Exact Distance Oracles for Planar Graphs with Failing Vertices

PANAGIOTIS CHARALAMPOPOULOS, Reichman University, Israel
SHAY MOZES, Reichman University, Israel
BENJAMIN TEBEKA, Reichman University, Israel

We consider exact distance oracles for directed weighted planar graphs in the presence of failing vertices. Given a source vertex $u$, a target vertex $v$ and a set $X$ of $k$ failed vertices, such an oracle returns the length of a shortest $u$-to-$v$ path that avoids all vertices in $X$. We propose oracles that can handle any number $k$ of failures. We show several tradeoffs between space, query time, and preprocessing time. In particular, for a directed weighted planar graph with $n$ vertices and any constant $k$, we show an $\tilde{O}(n)$-size, $\tilde{O}(\sqrt{n})$-query-time oracle. We then present a space vs. query time tradeoff: for any $q \in [1, \sqrt{n}]$, we propose an oracle of size $n^{k+1+o(1)}/q^{2k}$ that answers queries in $\tilde{O}(q)$ time. For single vertex failures ($k = 1$), our $n^{2+o(1)}/q^2$-size, $\tilde{O}(q)$-query-time oracle improves over the previously best known tradeoff of Baswana et al. [SODA 2012] by polynomial factors for $q \geq n^t$, for any $t \in (0, 1/2]$. For multiple failures, no planarity exploiting results were previously known.

A preliminary version of this work was presented in SODA 2019. In this version, we show improved space vs. query time tradeoffs relying on the recently proposed almost optimal distance oracles for planar graphs [Charalampopoulos et al., STOC 2019; Long and Pettie, SODA 2021].

CCS Concepts: Theory of computation → Shortest paths.

Additional Key Words and Phrases: planar graphs, distance oracles, shortest paths, Voronoi diagrams

1 INTRODUCTION

Computing shortest paths is one of the most well-studied algorithmic problems. In the data structure version of the problem, the aim is to compactly store information about a graph such that the distance (or the shortest path) between any queried pair of vertices can be retrieved efficiently. Data structures supporting distance queries are called distance oracles. The two main measures of efficiency of a distance oracle are the space it occupies and the time it requires to answer a distance query. Another quantity of interest is the time required to construct the oracle.

In recent decades researchers have investigated the shortest path problem in graphs subject to failures, or more broadly, to changes. One such variant is the replacement paths problem. In this problem we are given a graph $G$ and vertices $u$ and $v$. The goal is to report the $u$-to-$v$ distance in $G$ for each possible failure of a single edge along the shortest $u$-to-$v$ path. Another variant is that of constructing a distance oracle that answers $u$-to-$v$ distance queries subject to edge or vertex failures ($u$, $v$ and the set of failures are given at query time). Perhaps the most general of these variants is designing a fully-dynamic distance oracle; a data structure that supports distance queries as well as updates to the graph such as changes to edge lengths, edge insertions or deletions and vertex insertions or deletions.

1The $\tilde{O}(\cdot)$ notation hides polylogarithmic factors.
One obvious but important application of handling failures is in geographical routing \[31\]. Further motivation for studying this problem originates from Vickrey pricing in networks \[28, 40\]; see \[15\] for a concise discussion on the relation between the problems. A long-studied generalization of the shortest path problem is the $k$-shortest paths problem, in which not one but several shortest paths must be produced between a pair of vertices. This problem reduces to running $k$ executions of a replacement paths algorithm, and has many applications itself \[20\].

In this paper we focus on these problems, and in particular on handling vertex failures in planar graphs. Observe that edge failures easily reduce to vertex failures. Indeed, by replacing each edge $(a, e)$ of $G$ with a new dummy vertex $b$ and appropriately weighted edges $(a, b)$ and $(b, c)$; the failure of edge $(a, c)$ in $G$ corresponds to the failure of vertex $b$ in the new graph. Note that this transformation does not depend on planarity. In sparse graphs, such as planar graphs, this transformation only increases the number of vertices by a constant factor. Also note that there is no such obvious reduction in the other direction that preserves planarity. In general graphs, one can replace each vertex $v$ by two vertices $v_{in}$ and $v_{out}$, assign to $v_{in}$ (resp. $v_{out}$) all the edges incoming to $v$ (resp. outgoing from $v$) and add a 0-length directed edge $e$ from $v_{in}$ to $v_{out}$. The failure of vertex $v$ in the original graph corresponds to the failure of edge $e$ in the new graph. However, this transformation does not preserve planarity.

### 1.1 Related Work

**General graphs.** Demetrescu et al. presented an $O(n^2 \log n)$-size oracle answering single failure distance queries in constant time \[15\]. Bernstein and Karger, improved the construction time in \[6\]. Interestingly, Duan and Pettie, building upon this work, showed an $O(n^2 \log^3 n)$-size oracle that can report distances subject to two failures, in time $O(\log n)$ \[17\]. Based on this oracle, they then easily obtain an $O(n^k)$-size oracle answering distance queries in $O(1)$ time for any $k \geq 2$. Oracles that require less space for more than 2 failures have been proposed, such as the ones presented in \[41, 42\], but at the expense of $\Omega(n)$ query time. Such oracles are unsatisfactory for planar graphs, where single source shortest paths can be computed in linear or nearly linear time. See also \[18\] for a very recent result on answering distance queries subject to multiple edge failures in undirected graphs.

**Planar graphs.** Exact (failure-free) distance oracles for planar graphs have been studied extensively over the past three decades \[3, 8, 10, 13, 14, 16, 22, 24, 27, 35, 38\]. A very recent series of papers \[10, 11, 14, 27, 35\] has established Voronoi diagrams as a useful tool for designing distance oracles in planar graphs. In particular, in \[35\], the authors showed an $O^*(n)$-size, $\log^2 + o(1)$-query-time oracle.

As for handling failures, the replacement paths problem (i.e. when both the source and destination are fixed in advance) can be solved in nearly linear time \[19, 34, 43\]. For the single source, single failure version of the distance oracle problem (i.e. when the source vertex is fixed at construction time, and the query specifies just the target and a single failed vertex), Baswana et al. \[5\] presented an oracle with size and construction time $O(n \log^4 n)$ that answers queries in $O(\log^3 n)$ time. They then showed an oracle of size $O(n^2 \log q)$ for the general single failure problem (i.e. when the source, destination, and failed vertex are all specified at query time), that answers queries in time $O(q)$ for any $q \in [1, n^{1/2}]$. They concluded the paper by asking whether it is possible to design a compact distance oracle for a planar digraph which can handle multiple vertex failures. We answer this question in the affirmative.

Fakcharoenphol and Rao, in their seminal paper \[22\], presented distance oracles that require $O(n^{2/3} \log^{2/3} n)$ and $O(n^{4/5} \log^{3/5} n)$ amortized time per update and query for non-negative and arbitrary edge-weight updates respectively.\footnote{Though this is not mentioned in \[22\], the query time can be made worst case rather than amortized by standard techniques.} The space required by these oracles is $O(n \log n)$. Klein presented a similar data structure in \[32\] for the case where edge-weight updates are non-negative, requiring time $O(n^{2/3} \log^{5/3} n)$. Klein’s result was extended in \[29\], where, assuming non-negativity of edge-weight updates, the authors showed how to handle
edge deletions and insertions (not violating the planarity of the embedding), and in [30], where the authors showed how to handle negative edge-weight updates, all within the same time complexity. In fact, these results can all be combined, and along with a recent slight improvement on the running time of FR-Dijkstra [26], they yield a dynamic distance oracle that can handle any of the aforementioned edge updates and queries within time $O(n^{2/3} \frac{\log^{5/3} n}{\log^{3/2} \log n})$. We further extend these results by showing that vertex deletions and insertions can also be handled within the same time complexity. The main challenge lies in handling vertices of high degree.

An exact fault-tolerant distance labeling scheme for planar graphs, accommodating for a single failure was recently presented [4]. For the case where one is willing to settle for approximate distances, Abraham et al. [2] gave a $(1 + \epsilon)$ labeling scheme for undirected planar graphs with polylogarithmic size labels, such that a $(1 + \epsilon)$-approximation of the distance between vertices $u$ and $v$ in the presence of $|F|$ vertex or edge failures can be recovered from the labels of $u$, $v$ and the labels of the failed vertices in $O(|F|^2)$ time. They then use this labeling scheme to devise a fully dynamic $(1 + \epsilon)$-distance oracle with size $O(n)$ and $O(\sqrt{n})$ query and update time.\(^3\)

On the lower bounds side, it is known that an exact dynamic oracle requiring amortized time $O(n^{4/3-\delta})$, for any constant $\delta > 0$, for both edge-weight updates and distance queries, would refute the APSP conjecture, i.e. that there is no truly subcubic combinatorial algorithm for solving the all-pairs shortest path problems in weighted (general) graphs [1].

1.2 Our Results and Techniques

In this work we focus on distance queries subject to vertex failures in planar graphs. Our results can be summarized as follows.

1. We show how to preprocess a directed weighted planar graph $G$ in $O(n)$ time into an oracle of size $O(n)$ that, given a source vertex $u$, a target vertex $v$, and a set $X$ of $k$ failed vertices, reports the length of a shortest $u$-to-$v$ path in $G \setminus X$ in $O(\sqrt{kn})$ time. See Lemma 3.3.
2. We extend the exact dynamic distance oracles mentioned in the previous section to also handle vertex insertions and deletions without changing their space and time bounds. See Theorem 4.1.
3. For $k$ allowed failures, and for any $r \in [1, n]$, we show how to construct an $n^{k+1+o(1)}/r^k$-size oracle that answers queries in time $O(k \sqrt{r})$. For $k = 1$, this improves over the previously best known tradeoff of Baswana et al. [5] by polynomial factors for $r \geq n^t$, for any $t \in (0, 1)$. To the best of our knowledge, this is the first tradeoff for $k > 1$. See Fig. 1 for an illustration and Corollary 5.6 and Theorem 6.1 for more tradeoffs.

Our nearly-linear space oracle that reports distances in the presence of $k$ failures in $O(\sqrt{kn})$ time is obtained by adapting a technique of Fakcharoenphol and Rao [22]. In a nutshell, a planar graph can be recursively decomposed using small cycle separators, such that, in each level of the decomposition, the boundary of each piece (i.e. the vertices of the piece that also belong to other pieces in this level) is a union of a constant number of cycles with relatively few vertices. Instead of working with the given planar graph, one computes distances over its dense distance graph (DDG); a non-planar graph on the boundary vertices of the pieces which captures the distances between boundary vertices within each of the underlying pieces. Fakcharoenphol and Rao developed an efficient implementation of Dijkstra’s algorithm on the DDG. This algorithm, nicknamed FR-Dijkstra, runs in time roughly proportional to the number of vertices of the DDG (i.e. boundary vertices), rather than in time proportional to the number of vertices in the planar graph. Roughly speaking, Fakcharoenphol and Rao show that to obtain distances from $u$ to $v$ with $k$ edge failures, it (roughly) suffices to consider just the boundary vertices of the pieces in the recursive decomposition that contain failed edges. Since pieces at the same level of the recursive decomposition are edge-disjoint, the total number of boundary vertices in all the required pieces is only $O(\sqrt{kn})$. This $O(n)$-size,

\[^3\]A fully dynamic distance oracle supports arbitrary edge and vertex insertions and deletions, and length updates.
Fig. 1. Left: Tradeoff of the Space ($S$) vs. the Query time ($Q$) for exact distance oracles for a single failed vertex (i.e. $k = 1$) on a doubly logarithmic scale, hiding subpolynomial factors. The previous tradeoff is indicated by a solid line, while the new tradeoff is indicated by a dashed line. Right: The same tradeoff for $k = 1, \ldots, 5$, shown with different colours. The points on the $x$-axis correspond to the result of [17], while the new tradeoffs are indicated by dashed lines.

$\tilde{O}(\sqrt{kn})$-query-time oracle, supporting distance queries subject to a batch of $k$ edge cost updates, leads to their dynamic distance oracle.

The difficulty in handling vertex failures is that a high degree vertex $x$ may be a boundary vertex of many (possibly $\Omega(n)$) pieces in the recursive decomposition. Then, if $x$ fails, one would have to consider too many pieces and too many boundary vertices. Standard techniques such as degree reduction by vertex splitting are inappropriate because when a vertex fails all its copies fail. To overcome this difficulty we define a variant of the dense distance graph which, instead of capturing shortest path distances between boundary vertices within a piece, only captures distances of paths that are internally disjoint from the boundary. We show that such distances can be computed efficiently, and that it then suffices to include in the FR-Dijkstra computation (roughly) only pieces that contain $x$, but not as a boundary vertex. This leads to our nearly-linear-size oracle reporting distances in the presence of $k$ failures in $\tilde{O}(\sqrt{kn})$ time (item 1 above). See Section 3. Plugging the same technique into the existing dynamic distance oracles extends them to support vertex deletions (item 2 above). See Section 4.

Our main result, the space vs. query time tradeoff (item 3 above), is obtained by a combination of this technique, employment of external DDGs, and the recent static exact distance oracle presented in [35]. See Section 5. In the case where one is willing to sacrifice space in order to make preprocessing more efficient, we show an alternative tradeoff in Section 6. Such an oracle could be preferable in the case that one has to reconstruct the data structure every once in a while due to unfixable failures or other updates in the graph. This tradeoff is achieved by a combination of FR-Dijkstra on our variant of the DDG with $r$-divisions, external DDGs, and efficient point location in Voronoi diagrams—a tool that is used internally by the exact oracles we use as a black box in the other tradeoff. Finally, in Section 7 we show how to efficiently construct our oracles; in particular, the efficient construction of external DDGs may be of independent interest.

2 PRELIMINARIES

In this section we review the main techniques required for describing our result. Throughout the paper we consider a weighted directed planar graph $G = (V(G), E(G))$, embedded in the plane. (We use the terms weight and length for edges and paths interchangeably throughout the paper.) We use $|G|$ to denote the number of vertices in $G$. Since planar graphs are sparse, $|E(G)| = O(|G|)$ as well. For an edge $(u, v)$, we say that $u$ is its tail and $v$ is its head. $d_G(u, v)$ denotes the distance from $u$ to $v$ in $G$. We denote by $d_G(u, v, X)$ the distance from $u$ to $v$ in $G \setminus X$, where $X \in V(G)$ or $X \subset V(G)$; if the reference graph is clear from the context we may omit
the subscript. We assume that the input graph has no negative length cycles. If it does, we can detect this in \( O(n \log^2 n) \) time by computing single source shortest paths from any vertex [39]. In the same time complexity, we can transform the graph in a standard way so that all edge weights are non-negative and shortest paths are preserved. We further assume that shortest paths are unique as required for a result from [25] that we use; this can be ensured in \( O(n) \) time by a deterministic perturbation of the edge weights [21]. Each original distance can be recovered from the corresponding distance in the transformed graph in constant time.

**Separators and recursive decompositions in planar graphs.** Miller [36] showed how to compute a Jordan curve that intersects the graph at \( O(\sqrt{n}) \) vertices and separates it into two pieces with at most \( 2n/3 \) vertices each. Jordan curve separators can be used to recursively separate a planar graph until pieces have constant size. The authors of [33] show how to obtain a complete recursive decomposition tree \( T \) of \( G \) in \( O(n) \) time. \( T \) is a binary tree whose nodes correspond to subgraphs of \( G \) (pieces), with the root being all of \( G \) and the leaves being pieces of constant size. For each vertex \( u \) of \( G \), we fix an arbitrary leaf-piece in \( T \) that contains \( u \), and denote this piece by \( P_u \). We identify each piece \( P \) with the node representing it in \( T \). We can thus abuse notation and write \( P \in T \).

An \( r \)-division [23] of a planar graph, for \( r \in [1, n] \), is a decomposition of the graph into \( O(n/r) \) pieces, each of size \( O(r) \), such that each piece has \( O(\sqrt{r}) \) boundary vertices, i.e., vertices incident to edges in other pieces. Another usually desired property of an \( r \)-division is that the boundary vertices lie on a constant number of faces of the piece (holes). For every \( r \) larger than some constant, an \( r \)-division with this property (i.e., few holes per piece) is represented in the decomposition tree \( T \) of [33]. Throughout the paper, to avoid confusion, we use "nodes" when referring to \( T \) and "vertices" when referring to \( G \). We denote the boundary vertices of a piece \( P \) by \( \partial P \). We refer to non-boundary vertices as internal.

**Lemma 2.1 ([27]).** Each node in \( T \) corresponds to a piece such that (i) each piece has \( O(1) \) holes, (ii) the number of vertices in a piece at depth \( \ell \) in \( T \) is \( O(n/c^\ell_1) \), for some constant \( c_1 > 1 \), (iii) the number of boundary vertices in a piece at depth \( \ell \) in \( T \) is \( O(\sqrt{n}/c^\ell_2) \), for some constant \( c_2 > 1 \).

We use the following well-known bounds (see e.g., [27]).

**Proposition 2.2.** \( \sum_{P \in T} |P| = O(n \log n) \), \( \sum_{P \in T} |\partial P| = O(n) \) and \( \sum_{P \in T} |\partial P|^2 = O(n \log n) \).

We show the following bound that will be used in future proofs.

**Proposition 2.3.** \( \sum_{P \in T} |P| |\partial P| = O(n^2) \).

**Proof.** Let \( P_1^\ell, P_2^\ell, \ldots, P_j^\ell \) be the pieces at the \( \ell \)-th level of the decomposition. \( \sum_j |P_j^\ell| = O(n) \) since the pieces are edge-disjoint. We know by Lemma 2.1 that \( |\partial P_j^\ell| = O(\sqrt{n}/c^\ell_1) \) for all \( j \) and hence \( |\partial P_j^\ell|^2 = O(n/c^\ell_2) \) for all \( j \).

It follows that \( \sum_j |P_j^\ell|^2 = O(n^2/c^\ell_2) \) and the claimed bound follows by summing over all levels of \( T \). \( \square \)

**Dense distance graphs and FR-Dijkstra.** The dense distance graph of a piece \( P \), denoted \( DDG_P \), is a complete directed graph on the boundary vertices of \( P \). Each edge \((u, v)\) has weight \( d_P(u, v) \), equal to the length of the shortest \( u \)-to-\( v \) path in \( P \). \( DDG_P \) can be computed in time \( O((|\partial P|^2 + |P|) \log |P|) \) using the multiple source shortest paths (MSSP) algorithm [9, 32]. Over all pieces of the recursive decomposition this takes time \( O(n \log^2 n) \) in total and requires space \( O(n \log n) \) by Proposition 2.2. We next give a—convenient for our purposes—interface for FR-Dijkstra [22], which is an efficient implementation of Dijkstra’s algorithm on any union of \( DDGs \). The algorithm exploits the fact that, due to planarity, certain submatrices of the adjacency matrix of \( DDG_P \) satisfy the Monge property. (A matrix \( M \) satisfies the Monge property if, for all \( i < i' \) and \( j < j' \), \( M_{i,j} + M_{i',j} \leq M_{i,j'} + M_{i',j'} \) [37].) The interface is specified in the following theorem, which was essentially proved in [22], with some additional components and details from [30, 39].
Theorem 2.4 ([22, 30, 39]). A set of DDGs with $O(M)$ vertices in total (with multiplicities), each having at most $m$ vertices, can be preprocessed in time and extra space $O(M \log m)$ in total, so that, after this preprocessing, Dijkstra’s algorithm can be run on the union of any subset of these DDGs with $O(N)$ vertices in total (with multiplicities) in time $O(N \log N \log m)$, by relaxing edges in batches. Each such batch consists of edges that have the same tail.

The algorithm in the above theorem is called FR-Dijkstra. It is useful in computing distances in sublinear time, as demonstrated by Lemma 2.6 and Corollary 2.7 which are a reformulation of ideas from [22] and are provided below for completeness.

Definition 2.5. The cone of a vertex $u$ of $G$ is the union of the following DDGs: (i) $\text{DDG}_{P_u}$, with $u$ considered a boundary vertex of $P_u$. (ii) For every (not necessarily strict) ancestor $P$ of $P_u$, $\text{DDG}_Q$ of the sibling $Q$ of $P$.

Lemma 2.6. Let $x$ and $y$ be two vertices in the cone of a vertex $u$. The $x$-to-$y$ distance in $G$ equals the $x$-to-$y$ distance in this cone of $u$.

Proof. Let $P_u = P_0, P_1, \ldots, P_d = G$ be the ancestors of $P_u$ ordered by decreasing depth in $T$. Let $Q_i$ be the sibling of $P_i$ in $T$. Let cone$i$, be $\text{DDG}_{P_{i+1}} \cup \bigcup_{j < i} \text{DDG}_{Q_j}$. We will prove by induction that for any two vertices $x, y \in \text{cone}_i$, the $x$-to-$y$ distance in $P_i$ equals the $x$-to-$y$ distance in $\text{cone}_i$. This statement is trivially true for $i = 0$. Let us assume it is true for $k$. Consider an $x$-to-$y$ shortest path $p$ in $P_{k+1}$, where $x, y \in \text{cone}_k$. Path $p$ can be decomposed into maximal subpaths that are entirely contained in $P_k$ or $Q_k$ and whose endpoints are in $\{x, y\} \cup (\partial P_k \cap \partial Q_k)$. For each such subpath we either have a path with the same length in $\text{cone}_k$ by the inductive assumption, or an edge of $\text{DDG}_Q$. This shows that the length of $p$ is at least the length of the $x$-to-$y$ distance in $\text{cone}_k$. Since every edge of $\text{cone}_k$ corresponds to some path in $P_k$, the opposite also holds, so the two quantities are equal.

Corollary 2.7. Let $u, v$ be two distinct vertices in $G$. Let $p$ be a shortest $u$-to-$v$ path in $G$. If $p$ is not fully contained in $P_u$, then we can compute the length of $p$ by running FR-Dijkstra on the union of the cone of $u$ and the cone of $v$. This takes $O(\sqrt{n})$ time.

Proof. Since $p$ is not fully contained in $P_u$, $p$ must visit a vertex $w$ in the separator of the LCA of $P_u$ and $P_v$ in $T$. We are done by decomposing $p$ into the prefix ending at $w$ and the suffix beginning at $w$, and applying Lemma 2.6. The running time follows by Theorem 2.4 and Lemma 2.1.

3 Near Linear Space Data Structure for Any Number of Failures

In this section we show how to adapt the approach of [22] for distance oracles supporting cumulative edge changes to support distance queries with failed vertices. The main technical challenge lies in dealing with failures of high-degree vertices, since such vertices may belong to many pieces at each level of the decomposition. For example, think of a failure of the central vertex in a wheel graph, which belongs to all the pieces in the recursive decomposition. Note that standard degree reduction techniques such as vertex splitting are not useful because when a vertex fails all its copies fail. This is in contrast with the situation when dealing only with edge-weight updates, since each edge can be in at most one piece per level. We circumvent this by defining and employing the strictly internal dense distance graph. The main intuition is that strictly internal DDGs enable us to handle pieces that only contain failed boundary vertices, i.e. do not contain any internal vertex that fails. Then, only pieces that contain internal failed vertices are "problematic". Note however, that a vertex is internal in at most one piece per level of the decomposition.

Definition 3.1. The strictly internal dense distance graph of a piece $P$, denoted $\text{DDG}_i^P$, is a complete directed graph on the boundary vertices of $P$. An edge $(u, v)$ has weight $d_i^P(u, v)$ equal to the length of the shortest $u$-to-$v$ path in $P$ that is internally disjoint from $\partial P$.
The sole difference to the standard definition is that in our case paths are not allowed to go through ∂P. Observe that the shortest path in P between two vertices of ∂P is still represented in DDG_p, just not necessarily by a single edge as in DDG. This establishes the following lemma.

**Lemma 3.2.** For any piece P and any two boundary vertices u, v ∈ ∂P, the u-to-v distance in DDG^o_P equals the u-to-v distance in DDG_p.

We now discuss how to efficiently compute DDG^o_P. We construct a planar graph ˆP, by creating a copy of P and incrementing the weight of each edge w, such that u ∈ ∂P, by C = 2 ∑_{e ∈ E(G)} |w(e)|. DDG_p can be computed in O((|∂P|^2 + |P|) log |P|) time using MSSP [9, 32]. Observe that any u-to-v path in ˆP that starts at ∂ ˆP and is internally disjoint from ∂P has exactly one edge uw with u ∈ ∂P, so its length is at least C and less than 2C, while any u-to-v path that has an internal vertex in ∂P is of length at least 2C. Therefore, the u-to-v distance in ˆP is equal to C plus the length of the shortest u-to-v path in P that is internally disjoint from ∂P if the latter one is not ∞. We thus set ˆd_p(u, v) = d_p(u, v) − C. This completes the description of the computation of DDG^o_p. Note that since C is defined in terms of G rather than P, edge weights greater than C in DDG_p effectively represent infinite length in the sense that such edges will never be used by any shortest path (in P nor in G). Also note that it follows directly from the definition of the Monge property that subtracting C from each entry of a Monge matrix preserves the Monge property. Therefore, we can use ∪_P DDG^o_P in FR-Dijkstra (Theorem 2.4) instead of ∪_P DDG_p.

**Preprocessing.** We compute a complete recursive decomposition tree T of G in time O(n) as discussed in Section 2. We compute DDG^o_P for each non-leaf piece P ∈ T and preprocess it as in FR-Dijkstra. By Proposition 2.2, Theorem 2.4 and the above discussion, the time and space complexities are O(n log^2 n) and O(n log n), respectively.

**Query.** Upon query (u, v, X), we run FR-Dijkstra on the union of the following DDG^o_s, which we denote by D(u, v, X) or just D when the arguments are clear from the context (inspect Fig. 2 for an illustration):

1. For each w ∈ {u, v}, DDG^o_{P_w \setminus X} with w regarded as a boundary vertex. This can be computed on the fly in constant time since the size of the leaf-piece P_w is constant.
2. For each w ∈ {u, v}, for each ancestor P of P_w (including P_w), DDG^o_Q of the sibling Q of P if Q does not contain any internal (i.e. non-boundary) vertex that is in X.
3. For each x ∈ X, DDG^o_{P_x \setminus X}. This can be computed on the fly in constant time since the size of the leaf-piece P_x is constant.
4. For each x ∈ X, for each ancestor P of P_x (including P_x), DDG^o_Q of the sibling Q of P if Q does not contain any internal vertex that is in X.

We can identify these DDG^o_s in O(k log n) time by traversing the parent pointers from each P_i, for i ∈ X, and marking all the nodes that have an internal failed vertex. We make one small but crucial change to FR-Dijkstra. When running FR-Dijkstra, we do not relax edges whose tail is a failed vertex. This guarantees that, although failed vertices might appear in the graph on which FR-Dijkstra is invoked, the u-to-v shortest path computed by FR-Dijkstra does not contain any failed vertices. We therefore obtain the following lemma.

**Lemma 3.3.** There exists a data structure of size O(n log n), which can be constructed in O(n log^2 n) time, and answers the following queries in O(√kn log^2 n) time. Given vertices u and v, and a set X of k failed vertices, report the length of a shortest u-to-v path that avoids the vertices of X.

**Proof.** We have already discussed the space occupied by the oracle and the time required to build it. It remains to analyze the query algorithm.

**Correctness.** First, it is easy to see that no edge (y, z) of any of the DDG^o_s in D represents a path containing a vertex x ∈ X, unless {y, z} ∩ X ≠ ∅. The latter case does not affect the correctness of the algorithm, since in
Fig. 2. A portion of the complete recursive decomposition tree $T$ of a graph $G$. The light gray and red pieces are the ones that would be considered by the failure-free distance oracle upon query $d(u, v)$. However, given the failure of vertex $x$, the $DDG^x$ of the red piece is invalid. The dark gray pieces are the ones that our algorithm considers instead of the red piece. The $DDG^x$s of the dark gray pieces, that are descendants of the red piece, allow us to represent its $DDG^x$ subject to the failure of $x$.

For FR-Dijkstra we do not relax edges whose tail is a failed vertex. Hence, the algorithm never computes a distance corresponding to a path going through a failed vertex.

It remains to show that the shortest path in $G \setminus X$ is represented in $D$. For this, by Corollary 2.7, it suffices to prove that for each piece $A$ in the cone of $u$ (and similarly in the cone of $v$), either $DDG^x_A$ for $A \setminus X$ belongs to $D$, or $D$ contains enough information to reconstruct $DDG^x_A$ for $A \setminus X$ (i.e. subject to the failures) during FR-Dijkstra.

In the latter case we say that $DDG^x_A$ is represented in $D$. Note that, for any piece $P$, $DDG^x_P$ is represented in $D$ if the $DDG^x$s of its two children in $T$ are represented in $D$. (This follows by an argument identical to the one used in the proof of Lemma 2.6.) If $A$ contains no internal failed vertex then $DDG^x_A$ is in $D$ by point 1 or 2 above. We next consider the case that $A$ does contain some failed vertex $x \in X$, the $DDG^x$ of any non-root ancestor of $P_y$ in $T$ is represented in $D$.

We proceed by the minimal counterexample method. For any $y \in X$, $DDG^x_y$ is in $D$ since it is computed on the fly in point 3. Let $F$ be the deepest node in $T$ that is a strict ancestor of $P_y$ for some $y \in X$ and whose $DDG^x$ is not represented in $D$. It follows that one of $F$’s children must also be an ancestor of $P_y$ and by the choice of $F$ its $DDG^x$ is represented in $D$. Let the other child of $F$ be $J$. If $J$ is an ancestor of some $P_z$, $z \in X$, then $DDG^x_J$ is also represented in $D$ by the choice of $F$. Otherwise, $J$ does not contain any internal failed vertex, and hence
$DDG_f^r$ is in $D$ by point 4. In either case, the $DDG^r$’s of both children of $F$ are represented in $D$, so $DDG_f^r$ is also represented in $D$, a contradiction.

Time complexity. Let $r = n/k$ and consider an $r$-division of $G$ in $T$. The pieces of this $r$-division have $O(\frac{n}{\sqrt{r}}) = O(\sqrt{kn})$ boundary vertices in total and this is known to also be an upper bound on the total number of boundary vertices (with multiplicities) of ancestors of pieces in this $r$-division (cf. the discussion after Corollary 5.1 in [27]).

Recall that we have chosen a leaf-piece $P_i$ for each vertex $i \in \{u, v\} \cup X$. Each piece (other than the $P_i$’s) whose $DDG^r$ belongs to $D$ is a sibling of an ancestor of some $P_i$. This implies that each $i \in \{u, v\} \cup X$ contributes the $DDG^r$’s of at most two pieces per level of the decomposition. Let the ancestor of $P_i$ that is in the $r$-division be $R_i$. For each $P_i$, we only need to bound the total size of pieces it contributes that are descendants of $R_i$, since we have already bounded the total size of the rest. We do so by applying Lemma 2.1 for the subtree of $T$ rooted at each $R_i$. (The extra $O(\sqrt{r})$ boundary vertices we start with do not alter the analysis of this lemma as these many are anyway introduced by the first separation of $R_i$.) It yields $2 \sum \frac{\sqrt{r}}{c_2}$, where $c_2 > 1$, which is $O(\sqrt{r})$. Summing over all $k + 2$ pieces $P_i$ we obtain the upper bound $O(k\sqrt{r}) = O(\sqrt{kn})$.

FR-Dijkstra runs in time proportional to the total number of vertices of the $DDG^r$’s in $D$ up to a $\log^2 n$ multiplicative factor and hence the time complexity follows.

Remark. By using existing techniques (cf. [30, Section 5.4]), we can report the actual shortest path $\rho$ in time $O(|\rho| \log \log \Delta_\rho)$, where $\Delta_\rho$ is the maximum degree of a vertex of $\rho$ in $G$.

4 DYNAMIC DISTANCE ORACLES CAN HANDLE VERTEX DELETIONS

In this section we briefly explain how the techniques of Section 3, and specifically our notion of strict dense distance graphs ($DDG^r$’s) can be used to facilitate vertex deletions in dynamic distance oracles for planar graphs.

The dynamic distance oracle of [22] for non-negative edge-weight updates was improved and simplified in [32]. In [32], the algorithm obtains an $r$-division of $G$, and then computes and preprocesses the $DDG$’s of the pieces of the $r$-division in $O(n \log n)$ time to allow for FR-Dijkstra computations in the union of these $DDG$’s in time $O(\frac{n}{\sqrt{r}} \log^2 n)$. For a given query asking for the distance from some vertex $u$ to some vertex $v$, the algorithm performs standard Dijkstra computations within the piece containing $u$ (resp. $v$) to compute the distances from $u$ to the boundary vertices of the piece (resp. from the boundary vertices of the piece to $v$). The algorithm then combines this with an FR-Dijkstra computation on the boundary vertices of the $r$-division. Given an edge update, only the $DDG$ of the unique piece in the $r$-division containing the updated edge needs to get updated, and this requires $O(r \log r)$ time. The balance is $r = n^{2/5} \log^{2/3} n$, yielding $O(n^{2/5} \log^{5/3} n)$ time per update and query. This result was extended in [29], where the authors showed how to allow for edge insertions (not violating the planarity of the embedding) and edge deletions and further in [30] where the authors showed how to handle arbitrary (i.e. also negative) edge-weight updates. The time complexity was improved by a $\log^{4/3} \log n$ factor in [26].

We observe that, by using $DDG^r$’s instead of standard $DDG$’s, vertex deletions can also be handled as follows.

Each vertex is either a boundary vertex in each piece of the $r$-division containing it, or an internal vertex in a unique piece. If a deleted vertex is a boundary vertex, we just mark it as such and do not relax edges outgoing from it during (FR-)Dijkstra computations. If a deleted vertex is internal, we recompute the $DDG^r$ of the piece containing it, and reprocess it in time $O(r \log r)$ exactly as in the case of edge-weight updates. The only slightly technical issue we need to take into account is that in Section 3, edge weights in $DDG^r$ are shifted by $C$ (recall that $C$ is defined as twice the sum of edge weights in the entire graph $G$). The problem is that $C$ might change after

\footnote{This remark also applies to the dynamic distance oracle presented in Section 4 and to the oracles presented in Section 5. However, it does not apply to the oracles presented in Section 6, where we use some $DDG$’s without storing MSSP data structures or exact distance oracles for the underlying graphs, which would allow us to retrieve the path underlying each $DDG$ edge efficiently.}
each update operation, and this update affects the weights of all the edges in all DDG's. This can be easily solved using indirection. Instead of using the explicit value of $C$ in each edge weight, we represent $C$ symbolically, and store the actual value of $C$ explicitly at some placeholder. Updating $C$ can be done in constant time because only the explicit value at the placeholder needs to be updated. Whenever an edge weight is required by the algorithm, it is computed on the fly in constant time using the value of $C$ stored in the placeholder. The data structures underlying FR-Dijkstra do not make use of any integer data structures like predecessor data structures—all used data structures are comparison based. Hence, since the value of $C$ is greater than all edge-weights at the time they are built, they are identical to the data structures that would have been built for this piece with any subsequent value of $C$. Vertex additions do not alter shortest paths, and hence can be treated trivially. Note that, as in [29], we can afford to recompute the entire data structure from scratch after every $O(\sqrt{r})$ operations. This guarantees that the number of vertices and number of boundary vertices in each piece remain $O(r)$ and $O(\sqrt{r})$, respectively, throughout. We formalize the above discussion in the following theorem.

**Theorem 4.1.** A planar graph $G$ can be preprocessed in time $O(n \log^{\frac{2}{3}} n \log \log n)$ so that edge-weight updates, edge insertions not violating the planarity of the embedding, edge deletions, vertex insertions and deletions, and distance queries can be performed in time $O(n^{2/3} \log^{2/3} n \log \log n)$ each, using $O(n)$ space.

## 5 TRADEOFF I: SPACE VS. QUERY TIME

In this section we describe a tradeoff between the size of the oracle and the query time.

### 5.1 More Preliminaries and Notation

We will be using exact distance oracles as a black-box, and as there are different tradeoffs, we denote the space, the query time and the preprocessing time for such a distance oracle over a planar graph of size $n$ as $\sigma(n)$, $\mu(n)$ and $\tau(n)$, respectively. Let us formally state results from [35].

**Theorem 5.1 ([35]).** Given a planar graph $G$ of size $n$, there exists a distance oracle that can be built in $n^{3/2+o(1)}$ time and admits either of:

(a) $n^{1+o(1)}$ space and $\log^{2+o(1)} n$ query time, or
(b) $n \log^{2+o(1)} n$ space and $n^{o(1)}$ query time.

We now define another useful modification of dense distance graphs.

**Definition 5.2.** The strictly external dense distance graph $DDG_{ext}^o(P_1, \ldots, P_i)$ of $G$ for pieces $P_1, \ldots, P_i$ is a complete directed graph on the boundary vertices of $P_1, \ldots, P_i$. The edge $(u, v)$ has weight equal to the length of the shortest $u$-to-$v$ path in $G \setminus \left( \bigcup_{j=1}^{i} P_j \right) \setminus \{u, v\}$.

$DDG_{ext}^o$s can be preprocessed using Theorem 2.4 together with $DDG^o$s so that we can perform efficient Dijkstra computations in any union of $DDG_{ext}^o$s and $DDG^o$s.

The number of pieces in an $r$-division is at most $cn/r$ for some constant $c$. For convenience, we define

$$g(n, r, k) = \binom{cn/r}{k} \leq \frac{(cn)^k}{r^kk!} \leq \frac{n^k}{r^kk^3},$$

where the last inequality holds for $k$ larger than some constant depending on $c$. We use $g(n, r, k)$ throughout to encapsulate the dependency on $k$. 

ACM Trans. Algor.
5.2 The Oracle

Warm up. Let us first sketch a warm-up $\tilde{O}(n^{3/2})$-size oracle with $\tilde{O}(\sqrt{k})$ query time that can handle single failures, using the approach of Section 3. Suppose that we store $DDG^e_{ext}$ for all triples of pieces of the $r$-division and that we have preprocessed these for efficient FR-Dijkstra computations together with the $DDG^e$s. The total space required is $O(g(n, r, 3)(\sqrt{k})^2) = O(n^{3/2})$. Upon query, we first retrieve pieces $R_u, R_v,$ and $R_x$ containing $u, v,$ and $x$, respectively—assume for now that these pieces are distinct. Then, we run FR-Dijkstra on $DDG^e_{ext}(R_u, R_v, R_x)$, the cone of $u$ in $R_u$, the cone of $v$ in $R_v$, and the pieces that allow us to represent the $DDG$ of $R_x$ subject to the failure of $x$. The query time is $\tilde{O}(\sqrt{k})$. This approach can be generalized to give an oracle that can handle $k$ failures, by considering $(k + 2)$-tuples of pieces of the $r$-division.\(^5\) The space required is $O(n \log n + g(n, r, k + 2)((k \sqrt{k})^2 + k \sqrt{k} \log n)) = O(n^{3/2} k^{3/2})$ and the query time is $\tilde{O}(k \sqrt{k})$.

Strategy. Instead of storing information for $(k + 2)$-tuples of pieces as in the warm-up, we will store the analogous information for $(k + 1)$-tuples and more information for $k$-tuples. Given $u, v, X$, where $X = \{x_1, \ldots, x_k\}$, we show how to compute $d_G(u, v, X)$ relying on the information stored for the tuples $(R_u, R_{x_1}, \ldots, R_{x_k})$ and $(R_{x_1}, \ldots, R_{x_k})$. Let us define $Y = \left(\bigcup_{r \in [u] \cup X} \partial R_r\right) \setminus X$. Our aim is to decompose the sought path on the last vertex of $Y$ it visits.

Auxiliary data structure. We define $u$-to-$k$-Boundary$(u, X, R, v)$ queries as follows. The input is

- a vertex $u$,
- a set of vertices $X = \{x_1, \ldots, x_k\}$ of cardinality $k$,
- a set $R$ consisting of at most $k + 1$ pieces of a specified $r$-division, such that each $j \in \{u\} \cup X$ is in some $R \in R$,
- a vertex $v \in R$ for some $R \in R$, which may be null.

The output of the query is the distance from $u$ to each of the vertices of $\{v\} \cup \left(\bigcup_{R \in R} \partial R\right) \setminus X$ in $G \setminus X$.\(^6\)

Example 5.3. In order to develop some intuition for the proof of the following lemma, consider Fig. 2 from Section 3, and suppose that the red piece (say $R_1$) and the unique dark gray piece that contains both $u$ and $v$ (say $R_2$) are pieces of the $r$-division. Further, consider a $u$-to-$k$-Boundary$(u, \{x\}, \{R_1, R_2\}, v)$ query. We will answer this query by running FR-Dijkstra on the union of the $DDG$'s of all gray pieces that are weak descendants of either $R_1$ or $R_2$ (this set of pieces corresponds to $D_G$ in the proof of the following lemma) and $DDG^e_{ext}(R_1, R_2)$, not relaxing edges whose tail is in $X$ if encountered.

Lemma 5.4. There exists a data structure of size $O(n^{k+1} \log n)$, which can be constructed in time $O(n^{k+1} \log^2 n)$, and answers $u$-to-$k$-Boundary$(u, X, R, v)$ queries in $O(k \sqrt{k} \log^2 n)$ time.

Proof. We first perform the precomputations of Section 3. We also obtain an $r$-division of $G$ from $T$ in $O(n)$ time. Let us denote the pieces of this $r$-division by $R_1, \ldots, R_q$.

We compute $DDG^o_{ext}(R_{i_1}, \ldots, R_{i_{k+1}})$ for each $(k + 1)$-tuple $(R_{i_1}, \ldots, R_{i_{k+1}})$ of pieces in the $r$-division. The $DDG^o_{ext}$ for all $(k + 1)$-tuples can be computed in time $O((cn)^{k+1} \log^2 n)$ for some constant $c$, as shown in Lemma 7.1 in Section 7; this dominates the preprocessing time. We preprocess these $DDG^o_{ext}$ together with the standard $DDG$'s, using Theorem 2.4, to allow for efficient FR-Dijkstra computations. The total space required is $O(g(n, r, k + 1)((k \sqrt{k})^2 + k \sqrt{k} \log n)) = O(n^{k+1} \log n)$.

\(^5\)We consider the elements of tuples to be unordered throughout.

\(^6\)If $v$ is null, we can just set it to be any vertex in $\left(\bigcup_{R \in R} \partial R\right) \setminus X$. In what follows, we thus do not treat this case separately.

ACM Trans. Algor.
Let us now consider a \( u \)-to-\( k \)-boundary \((u, X, \mathcal{R}, v) \) query. Let \( D = D(u, v, X) \) be the set of DDGs specified in the query procedure of Section 3 (and illustrated in Fig. 2). Now, let the restriction of \( D \) to \( \mathcal{R} \) be defined as
\[
\mathcal{D}_\mathcal{R} = \{ \text{DDG}_p \in D : P \text{ is a weak descendant of some } R \in \mathcal{R} \}.
\]
We run FR-Dijkstra on the union of \( \mathcal{D}_\mathcal{R} \) and the \( \text{DDG}_\text{ext} \) of a \((k + 1)\)-tuple that contains all elements of \( \mathcal{R} \), not relaxing edges whose tail is in \( X \) if encountered. This takes time \( O(k \sqrt{r} \log^2 n) \).

**Extra preprocessing: exact distance oracles.** For each \( k \)-tuple of pieces \( \mathcal{R} = (R_{i_1}, \ldots, R_{i_k}) \) of the \( r \)-division, we compute and store an exact distance oracle for graph \( G_R \), which is obtained from \( G \setminus (\cup_{j \in \{i_1, \ldots, i_k\}}(R_j \setminus \partial R_j)) \) by increasing the weight of each edge whose tail is in \( \partial R_j \) for some \( j \in \{i_1, \ldots, i_k\} \) by \( W = 2 \sum_{e \in E(G)} |w(e)| \). This way, any edge whose tail is in \( \partial R_j \) for some \( j \in \{i_1, \ldots, i_k\} \) is not internal in any shortest path in \( G_R \).

We are now ready to describe the query procedure; Fig. 3 provides an illustration of the setting.

![Fig. 3. An illustration of the query setting in the case where \( v \) does not lie in any piece from \( \mathcal{R} \). Suppose that the shortest \( u \)-to-\( v \) path is as shown. Part I of the query computes the length of its black solid portion using FR-Dijkstra, while part II computes the length of its red dashed portion by querying an exact distance oracle for \( G_{R'} \).](image)

**Query part I: \( u \)-to-\( Y \).** With the above lemma at hand, we can easily compute the \( u \)-to-\( Y \) distances. We first retrieve \( k + 1 \) not necessarily distinct pieces, \( R_u, R_{x_1}, \ldots, R_{x_k} \), such that \( u \in R_u \) and \( x_i \in R_{x_i} \). To support that, each vertex stores a pointer to some piece of the \( r \)-division that contains it. In the degenerate case that \( v \) is in one of those pieces, then we are done by performing a \( u \)-to-\( k \)-boundary \((u, X, \mathcal{R}, v) \) query, with \( \mathcal{R} = \{R_u, R_{x_1}, \ldots, R_{x_k}\} \).

(In order to be able to check whether a vertex is in some particular piece of \( T \) efficiently, we store, for each piece in \( T \), a binary tree with the vertices in the piece.) If on the other hand \( v \) does not lie in any piece from \( \mathcal{R} \), then any shortest \( u \)-to-\( v \) path must go through some vertex \( y \in Y \). A \( u \)-to-\( k \)-boundary \((u, X, \mathcal{R}, \text{null}) \) query returns the distance from \( u \) to each \( y \in Y \) in \( G \setminus X \). This takes time \( O(k \sqrt{r} \log^2 n) \).

**Query part II: \( Y \)-to-\( v \).** For each \( y \in Y \) we perform a \( y \)-to-\( v \) distance query in the exact distance oracle for \( G_{R'} \), where \( \mathcal{R}' = \{R_{x_1}, \ldots, R_{x_k}\} \). We then simply take the minimum of \( d_G(u, y, X) + d_{G_{R'}}(y, v) \) among all \( y \in Y \) and substract \( W \) from this value to retrieve the sought distance. The time required is \( O(k \sqrt{r}(\log^2 n + \mu(n))) \).

**Theorem 5.5.** Assume that given a planar graph \( G \) with \( n \) vertices, one can construct in \( r(n) \) time a \( \sigma(n) \)-size distance oracle that answers queries in \( \mu(n) \) time. Then, for any integer \( r \in [1, n] \) and for any integer \( k \leq \frac{r}{2} \), there...
exists a data structure of size $\tilde{O}(\frac{n^k}{r^k} \sigma(n))$, which can be constructed in time $\tilde{O}(\frac{n^k}{r^k} \tau(n))$, and can answer the following queries in $O(k \sqrt{r}(\log^2 n + \mu(n)))$ time. Given vertices $u$ and $v$ and a set $X$ of at most $k$ failed vertices, report the length of a shortest $u$-to-$v$ path in $G \setminus X$.

Remark. Our distance oracle can actually handle any number $f$ of failures that lie in at most $k$ pieces of the $r$-division in time $\tilde{O}((k + \sqrt{fk})\sqrt{r})$. This follows from the fact that the DDG’s we will add for a piece with $f_i$ failures have total size $\tilde{O}(\sqrt{fr})$ by the same analysis as in the proof of Lemma 3.3 and the fact that, given $f_1, \ldots, f_k$ such that $\sum_{i=1}^k f_i = f$, we have $\sum_{i=1}^k \sqrt{f_i} \leq \sqrt{fk}$ by the Cauchy-Schwarz inequality. This remark applies to Theorem 6.1 as well.

**Proof of Theorem 5.5.** The time complexity of the query algorithm is analyzed above. We next analyze its correctness, the space required by our data structure and its construction time.

**Query correctness.** Let $\rho$ be a shortest $u$-to-$v$ path in $G \setminus X$. Let $z$ be the last vertex of $\rho$ that belongs to $Y$, if any such vertex exists. The distance $d_G(u, z, X)$ from $u$ to $z$ in $G \setminus X$ is computed by FR-Dijkstra, while the distance from $z$ to $v$ in $G \setminus X$ is obtained from the query to the exact distance oracle. In the complementary case, in which no vertex of $\rho$ is in $Y$, we have that $v \in R_u$, and hence the sought distance is computed by the $u$-to-$k$-Boundary$(u, X, R, v)$ query. It is easy to see that we do not obtain any distance that does not correspond to an actual path in $G \setminus X$.

**Space complexity.** The space required for the exact distance oracles is $\tilde{O}(\frac{n^k}{r^k} \sigma(n))$. This dominates the $O(\frac{n^{k+1}}{r^k} \log n)$ space required by the data structure of Lemma 5.4 in the $\tilde{O}(\cdot)$ notation.

**Preprocessing time.** This is also dominated by the $O(\frac{n^k}{r^k} \tau(n))$ time required to build the exact distance oracles, at least in the $\tilde{O}(\cdot)$ notation. $\square$

By combining Theorems 5.1 and 5.5 we obtain the following tradeoffs.

**Corollary 5.6.** Given a planar graph $G$ of size $n$, there exists a distance oracle that supports up to $k$ vertex failures that can be built in time $\frac{n^k}{r^k} \cdot n^{3/2+o(1)}$ and admits either of:

(a) $\frac{n^k}{r^k} \cdot n^{1+o(1)}$ space and $k \sqrt{r} \cdot \log^2 n$ query time, or
(b) $\tilde{O}\left(\frac{n^k}{r^k} \cdot n\right)$ space and $k \sqrt{r} \cdot n^{o(1)}$ query time.

6 TRADEOFF II: FASTER PREPROCESSING, MORE SPACE

We now proceed to describe the tradeoff that was the main result in a preliminary version of this work [12], and is encapsulated in the following theorem.$^2$

**Theorem 6.1.** For any integer $r \in [1, n]$ and for any integer $k \leq \frac{n}{r}$, there exists a data structure of size $O\left(\frac{n^{k+1}}{r^k} \sqrt{nr} + n \log^2 n\right)$, which can be constructed in time $\tilde{O}\left(\frac{n^{k+1}}{r^k} \sqrt{nr} + n^2\right)$, and can answer the following queries in $O(k \sqrt{r} \log^2 n)$ time. Given vertices $u$ and $v$ and a set $X$ of at most $k$ failed vertices, report the length of a shortest $u$-to-$v$ path that avoids $X$.

For a fixed $r$, such that the oracles underlying both Corollary 5.6 and Theorem 6.1 have query time roughly $k \sqrt{r}$, the oracles of Corollary 5.6 require space smaller by roughly a factor of $n^{1/2}/r^{1/2}$ compared to the oracle that we present in this section. Interestingly, however, the preprocessing time of the oracles of Corollary 5.6 are worse by polynomial factors for some range of values of $r$. E.g., for $r = n^{1/4}$ and $k \geq 1$, compare $\frac{n^k}{r^k} \cdot n^{3/2} = n^{3k/4+12/8}$ with $\frac{n^k}{r^k} \cdot n^{3/2} / r^{1/2} + n^2 \leq 2 \cdot n^{3k/4+11/8}$.

$^2$This result was obtained before the recent breakthroughs in exact distance oracles, which have now allowed us to get the tradeoffs of Corollary 5.6.

ACM Trans. Algor.
6.1 Voronoi Diagrams with Point Location

**Definition 6.2.** Let $P$ be a directed planar graph with real edge-lengths and no negative-length cycles. Let $S$ be a set of vertices that lie on a single face of $P$, and call the elements of $S$ sites. Let $\omega : S \rightarrow \mathbb{R}^+$ be a weight function, i.e. each site has an additive weight $\omega(s) \geq 0$ associated with it. The additively weighted distance from a site $s \in S$ to a vertex $u \in V$, denoted by $d^\omega_P(u, v)$, is defined as $\omega(s)$ plus the length of the $u$-to-$v$ shortest path in $P$.

The additively weighted Voronoi diagram $(S, \omega) \ (V_D(P, S, \omega))$ is a partition of $V(P)$ into pairwise disjoint sets, each set $\text{Vor}(s)$ for each site $s \in S$. The set $\text{Vor}(s)$ is called the Voronoi cell of $s$, containing all vertices in $V(P)$ that are closer (w.r.t. $d^\omega_P(\cdot, \cdot)$) to $s$ than to any other site in $S$ (assuming that the distances are unique). (There is a dual representation $V_D^\ast(P, S, \omega)$ of a Voronoi diagram $V_D(P, S, \omega)$ as a planar graph with $O(|S|)$ vertices and edges.)

**Theorem 6.3 ([25, 27]).** Given a graph $P$, a set of sites $S$ that lie on a single face of $P$, and, for each integer $i \in [1, m]$, a weight function $\omega_i : S \rightarrow \mathbb{R}^+$, we can construct a data structure of size $O(|P| \log |P| + m|S|)$ that supports the following (point location) queries in $O(|P| \log |P|)$ time. Given $i \in [1, m]$ and a vertex $v$ of $P$ report the site $s \in S$ such that $v \in \text{Vor}(s)$ in the additively weighted Voronoi diagram $V_D(P, S, \omega_i)$ and the distance $d^\omega_i(u, v)$. The time and space required to construct this data structure are both $O(|P||S|^2 + m|S|)$.

**Remark.** Part of Theorem 6.3 is proved in [27], though not stated there explicitly as a theorem. It is a tradeoff to Theorem 1.1 of [27], requiring less space, and hence more applicable to our problem. The efficient construction algorithm is from [25].

6.2 Handling a Single Failure

For ease of presentation we first describe an oracle that can handle just a single failure. We prove the following lemma, which is a restricted version of Theorem 6.1.

**Lemma 6.4.** For any $r \in [1, m]$, there exists a data structure of size $O\left(\frac{5r}{\sqrt{27}r} + n \log^2 n\right)$, which can be constructed in time $O\left(\frac{5r}{\sqrt{27}r} + n^2\right)$, and can answer the following queries in $O(\sqrt{r} \log^2 n)$ time. Given vertices $u, v, x$, report the length of a shortest $u$-to-$v$ path that avoids $x$.

**Strategy.** As in the previous section, we first retrieve pieces $R_u$ and $R_x$ of the $r$-division and compute $u$-to-$Y$ distances in $G \setminus \{x\}$, where $Y = (\partial R_u \cup \partial R_x) \setminus \{x\}$, using FR-Dijkstra. Fig. 4(b) provides an illustration of the setting. Our goal then is to compute $Y$-to-$u$ distances with the aim of composing the partial results and taking the minimum over all $y \in Y$. To this end, we first identify an appropriate piece $Q$ in $T$ that contains $v$, and does not contain $u$ nor $x$. Suppose for simplicity that all vertices of $\partial Q$ lie on a single hole. Exploiting the fact that distances within $Q$ remain unchanged when $x$ fails, we employ Voronoi diagrams with point location (Theorem 6.3) for the piece $Q$, adapting ideas from [27]. Namely, for each $y \in Y$, we store $V_D(Q, \partial Q, \omega_y)$ (with point location functionality), where $\omega_y$ is a weight function that captures the $y$-to-$\partial Q$ distances in $G \setminus ((R_u \cup R_x) \setminus \{y\})$. We can then compute the required $y$-to-$v$ distance by performing a single point location query, which returns

$$\min_{s \in \partial Q} d^\omega_Q(s, v) = \min_{s \in \partial Q} \left( d_Q(y, s, (R_u \cup R_x) \setminus \{y\}) + d_Q(s, v) \right) = d(y, v, (R_u \cup R_x) \setminus \{y\}).$$

**Additional preprocessing.** For each (ordered) pair of pieces $(R_i, R_j)$ of the $r$-division we compute and store the following. Let $\Sigma$ be the set of separators $\sigma$ in the recursive decomposition that separate a piece into two subpieces $Q_\sigma$ and $R_\sigma$, such that $R_i \subseteq R_\sigma$ and $R_j \subseteq Q_\sigma$; note that this set may be empty. For each $\sigma \in \Sigma$, for each $y \in \partial R_i \cup \partial R_j$, for each hole $h$ of $Q_\sigma$, we compute and store a Voronoi diagram (with point location functionality) for $Q_\sigma$ with sites the boundary vertices of $Q_\sigma$ that lie on $h$, and additive weights the distances from $y$ to these sites in $G \setminus ((R_i \cup R_j) \setminus \{y\})$. 

ACM Trans. Algor.
Query. We first retrieve a piece $R_v$ of the $r$-division that contains $v$. We then proceed as follows (inspect Fig. 4 for an illustration).

1. First, if $\{u, x\} \cap V(R_v) \neq \emptyset$, we can retrieve at most two $r$-division pieces $R_1$ and $R_2$ whose union contains all of $u$, $v$, and $x$, and hence we can answer the distance query in $O(\sqrt{r} \log^2 n)$ time by performing query $u$-to-1-Boundary$(u, \{x\}, (R_1, R_2), v)$. We thus henceforth assume that $\{u, x\} \cap V(R_v) = \emptyset$.

2. Following parent pointers from $R_v$ in $\mathcal{T}$, we find the highest weak ancestor $Q$ of $R_v$ containing neither $u$ nor $x$. Such an ancestor exists since $R_v$ itself contains neither $u$ nor $x$. Thus, the sibling $R$ of $Q$ in $\mathcal{T}$ contains a vertex $i \in \{u, x\}$. We find a descendant $R_i$ of $R$ that is in the $r$-division and contains $i$. We then find any piece $R_j$ of the $r$-division containing the vertex $j \in \{u, x\}$ that is not $i$. Note that, by choice of $Q$, $R_j$ is not a descendant of $Q$. Finding these pieces requires time $O(\log^2 n)$.

3. We perform query $u$-to-1-Boundary$(u, \{x\}, (R_u, R_x), \text{null})$. This takes time $O(\sqrt{r} \log^2 n)$ and returns $d_G(u, y, x)$ for each $y \in \partial R_u \cup \partial R_x$.

4. For each $y \in (\partial R_u \cup \partial R_x) \setminus \{x\}$, we obtain the distance from $y$ to $v$ in $G \setminus (R_u \cup R_x \setminus \{y\})$ as follows. For each hole $h$ of $Q$, we perform an $O(\log^2 n)$-time point location query to the Voronoi diagram stored for $R_u, R_x, y$, and $h$. Then, we take the minimum returned additive distance over all holes of $Q$. Over all vertices of $Y$ this takes $O(\sqrt{r} \log^2 n)$ time.
(5) Finally, we compose the partial results by setting
\[ d_G(u, v, x) := \min_{y \in Y} d_G(u, y, x) + d(y, v, (R_u \cup R_v) \setminus \{y\}). \]

Correctness. We now argue the correctness of the query algorithm. Let \( r \) be a shortest \( u \)-to-\( v \) path that avoids \( x \). If the distance is computed in the first step of the algorithm, then the correctness follows by the definition of \( u \)-to-1-Boundary queries. We henceforth suppose this is not the case. Let \( z \) be the last vertex of \( \rho \) that belongs to \( \partial R_u \cup \partial R_v \). Let \( h' \) be the hole of \( Q \) such that the last vertex \( s \) of \( \rho \) that belongs to \( \partial Q \) belongs to \( h' \). The distance \( d_G(u, z, x) \) from \( u \) to \( z \) in \( G \setminus \{x\} \) is computed by the FR-Dijkstra computation in step 3, while the distance from \( z \) to \( v \) in \( G \setminus ((R_i \cup R_j) \setminus \{z\}) \) is obtained from the query to the Voronoi diagram stored for \( R_u, R_v, z, \) and \( h' \) in step 4, since
\[
\min_{w \in \partial Q \setminus \partial h'} d_{Q'}(w, v) = \min_{w \in \partial Q \setminus \partial h'} \left( \omega_z(w) + d_Q(w, v) \right) = \min_{w \in \partial Q \setminus \partial h'} \left( d_G(z, w, (R_u \cup R_v) \setminus \{z\}) + d_Q(w, v) \right) = d_G(z, s, (R_u \cup R_v) \setminus \{z\}) + d_Q(s, v) = d(z, v, (R_u \cup R_v) \setminus \{z\}).
\]

It is easy to see that we do not obtain any distance that does not correspond to an actual path in \( G \setminus \{x\} \) and hence the correctness of the query algorithm follows.

Space complexity. We now show that the space complexity is \( O\left(\frac{n^{5/2}}{\sqrt{\log n}}\right) \). The space required for the data structure allowing us to efficiently perform \( u \)-to-1-Boundary queries is \( O\left(\frac{n^2}{\sqrt{\log n}}\right) \) by Lemma 5.4. We next analyze the space required for storing the Voronoi diagrams. We consider \( O(g(n, r, 2)) = O_1(n^{\frac{r}{2}}) \) pairs of pieces \((R_i, R_j)\), and for each of the \( O(\sqrt{r}) \) boundary vertices of each such pair we store, in the worst case, a Voronoi diagram for each of the \( O(1) \) holes of each sibling of the nodes in the root-to-\( R_i \) and root-to-\( R_j \) paths in \( T \). The total number of sites of all Voronoi diagrams we store for a pair of pieces can be upper bounded by \( O(\sqrt{n}) \) by noting that the number of sites of a Voronoi diagram for a piece at level \( \ell \) of \( T \) is \( O(\sqrt{n}/c_{1, \ell}^2) \) by Lemma 2.1. By Theorem 6.3, the space required to store a representation of a set of Voronoi diagrams with the functionality allowing for efficient point location queries for a piece \( P \), with sites the boundary vertices of \( P \) that lie on a hole \( h \) is \( O\left(\sum_{P \in T} (S_{P, h} + |P| \log |P|)\right) \), where \( S_{P, h} \) is the total cardinality of these sets of sites. Summing over all holes of all pieces \( P \), noting that \( \sum_{P \in T} \sum_h S_{P, h} = O\left(\frac{n^{5/2}}{\sqrt{\log n}} \cdot \sqrt{n}\right) = O\left(\frac{n^{5/2}}{\sqrt{\log n}}\right) \) by the above discussion, and using Proposition 2.2 to bound \( \sum_{P \in T} \sum_h |P| \log |P| \), we conclude that the total space required for all Voronoi diagrams is \( O\left(\frac{n^{5/2}}{\sqrt{\log n}} + n \log^2 n\right) \).

Preprocessing time. As for the preprocessing time, the precomputations of Lemma 5.4 take \( O\left(\frac{n^{5/2}}{\sqrt{\log n}}\right) \). The additive weights can be computed in time \( O\left(\frac{n^{5/2}}{\sqrt{\log n}} \log^3 n\right) \); see Lemma 7.2 in Section 7. As for efficiently constructing Voronoi diagrams, we apply Theorem 6.3 to construct all Voronoi diagrams corresponding to each of the \( O(1) \) holes of each piece as a batch. The time required by Theorem 6.3 is \( O(|P| \log |P|) \). Further, by Proposition 2.3, we have \( \sum_{P \in T} |P| \log |P| = O(n^2) \), and hence the stated bound follows.

6.3 Handling Multiple Failures
We now explain how to straightforwardly generalize the approach presented in the previous subsection to obtain oracles that can handle multiple failures.

Preprocessing.
(1) We perform the precomputations of Lemma 5.4 so that we can efficiently perform $u$-to-$k$-Boundary queries.

(2) For each $(k + 1)$-tuple of pieces $(R_i, \ldots, R_{i+k})$ of the $r$-division we compute and store the following. Let $\Sigma$ be the (possibly empty) set of separators $\sigma$ in the recursive decomposition that separate a piece into two subpieces $Q_{\sigma}$ and $R_{\sigma}$, such that for some $j, R_j \subseteq R_{\sigma}$ and none of the other pieces of the tuple is a subgraph of $Q_{\sigma}$. For each $\sigma \in \Sigma$, for each $y \in \bigcup_{j=1}^{k+1} \partial R_j$, for each hole $h$ of $Q_{\sigma}$, we store a Voronoi diagram with the point location data structure for $Q_{\sigma}$, with sites the boundary vertices of $Q_{\sigma}$ that lie on $h$, and additive weights the distances from $y$ to these sites in $G \setminus \left( \bigcup_{j=1}^{k+1} \partial R_j \setminus \{y\} \right)$.

Query. The algorithm is then essentially the same as that of Section 6.2. We first retrieve a piece $R_v$ of the $r$-division that contains $v$.

(1) First, if $R_v$ contains any vertex from $\{u\} \cup X$, we can efficiently retrieve at most $k + 1$ $r$-division pieces $R_1, \ldots, R_m$ whose union contains all the vertices in $\{v\} \cup \{u\} \cup X$. Hence, in this case we can answer the distance query in $O(k \sqrt{r} \log^2 n)$ time by performing query $u$-to-$k$-Boundary$(u, \{x\}, (R_1, \ldots, R_m), v)$. We thus henceforth assume that $R_v$ is disjoint from $\{u\} \cup X$.

(2) We find the highest weak ancestor $Q$ of $R_v$ in $T$ that does not contain any of the elements of $\{u\} \cup X$ and retrieve a piece $R_i$ of the $r$-division that is a descendant of the sibling of $Q$ and contains some element $i \in \{u\} \cup X$. Note that $Q$ is well-defined since $R_v$ itself is disjoint from $\{u\} \cup X$. We then identify a piece $R_j$ in the $r$-division for each $j \in \{u\} \cup X \setminus \{i\}$. This requires time $O(k \log^2 n)$.

(3) We perform a $u$-to-$k$-Boundary$(u, X, R, \text{null})$ query for $R = (R_u, R_X, \ldots, R_X)$. We thus compute all $u$-to-$Y$ distances in $G \setminus X$, where $Y = \bigcup_{i \in R} \partial P \setminus X$, in time $O(k \sqrt{r} \log^2 n)$.

(4) For each vertex $y \in Y$, we obtain the distance from $y$ to $v$ in $G \setminus (\bigcup_{P \in R} \partial P \setminus \{y\})$ as follows. For each hole $h$ of $Q$, we perform an $O(\log^2 n)$-time point location query to the Voronoi diagram stored for $R, y$, and $h$. Then, we take the minimum returned additive distance over all holes of $Q$. Over all vertices of $Y$ this takes $O(k \sqrt{r} \log^2 n)$ time.

(5) Finally, we compose the partial results by setting

$$d_G(u, v, X) := \min_{y \in Y} d_G(u, y, X) + d(y, v, G \setminus (\bigcup_{P \in R} \partial P \setminus \{y\})).$$

We hence obtain our second tradeoff theorem, restated here for convenience.

**Theorem 6.1.** For any integer $r \in [1, n]$ and for any integer $k \leq \frac{n}{r}$, there exists a data structure of size $O\left(\frac{\sqrt{k+1}}{\sqrt{r+1}} + \sqrt{k+1} \log^2 n\right)$, which can be constructed in time $O\left(\frac{\sqrt{k+1}}{\sqrt{r+1}} + \sqrt{k+1} \log^2 n\right)$, and can answer the following queries in $O(k \sqrt{r} \log^2 n)$ time. Given vertices $u$ and $v$ and a set $X$ of at most $k$ failed vertices, report the length of a shortest $u$-to-$v$ path that avoids $X$.

**Proof.** The correctness of the query algorithm follows by an argument identical to the one for the case of single failures (see Section 6.2); its time complexity is analyzed above. We next analyze the space required by our data structure and its construction time.

**Space complexity.** The space occupied by the data structure of Lemma 5.4 is $O(\frac{\sqrt{k+1}}{\sqrt{r+1}} \log n)$. We bound the space required for the Voronoi diagrams by $O(g(n, r, k + 1)k \sqrt{nk} + n \log^2 n)$ as follows. For each of the $O(k \sqrt{r})$ boundary vertices of each of the $O(g(n, r, k + 1)) (k + 1)$-tuples, we store a Voronoi diagram for each of the $O(1)$ holes, of (at most) each of the siblings of the nodes in the root-to-$R_i$ path in $T$ for each $R_i$ in the tuple. With an argument identical to the one used in the proof of Lemma 3.3, the total number of boundary vertices (with multiplicities) of all of these pieces is $O(\sqrt{kn})$. Hence the total number of sites of all Voronoi diagrams that we store is $O(g(n, r, k + 1)k \sqrt{nk})$. By Theorem 6.3, the size required to store them with the required functionality is $O(g(n, r, k + 1)k \sqrt{nk})$. Therefore, the total space required is $O\left(\frac{\sqrt{k+1}}{\sqrt{r+1}} + \sqrt{k+1} \log^2 n\right)$.
thus \(O(g(n, r, k + 1)k\sqrt{nk}r + \sum_{P \in T} |P| \log |P|) = O \left(\frac{(cn)^{k+1}}{r^{k+1}} \cdot \frac{1}{k!} \cdot \sqrt{nk}r + n \log^2 n\right)\), where the last equality follows by Proposition 2.2.

Thus, since \(k \leq n/r\), the total space required is

\[
O \left(\frac{(cn)^{k+1}}{r^{k+1}} \cdot \frac{1}{k!} \cdot (kr + \sqrt{nk}r) + n \log^2 n\right) = O \left(\frac{(cn)^{k+1}}{r^{k+1}} \cdot \frac{1}{k!} \cdot \sqrt{nk}r + n \log^2 n\right).
\]

**Preprocessing time.** The preprocessing of Lemma 5.4 takes \(O(n^{k+1}/r\log^2 n)\) time. We can compute the required additive weights of all \((k+1)\)-tuples in time \(\tilde{O} \left(\frac{(cn)^{k+1}}{r^{k+1}} \cdot \frac{1}{(k-1)!} \cdot \sqrt{nk}r\right)\), employing Lemma 7.2. Finally, constructing the Voronoi diagrams requires time \(\tilde{O}(n^2 + S)\), where \(S\) is the total size of their representation. This can be achieved by applying Theorem 6.3 to construct all Voronoi diagrams corresponding to each of the \(O(1)\) holes of each piece as a batch. Let \(S_{P,h}\) be the number of sites (with multiplicities) of all Voronoi diagrams that we build for a hole \(h\) of a piece \(P\); in other words the product of the number of such diagrams and the number of vertices of \(\partial P\) that lie on \(h\). Then, the time required by Theorem 6.3 is \(\tilde{O}(|P||\partial P|^2 + S_{P,h})\). First, note that \(\sum_{P \in T} \sum_{h} S_{P,h} = O(S)\). Further, by Proposition 2.3, we have \(\sum_{P \in T} (|P||\partial P|^2) = O(n^2)\), and hence the stated bound follows.

\[\square\]

### 7 Efficient Preprocessing

In this section we show how to efficiently compute the data structures described in Sections 5 and 6. Throughout this section, and similarly to Section 3, when using FR-Dijkstra to compute DDG\(_s\)’s, or other distances corresponding to shortest paths with a restriction on the vertices they can go through, we do not relax edges whose tail is a vertex that is not allowed to be on a shortest path.

It is shown in [33, Theorem 3] that, given a geometrically increasing sequence of numbers \(\mathbf{v} = (r_1, r_2, \ldots, r_v)\), where \(r_1\) is a sufficiently large constant, \(r_{i+1}/r_i = b\), for all \(i\), for some constant \(b > 1\), and \(r_v = n\), we can obtain \(r\)-divisions for all \(r \in \mathbf{v}\) in time \(O(n)\) in total. These \(r\)-divisions satisfy the property that a piece in the \(r_i\)-division is a weak descendant (in \(T\)) of a piece in the \(r_j\)-division for each \(j > i\).

We first show how to efficiently compute the external DDG\(_s\)’s for all \(k\)-tuples of pieces of an \(r\)-division, \(r \in \mathbf{v}\). Our algorithm is a natural adaptation of the top-down technique of [7] for computing external DDGs to computing strictly external DDGs of \(k\)-tuples.

**Lemma 7.1.** Given \(r_i \in \mathbf{v}\) and an integer \(d \leq \frac{n}{r_i}\), one can compute DDG\(_{ext}\) for all \(d\)-tuples of pieces of each \(r_i\)-division, \(i \geq i\), in time \(O\left(\frac{(cn)^i}{r_i^{i+1}} \cdot \frac{1}{(d-1)!} \log^2 n\right)\) for some constant \(c > 1\).

**Proof.** We prove this lemma by induction on \(\mathbf{v}\) from top to bottom. For \(r_v = n\), the only piece is \(G\), and DDG\(_{ext}\) (G) is the empty graph. Assume inductively that we have DDG\(_{ext}\)(\(R_1, \ldots, R_d\)) for every \(d\)-tuple \((R_1, \ldots, R_d)\) of pieces at the \(r_{i+1}\)-division. Let \(Q_1, \ldots, Q_d\) be pieces at the \(r_i\)-division. Note that every piece at level \(r_i\) is contained in some piece at level \(r_{i+1}\), but a piece at level \(r_{i+1}\) might contain multiple pieces at level \(r_i\). Let \(R_1, \ldots, R_d\) be pieces of the \(r_{i+1}\)-division such that each \(Q_j\) is a subgraph of some \(R_j\); see Fig. 5 for an illustration. (If the \(r_{i+1}\)-division has less than \(d\) pieces we just take all of them.) Let \(Q_{R_j}\) be the maximal subset of \(|Q_1, \ldots, Q_d|\) such that each piece in \(Q_{R_j}\) is contained in \(R_j\). For every \(j \in \{1, \ldots, d\}\) let us denote the allowed internal part of \(R_j\) by \(R_j^I\). Formally,

\[
R_j^I = R_j \setminus \bigcup_{Q \in Q_{R_j}} Q \setminus \partial Q.
\]

ACM Trans. Algor.
Let us define the boundary of $R'_j$ to be

$$\partial R_j \cup \left( \bigcup_{Q \in QQ_{R_j}} \partial Q \right).$$

Since $R_j$ and each $Q_m \in Q_{R_j}$ have $O(\sqrt{t_{j+1}})$ and $O(\sqrt{t_j})$ boundary vertices respectively, $R'_j$ has $O(\sqrt{t_{j+1}} + \sqrt{t_j}|Q_{R_j}|) = O(|Q_{R_j}|/\sqrt{t_{j+1}})$ boundary vertices (recall that $r_{j+1}/r_j = b$).

Let $DDG_{R_j}^o$ be the a complete directed graph on the boundary vertices of $R'_j$ such that the edge $(u, v)$ has weight $d_{R_j}^o(u, v)$ equal to the length of the shortest $u$-to-$v$ path in $R'_j$ that is internally disjoint from the boundary of $R'_j$.

We compute $DDG_{R_j}^o$ in a similar manner to the query of Section 3 by running FR-Dijkstra on the union of the following $DDG$’s. For each piece $Q_m \in Q_{R_j}$, for each ancestor $Q$ of $Q_m$ (including $Q_m$) that is a strict descendant of $R_j$ in $T$, we take the $DDG^o_{Q_j}$ of the sibling $P$ of $Q$ if $P$ contains no piece of $Q_{R_j}$. The pieces of $Q_{R_j}$ have $O(|Q_{R_j}|/\sqrt{t_j})$ boundary vertices in total and the total number of boundary vertices for their considered ancestors is bounded by $O(|Q_{R_j}|/\sqrt{t_{j+1}})$, as the number of boundary vertices in any root-to-leaf path in $T$ decreases geometrically (cf. Lemma 2.1). Running FR-Dijkstra from each of the $O(|Q_{R_j}|/\sqrt{t_{j+1}})$ boundary vertices of $R'_j$ yields $DDG_{R_j}^o$ and requires $O(|Q_{R_j}|/\sqrt{t_{j+1}} |Q_{R_j}| |Q_{R_j}| |Q_{R_j}| \log^2 n) = O(|Q_{R_j}|^2 r_{j+1} \log^2 n)$ time in total. When summing over $R_1, \ldots, R_d$ we get

$$\sum_{j=1}^d |Q_{R_j}|^2 \cdot r_{j+1} \cdot \log^2 n \leq r_{j+1} \cdot \log^2 n \cdot \left( \sum_{j=1}^d |Q_{R_j}| \right)^2 = d^2 \cdot r_{j+1} \cdot \log^2 n.$$

Note that the equality follows from the fact that $\sum_{j=1}^d |Q_{R_j}| = d$.

Let $D = DDG_{ext}(R_1, \ldots, R_d) \cup \bigcup_{j=1}^d DDG_{R_j}^o$. Each of $DDG_{ext}(R_1, \ldots, R_d)$ and $\bigcup_{j=1}^d DDG_{R_j}^o$ contributes $O(d \sqrt{t_{j+1}})$ boundary vertices to $D$. We run FR-Dijkstra on $D$ from each boundary vertex of $Q_m$ for $m \in \{1, \ldots, d\}$ to obtain $DDG_{ext}(Q_1, \ldots, Q_d)$. There are $O(d \sqrt{t_j})$ such boundary vertices, so this requires time $O(d \sqrt{t_j} (\sqrt{t_{j+1}} + \sqrt{t_j}) \log^2 n) = O(d^2 \cdot r_{j+1} \cdot \log^2 n)$.
We can thus compute $DDG_{\text{ext}}^c(Q_1, \ldots, Q_d)$ for all $d$-tuples at level $r_i$ in time

$$O((g(n, r_i, d) \cdot d^2 \cdot r_{i+1} \cdot \log^2 n) = O\left(\frac{(cn)^d}{r^d_i} \cdot r_{i+1} \cdot \frac{1}{d!} \cdot d^2 \cdot \log^2 n\right) = O\left(\frac{(cn)^d}{r^d_i} \cdot \frac{1}{(d-2)!} \cdot \log^2 n\right),$$

assuming that we have the $DDG_{\text{ext}}^c$s for all $d$-tuples of pieces of $r_{t}$-divisions, $t > i$.

The time to compute the $DDG_{\text{ext}}^c$s for all $d$-tuples of pieces of all $r_i$-divisions, $t > i$, is, inductively,

$$O\left((cn)^d \cdot \frac{1}{(d-2)!} \cdot \log^2 n \cdot \sum_{i=1}^{n} \frac{1}{r^d_{i+1}}\right), \text{ and } \sum_{i=1}^{n} \frac{1}{r^d_{i+1}} = \frac{1}{r^d_{i+1}} \sum_{i=1}^{n} \left(\frac{1}{b^d_{i+1}}\right)^t = O\left(\frac{1}{r^d_{i+1}}\right)$$

since $b^{d-1} > 1$. Thus, computing the $DDG_{\text{ext}}^c$s for $d$-tuples of pieces of the $r_i$-division dominates the time complexity.

We next show how to efficiently compute the additive distances with respect to which the Voronoi diagrams stored by our oracle are computed.

**Lemma 7.2.** Let $\mathcal{R}_r$ be an $r$-division, such that $r \in \mathcal{V}$, and let $d \leq \frac{n}{r}$ be an integer. For all $d$-tuples of pieces $R_1, \ldots, R_d$ in $\mathcal{R}_r$ and for all pieces $Q \in T$ such that $Q$ does not contain any of the pieces $R_i$, and $Q$ is a sibling of a node in the root-to-$R_i$ path in $T$ for some $R_i$, one can compute the distances from each $y \in \bigcup_{i=1}^{d} \partial R_i$ to each boundary vertex of $Q$ in the graph $G \setminus \left(\bigcup_{i=1}^{d} R_i\right)$ \{ $\{y\}$ in time $O\left(\frac{(cn)^d}{r^d} \cdot \frac{1}{(d-2)!} \cdot \sqrt{nr} \cdot \log^3 n\right)$ in total, for some constant $c > 1$.

**Proof.** Let us consider a $d$-tuple of pieces $(R_1, \ldots, R_d)$ and a piece $Q$, satisfying the properties in the statement of the lemma. To compute the desired distances, we run FR-Dijkstra from each $y \in \bigcup_{i=1}^{d} \partial R_i$ on the union of the following DDGs:

1. $DDG_{Q}^c$.
2. For each piece $R_i = \{R_1, \ldots, R_d\}$ for each ancestor $A$ of $R_i$ (including $R_i$) in $T$, we take the $DDG_{Q}^c$ of the sibling $B$ of $A$ if $B$ contains no piece of $R_1, \ldots, R_d$.

This correctly computes the distances by the same arguments that were applied in Section 3. It remains to analyze the time complexity. Consider the $(n/d)$-division of $G$ in $T$. By the same argument that was applied in the proof of Lemma 3.3 we can bound the number of boundary vertices for all the included $DDG^s$ by $O(\sqrt{dn})$. There are $O(d \sqrt{r})$ choices of $y \in \bigcup_{i=1}^{d} \partial R_i$, so the time required to run FR-Dijkstra from each $y$ is $O(d \sqrt{r} \cdot \sqrt{dn} \cdot \log^2 n) = O(d \cdot \sqrt{nr} \cdot \log^2 n)$.

Each piece $R_i = \{R_1, \ldots, R_d\}$ has $O(\log n)$ nodes in the root-to-$R_i$ path in $T$, hence computing the distances for all possible choices of $Q$ requires time $O(d^2 \sqrt{nr} \cdot \log^3 n)$. Finally, in order to compute the distances for all $d$-tuples of pieces we need time

$$O((g(n, r, d) \cdot d^2 \cdot \sqrt{nr} \cdot \log^3 n) = O\left(\frac{(cn)^d}{r^d} \cdot \frac{1}{d!} \cdot d^2 \sqrt{nr} \cdot \log^3 n\right),$$

as claimed.

**8 FINAL REMARKS**

Perhaps the most intriguing open question related to our results is whether it is possible to answer distance queries subject to even one failure in time $O(1)$ with an $o(n^2)$-size oracle.
ACKNOWLEDGMENTS

This work was partially supported by Israel Science Foundation (ISF) grants 794/13, 592/17, and 810/21.

REFERENCES

[1] Amir Abboud and Søren Dahlgaard. 2016. Popular Conjectures as a Barrier for Dynamic Planar Graph Algorithms. In 57th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2016. 477–486. https://doi.org/10.1109/FOCS.2016.58

[2] Ittai Abraham, Shiri Chechik, and Cyril Gavoille. 2012. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In 44th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2012. 1199–1218. https://doi.org/10.1145/2213977.2214084

[3] Srinivasa Rao Arikati, Danny Z. Chen, L. Paul Chew, Gautam Das, Michael H. M. Smid, and Christos D. Zaroliagis. 1996. Planar Spanners and Approximate Shortest Path Queries among Obstacles in the Plane. In 4th Annual European Symposium on Algorithms, ESA 1996. 514–528. https://doi.org/10.1007/3-540-61680-2_79

[4] Aviv Bar-Natan, Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2021. Fault-Tolerant Distance Labeling for Planar Graphs. In 28th International Colloquium on Structural Information and Communication Complexity, SIROCCO 2021. 315–333. https://doi.org/10.4230/LIPIcs.SIROCCO.2021.315

[5] Surender Baswana, Utkarsh Lath, and Anuradha S. Mehta. 2012. Single source distance oracle for planar digraphs avoiding a failed node or link. In 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012. 221–232. https://doi.org/10.1137/1.9781611973099.20

[6] Aaron Bernstein and David R. Karger. 2009. A nearly optimal oracle for avoiding failed vertices and edges. In 41st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2009. 101–110. https://doi.org/10.1145/1536414.1536431

[7] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. 2015. Min st-Cut Oracle for Planar Graphs with Near-Linear Preprocessing Time. ACM Trans. Algorithms 11, 3 (2015), 16:1–16:29. https://doi.org/10.1145/2734707.2734722

[8] Sergio Cabello. 2012. Many Distances in Planar Graphs. Algorithmica 62, 1-2 (2012), 361–381. https://doi.org/10.1007/s00453-010-9459-0

[9] Sergio Cabello, Erin W. Chambers, and Jef Erickson. 2013. Multiple-Source Shortest Paths in Embedded Graphs. SIAM J. Comput. 42, 4 (2013), 1542–1571. https://doi.org/10.1137/120864271

[10] Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2019. Almost optimal distance oracles for planar graphs. In 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019. 138–151. https://doi.org/10.1145/3313276.3316316

[11] Panagiotis Charalampopoulos and Adam Karczmarz. 2020. Single-Source Shortest Paths and Strong Connectivity in Dynamic Planar Graphs. In 28th Annual European Symposium on Algorithms, ESA 2020. 31:1–31:23. https://doi.org/10.4230/LIPIcs.ESA.2020.31

[12] Panagiotis Charalampopoulos, Shay Mozes, and Benjamin Tebeka. 2019. Exact Distance Oracles for Planar Graphs with Failing Vertices. In 30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019. 2110–2123. https://doi.org/10.1137/1.9781611975482.127

[13] Danny Z. Chen and Jinhui Xu. 2000. Shortest path queries in planar graphs. In Algorithmica 28, 2 (1998), 652–673. https://doi.org/10.1007/s00453-010-9459-0

[14] Ittai Abraham, Shiri Chechik, and Cyril Gavoille. 2012. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In 44th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2012. 1199–1218. https://doi.org/10.1145/2213977.2214084

[15] Srinivasa Rao Arikati, Danny Z. Chen, L. Paul Chew, Gautam Das, Michael H. M. Smid, and Christos D. Zaroliagis. 1996. Planar Spanners and Approximate Shortest Path Queries among Obstacles in the Plane. In 4th Annual European Symposium on Algorithms, ESA 1996. 514–528. https://doi.org/10.1007/3-540-61680-2_79

[16] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. 2015. Min st-Cut Oracle for Planar Graphs with Near-Linear Preprocessing Time. ACM Trans. Algorithms 11, 3 (2015), 16:1–16:29. https://doi.org/10.1145/2734707.2734722

[17] Sergio Cabello. 2012. Many Distances in Planar Graphs. Algorithmica 62, 1-2 (2012), 361–381. https://doi.org/10.1007/s00453-010-9459-0

[18] Sergio Cabello, Erin W. Chambers, and Jef Erickson. 2013. Multiple-Source Shortest Paths in Embedded Graphs. SIAM J. Comput. 42, 4 (2013), 1542–1571. https://doi.org/10.1137/120864271

[19] Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2021. Fault-Tolerant Distance Labeling for Planar Graphs. In 28th International Colloquium on Structural Information and Communication Complexity, SIROCCO 2021. 315–333. https://doi.org/10.4230/LIPIcs.SIROCCO.2021.315

[20] Surender Baswana, Utkarsh Lath, and Anuradha S. Mehta. 2012. Single source distance oracle for planar digraphs avoiding a failed node or link. In 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012. 221–232. https://doi.org/10.1137/1.9781611973099.20

[21] Aaron Bernstein and David R. Karger. 2009. A nearly optimal oracle for avoiding failed vertices and edges. In 41st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2009. 101–110. https://doi.org/10.1145/1536414.1536431

[22] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. 2015. Min st-Cut Oracle for Planar Graphs with Near-Linear Preprocessing Time. ACM Trans. Algorithms 11, 3 (2015), 16:1–16:29. https://doi.org/10.1145/2734707.2734722

[23] Srinivasa Rao Arikati, Danny Z. Chen, L. Paul Chew, Gautam Das, Michael H. M. Smid, and Christos D. Zaroliagis. 1996. Planar Spanners and Approximate Shortest Path Queries among Obstacles in the Plane. In 4th Annual European Symposium on Algorithms, ESA 1996. 514–528. https://doi.org/10.1007/3-540-61680-2_79

[24] Viktor Fredslund-Hansen, Shay Mozes, and Christian Wulff-Nilsen. 2021. Truly Subquadratic Exact Distance Oracles with Constant Query Time for Planar Graphs. In 32nd International Symposium on Algorithms and Computation, ISAAC 2021. 25:1–25:12. https://doi.org/10.4230/LIPIcs.ISAAC.2021.25

ACM Trans. Algor.
[25] Paweł Gawrychowski, Haim Kaplan, Shay Mozes, Micha Sharir, and Oren Weimann. 2021. Voronoi Diagrams on Planar Graphs, and Computing the Diameter in Deterministic $O(n^{3/2})$ Time. *SIAM J. Comput.* 50, 2 (2021), 509–554. https://doi.org/10.1137/18M1193402

[26] Paweł Gawrychowski and Adam Karczmarz. 2018. Improved Bounds for Shortest Paths in Dense Distance Graphs. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*. 61:1–61:15. https://doi.org/10.4230/LIPIcs.ICALP.2018.61

[27] Paweł Gawrychowski, Shay Mozes, Oren Weimann, and Christian Wulff-Nilsen. 2018. Better Tradeoffs for Exact Distance Oracles in Planar Graphs. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*. 515–529. https://doi.org/10.1137/1.9781611975031.34

[28] John Hershberger and Subhash Suri. 2001. Vickrey Prices and Shortest Paths: What is an Edge Worth?. In *42nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2001*. 252–259. https://doi.org/10.1109/SFCS.2001.959899

[29] Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. 2011. Improved algorithms for min cut and max flow in undirected planar graphs. In *23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012*. 209–222. https://doi.org/10.1137/1.9781611973099.19

[30] Noam Nisan and Amir Ronen. 1999. Algorithmic Mechanism Design (Extended Abstract). In *31st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*. 129–140. https://doi.org/10.1137/100795276

[31] Jan van den Brand and Thatchaphol Saranurak. 2019. Sensitive Distance and Reachability Oracles for Large Batch Updates. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*. 424–435. https://doi.org/10.1109/FOCS.2019.00034

[32] Christian Wulff-Nilsen. 2010. Solving the Replacement Paths Problem for Planar Directed Graphs in $O(n \log n)$ Time. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*. 736–745. https://doi.org/10.1137/1.9781611973075.62

ACM Trans. Algor.