1, V_2 are the subsets of $V(G)$ strictly on one side of the separator. In this case, in $O(1)$ time we reduce our problem to searching for two vertex-disjoint $u \to v$ paths in a graph $G'[V_i \cup \{r'\}]$ with suppressed set $(A \cap V_i) \cup \{r'\}$, as described in Section 3.

Otherwise, there exists a $u \to v$ path in $G - A$ that goes through $V(S_{ab})$. We handle this case without further recursive calls (so, the time needed to reduce the original problem to this case is clearly $O(\log n)$). Observe that this guarantees that, in fact, no $u \to v$ path goes through a vertex of A in G. This is because the suppressed set can only contain (possibly

contracted) vertices of the separators in the ancestors of G in the recursion tree, and no $u \to v$ path could go through these separators. Moreover, every $u \to v$ path in the input graph G_0 (i.e., G_0 denotes the ancestor of G in the recursion tree that is the root) is preserved in G. Hence, for any $x \in V(G)$ we can test whether there exists a $u \to v$ path in $G - x$ by issuing a (u, v, x) query to the 1-sensitivity oracle built on the input graph G_0.

In fact, we will focus on finding some $x \in V(G)$ that lies on all $u \to v$ paths in G (equivalently, in $G - A$). In other words, we will be looking for a *separating vertex* certifying that v is not 2-reachable from u. There are a few cases to consider.

First, assume that we can find two paths P_i, P_j, $i \neq j$ such that there exists both a $u \to v$ path through P_i and a $u \to v$ path through P_j in G. If some vertex $x \in V(G)$ lies on all $u \to v$ paths in G, then either $x \in V(P_i) \cap V(P_j)$, or $x \in V(G) \setminus V(P_i)$, or $x \in V(G) \setminus V(P_j)$. We can check all $x \in V(P_i) \cap V(P_j)$ in $O(\log n)$ time using only $O(1)$ queries to our 1-sensitivity reachability oracle of Theorem 1.1, since $|V(P_i) \cap V(P_j)| = O(1)$. In the full version we show that the two latter cases, i.e., finding a separating vertex outside a path through which one can reach v from u, can be handled separately in $O(\log n)$ time after additional linear preprocessing.

Finally, suppose there is a unique path P_i such that there exists a $u \to v$ path through $V(P_i)$ in G. As we mentioned above, we can find some vertex $x \in V(G) \setminus V(P_i)$ that lies on all $u \to v$ paths in G, if such a vertex exists. Suppose that $x \in V(P_i)$. Note that by the uniqueness of i, all $u \to v$ paths in G are preserved in $G[\bar{V}_i]$, where $\bar{V}_i = V(G) \setminus V(S_{ab}) \cup V(P_i)$. Similarly to Section 4, we can assume that u and v lie in a single connected component of $G[\bar{V}_i]$ and the endpoints of P_i lie on a single face of that component. We show that these assumptions enable us to find a separating vertex on a path through which one can reach v from u in $O(\log^{2+o(1)} n)$ time after additional $O(n \log^{5+o(1)} n)$ preprocessing and using $O(n \log^{2+o(1)} n)$ space (per node of the recursion tree). The total preprocessing time and space consumption can be analyzed analogously as in Section 3. The following theorem, whose proof is deferred to the full version of the paper, summarizes our data structure.

THEOREM 1.2. *In $O(n \log^{6+o(1)} n)$ time one can construct an $O(n \log^{3+o(1)} n)$-space data structure supporting the following queries in $O(\log^{2+o(1)} n)$ time. For $u, v \in V(G)$, either find some separating vertex $x \notin \{u, v\}$ lying on all $u \to v$ paths in G, or declare v 2-reachable from u.*

References

[1] A. Abboud, L. Georgiadis, G. F. Italiano, R. Krauthgamer, N. Parotsidis, O. Trabelsi, P. Uznanski, and D. Wolleb-Graf. Faster Algorithms for All-Pairs Bounded Min-Cuts. In *ICALP 2019*, pages 7:1–7:15, 2019.

[2] I. Abraham, S. Chechik, and C. Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *STOC 2012*, pages 1199–1218, 2012.

[3] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., 1972.

[4] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.

[5] S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.

[6] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

[7] M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, March 2001.

[8] S. Baswana, K. Choudhary, and L. Roditty. Fault-tolerant subgraph for single-source reachability: General and optimal. *SIAM Journal on Computing*, 47(1):80–95, 2018.

[9] S. Baswana, K. Choudhary, and L. Roditty. An efficient strongly connected components algorithm in the fault tolerant model. *Algorithmica*, 81(3):967–985, Mar 2019.

[10] S. Baswana, U. Lath, and A. S. Mehta. Single source distance oracle for planar digraphs avoiding a failed node or link. In *SODA 2012*, pages 223–232, 2012.

[11] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

[12] G. Borradaile, S. Pettie, and C. Wulff-Nilsen. Connectivity oracles for planar graphs. In *SWAT 2012*, pages 316–327, 2012.

[13] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

[14] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–96, 1998. Corrigendum in 27(3):383-7, 2005.

[15] T. M. Chan and K. Tsakalidis. Dynamic orthogonal

range searching on the RAM, revisited. In *SoCG 2017*, pages 28:1–28:13, 2017.

[16] P. Charalampopoulos, P. Gawrychowski, S. Mozes, and O. Weimann. Almost optimal distance oracles for planar graphs. In *STOC 2019*, pages 138–151, New York, NY, USA, 2019. ACM.

[17] P. Charalampopoulos, S. Mozes, and B. Tebeka. Exact distance oracles for planar graphs with failing vertices. In *SODA 2019*, pages 2110–2123, 2019.

[18] H. Cheung, L. Lau, and K. Leung. Graph connectivities, network coding, and expander graphs. *SIAM Journal on Computing*, 42(3):733–751, 2013.

[19] K. Choudhary. An Optimal Dual Fault Tolerant Reachability Oracle. In *ICALP 2016*, pages 130:1–130:13, 2016.

[20] E. D. Demaine, G. M. Landau, and O. Weimann. On Cartesian trees and range minimum queries. *Algorithmica*, 68(3):610–625, 2014.

[21] K. Diks and P. Sankowski. Dynamic plane transitive closure. In *ESA 2007*, pages 594–604, 2007.

[22] R. Duan and S. Pettie. Connectivity oracles for failure prone graphs. In *STOC 2010*, pages 465–474, New York, NY, USA, 2010. ACM.

[23] R. Duan and S. Pettie. Connectivity oracles for graphs subject to vertex failures. In *SODA 2017*, pages 490–509, 2017.

[24] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013.

[25] H. N. Gabow. The minset-poset approach to representations of graph connectivity. *ACM Trans. Algorithms*, 12(2), February 2016.

[26] L. Georgiadis, D. Graf, G. F. Italiano, N. Parotsidis, and P. Uznanski. All-Pairs 2-Reachability in $O(n^\omega \log n)$ Time. In *ICALP 2017*, pages 74:1–74:14, 2017.

[27] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. *ACM Trans. Algorithms*, 13(1):9:1–9:24, 2016. Announced at SODA 2015.

[28] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-vertex connectivity in directed graphs. *Information and Computation*, 261(2):248–264, 2018. Announced at ICALP 2015.

[29] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Strong connectivity in directed graphs under failures, with applications. In *SODA 2017*, pages 1880–1899, 2017.

[30] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *SODA 2004*, pages 862–871, 2004.

[31] L. Georgiadis and R. E. Tarjan. Dominator tree certification and divergent spanning trees. *ACM Trans. Algorithms*, 12(1):11:1–11:42, November 2015.

[32] M. Henzinger, A. Lincoln, S. Neumann, and V. Vassilevska Williams. Conditional Hardness for Sensitivity Problems. In *ITCS 2017*, volume 67, pages 26:1–26:31, 2017.

[33] J. Holm, E. Rotenberg, and M. Thorup. Planar reachability in linear space and constant time. In *FOCS 2015*, pages 370–389, 2015.

[34] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[35] P. W. Purdom Jr. and E. F. Moore. Immediate predominators in a directed graph [H] (algorithm 430). *Commun. ACM*, 15(8):777–778, 1972.

[36] M. Karpinski and Y. Nekrich. Space efficient multi-dimensional range reporting. In *COCOON 2009*, pages 215–224, 2009.

[37] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150–167, 2002.

[38] P. N. Klein and S. Mozes. Optimization algorithms for planar graphs, 2017.

[39] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

[40] E. S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, 1969.

[41] J. Łącki and Y. Nussbaum and P. Sankowski and C. Wulff-Nilsen. Single source - all sinks max flows in planar digraphs. In *FOCS 2012*, pages 599–608, 2012.

[42] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *KDD 2010*, pages 115–124, 2010.

[43] S. Mozes and E. E. Skop. Efficient vertex-label distance oracles for planar graphs. *Theory Comput. Syst.*, 62(2):419–440, 2018.

[44] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *PADL 2006*, pages 73–87, 2006.

[45] P. Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *FOCS 2004*, pages 509–517, 2004.

[46] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[47] S. Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *ESA 1993*, pages 372–383, 1993.

[48] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[49] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[50] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.

[51] J. van den Brand and T. Saranurak. Sensitive distance and reachability oracles for large batch updates. In *FOCS 2019*, 2019.

[52] B. T. Wilkinson. Amortized bounds for dynamic orthogonal range reporting. In *ESA 2014*, pages 842–856, 2014.