Fully Dynamic Algorithms for Graph Spanners via Low-Diameter Router Decomposition

Julia Chuzhoy*
Toyota Technological Institute at Chicago
USA

Merav Parter†
Weizmann Institute of Science
Israel

Abstract

A t-spanner of an undirected n-vertex graph G is a sparse subgraph H of G that preserves all pairwise distances between its vertices to within multiplicative factor t, also called the *stretch*. Spanners play an important role in the design of efficient algorithms for distance-based graph optimization problems, as they allow one to sparsify the graph, while approximately preserving all distances. It is well known that any n-vertex graph admits a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges, and that this stretch-size tradeoff is optimal assuming the Erdös Girth Conjecture. In this paper we investigate the problem of efficiently maintaining spanners in the fully dynamic setting with an adaptive adversary. Despite a long and intensive line of research, this problem is still poorly understood: for example, no algorithm achieving a sublogarithmic stretch, with a sublinear in n update time, and a strongly subquadratic in n bound on the size of the spanner is currently known in this setting. One of our main results is a *deterministic* (and therefore, adaptive-adversary) algorithm, that, for any $512 \leq k \leq (\log n)^{1/49}$ and $1/k \leq \delta \leq 1/400$, maintains a spanner H of a fully dynamic graph with stretch $\mathrm{poly}(k) \cdot 2^{O(1/\delta^6)}$ and size $|E(H)| \leq O(n^{1+O(1/k)})$, with worst-case update time $n^{O(\delta)}$ and recourse $n^{O(1/k)}$.

Our algorithm relies on a new technical tool that we develop, and that we believe to be of independent interest, called *low-diameter router decomposition*. Specifically, we design a deterministic algorithm that maintains a decomposition of a fully dynamic graph into edge-disjoint clusters with bounded vertex overlap, where each cluster C is guaranteed to be a bounded-diameter router, meaning that any reasonable multicommodity demand over the vertices of C can be routed along short paths and with low congestion inside C. A similar graph decomposition notion was introduced by [Haeupler et al., STOC 2022] and recently strengthened by [Haeupler et al., FOCS 2024]; the latter result was already used to obtain fast algorithms for multicommodity flows [Haeupler et al., STOC 2024] and dynamic distance oracles [Haeupler et al., FOCS 2024]. However, in contrast to these and other prior works, the decomposition that our algorithm maintains is guaranteed to be *proper*, in the sense that the routing paths between the pairs of vertices of each cluster C are contained inside C (rather than in the entire graph G). Additionally, our algorithm maintains, for each cluster C of the decomposition, a subgraph $C' \subseteq C$ of a prescribed density, that also has strong routing properties.

We show additional applications of our low-diameter router decomposition, by obtaining new deterministic dynamic algorithms for fault-tolerant spanners and low-congestion spanners. Several of these applications crucially rely on the fact that our router decomposition is proper.

1 Introduction

A spanner of a graph G is a sparse subgraph of G, that approximately preserves all pairwise distances between its vertices. The notion of spanners naturally falls within the broader class of graph-theoretic objects called *graph sparsifiers*. A sparsifier of a given graph G is a sparse subgraph of G, that approximately preserves some central properties of G, such as, for example, cut values, flows, or, in this case, distances. A natural and powerful paradigm for designing fast algorithms for graph optimization problems is to first compute a sparsifier H of the input graph G, that preserves properties of G relevant to the problem, and then solve the problem on the much

*Supported in part by NSF grant CCF-2402283 and NSF HDR TRIPODS award 2216899. Email: cjulia@ttic.edu

†This project is funded by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No. 949083). Email: merav.parter@weizmann.ac.il

sparser graph H. It is then not surprising that various types of sparsifiers increasingly play a central role in the design of fast algorithms for various combinatorial optimization problems, in static, dynamic, and other settings. Since their introduction by Peleg and Schaffer [PS89], spanners have been studied extensively in various settings (see, e.g. [PU87, EP01, EZ04, Bas06, BS07, Bas08, DGPV08, BKS12, GK18, DFKL21]), and have been used in a wide range of applications: from distance oracles [TZ01], shortest path computation [BK06] and routing schemes [PU89] to spectral sparsification [KP12].

Formally, given an n-vertex graph G with non-negative edge lengths, a k-spanner for G is a subgraph $H \subseteq G$, such that, for every pair u, v of vertices of G, $\mathsf{dist}_H(u,v) \leq k \cdot \mathsf{dist}_G(u,v)$. In addition to the stretch parameter k, a central measure of the quality of the spanner H is its size $|E(H)|$. It is well known that any n-vertex graph admits a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges, and this tradeoff is tight under the Erdös Girth Conjecture [Erd64]. In their influential paper, Baswana and Sen [BS07] presented a randomized algorithm for computing a $(2k-1)$-spanner of a static n-vertex and m-edge graph G, that contains $O(k \cdot n^{1+1/k})$ edges, in expected time of $O(km)$. Roditty, Thorup and Zwick [RTZ05] derandomized this construction, while achieving similar parameters.

In this paper we focus on the problem of efficiently maintaining a spanner of a fully dynamic graph G, that is, a graph undergoing an online sequence of edge insertions and deletions. As usual in the area of dynamic algorithms, we distinguish between the *oblivious-adversary* and the *adaptive-adversary* settings. In the case of oblivious adversary, it is assumed that the sequence of updates to the input graph G is fixed in advance, and may not depend on the algorithm's behavior and random choices. In contrast, in the *adaptive-adversary* setting, each update to the graph may depend on the algorithm's past behavior and inner state arbitrarily. It is often the case that the oblivious adversary assumption considerably limits the applicability of a dynamic algorithm. As an example, one common use of such algorithms is in speeding up algorithms for static problems, which typically requires that the dynamic algorithm can withstand an adaptive adversary. A large number of breakthrough result in recent years, that obtained fast algorithms for central graph optimization problems in the static setting, crucially rely on dynamic algorithms that can withstand an adaptive adversary. With this motivation in mind, we focus in this paper on the adaptive-adversary setting. We note that deterministic algorithms are always guaranteed to work against an adaptive adversary, and as such they are especially desirable. For brevity, we will refer to algorithms that can withstand oblivious or adaptive adversaries as *oblivious-update* or *adaptive-update* algorithms, respectively.

Dynamic Spanners Dynamic algorithms for graph spanners have been studied extensively, since the pioneering work of Ausiello et al. [ADF$^+$07]. This line of research can be partitioned into three main eras.

Era 1: Optimal Spanners with Low Amortized Update Time. Earlier work studied the problem of maintaining spanners of fully dynamic graphs with near-optimal tradeoff between the stretch and the size of the spanner, with the focus on optimizing the *amortized* update time, that is, the **average** amount of time that the algorithm spends in order to process each update to the input graph G. Ausiello, Fraciosa and Italiano [ADF$^+$07] provided deterministic dynamic algorithms for spanners with stretch 3 and 5, with amortized update time proportional to the maximum vertex degree in the graph. Elkin [Elk07] presented a randomized oblivious-update algorithm for maintaining $(2k-1)$-spanners for arbitrary values of k, with amortized update time $O(mn^{-1/k})$, where m is the number of edges in the initial graph; note that, for sufficiently dense graphs, this amortized update time may be superlinear in n. This line of work culminated with the result of Baswana, Khurana, and Sarkar [BKS12], that achieved expected amortized update time $\min\left\{O(k^2 \log^2 n), 2^{O(k)}\right\}$ against an oblivious adversary. Forster and Goranci [FG19] later improved the size and update time by factor k. The main drawbacks of these algorithms are that (i) their worst-case update time may be as high as $\Omega(n)$; and (ii) except for the work of [ADF$^+$07], all results cited above work in the oblivious-adversary setting only.

Era 2: Low Worst-Case Update Time against an Oblivious Adversary. Bodwin and Krinninger [BK16] designed the first algorithms for maintaining spanners of dynamic graphs with worst-case update time that is sublinear in n, for the special case of stretch values 3 and 5. Specifically, they design randomized oblivious-update algorithms that maintain 3-spanners with $\widetilde{O}(n^{3/2})$ edges and worst-case update time $\widetilde{O}(n^{3/4})$, and 5-spanners with $\widetilde{O}(n^{4/3})$ edges and worst-case update time $\widetilde{O}(n^{5/9})$. Later, Bernstein, Forster and Henzinger [BFH19] significantly narrowed the gap between amortized and worst-case update time guarantees by "deamortizing" the result of Baswana et al. [BKS12]. Their algorithms are based on a general technique for randomized data structures, that

converts amortized update time guarantees to worst-case ones. This approach yields a randomized oblivious-update algorithm for dynamic k-spanners with $k = O(1)$, with worst-case expected update time $2^{O(k)} \log^3 n$, nearly matching the amortized update time bound of [BKS12].

A key limitation of the algorithm of [BFH19], as well as all of the above-mentioned previous dynamic algorithms for stretch values $k > 5$, is that they are randomized, and more crucially, they only work against an *oblivious adversary*. As mentioned above, this significantly limits the applicability of such algorithms, and addressing this drawback became the next major goal in this area.

Era 3: The Quest for Adaptive-Update Algorithms. Bernstein et al. [BvdBG+22] provided the first dynamic algorithms for maintaining spanners that can withstand an adaptive adversary for stretch values $k > 5$. Their algorithm is randomized, and with high probability maintains a spanner with stretch $O(\text{poly} \log n)$ and size $\widetilde{O}(n)$, with $O(\text{poly} \log n)$ *amortized* update time. Their algorithm can also be adapted to obtain worst-case update time at most $n^{o(1)}$, at the cost of increasing the stretch bound to $n^{o(1)}$. Their approach relies on expander decomposition, and as such is unlikely to lead to the construction of spanners with sublogarithmc stretch, as was also noted in [BSS22]. Lastly, the recent work of [CKL+22, vdBCK+23] provided a deterministic algorithm for maintaining an $n^{o(1)}$-spanner of a fully dynamic graph, whose size is $n^{1+o(1)}$, and worst-case update time bound is at least as high as $n^{o(1)}$ times the maximum vertex degree in the graph. Their algorithm has several other important properties, such as, for example, it has a low recourse, and it maintains a low-congestion embedding of G into the sparsifier, which we discuss below. This algorithm, in turn, serves as one of the main subroutines used in their breakthrough almost-linear time algorithm for min-cost flow, and it was also used in a recent work of [KMPG24] for obtaining low-congestion vertex sparsifiers and algorithms for dynamic All-Pairs Shortest Paths. We provide a summary of known results for fully-dynamic spanners in Table 1. To summarize, to the best of our knowledge, the following fundamental question still remains open:

Question 1.1. *Is there an adaptive-update algorithm for maintaining a spanner of an n-vertex dynamic graph with any stretch value $k < o(\log n)$, whose size is at most $n^{2-\epsilon}$ for any constant ϵ, and worst-case (or even amortized) update time is $o(n)$?*

Dynamic Spanners with Small Recourse. An additional desired property of dynamic algorithms that maintain spanners (as well as other types of sparsifiers) is to ensure that the resulting spanner changes slowly, as the input graph G undergoes updates. The reason is that, as mentioned already, a commonly used paradigm is to apply algorithms for various distance-based problems to the spanner rather than the original graph, in order to obtain faster running time guarantees. If, however, following an update to the input graph G, the resulting spanner H may undergo a large number of changes, then it is unlikely that such an algorithm may provide low worst-case update time guarantees. This naturally leads to the notion of a *recourse*: the largest number of edge insertions and deletions that the spanner H may undergo after each update to the input graph G. This measure naturally plays an important role in dynamic algorithms, see e.g. [CHK16, BC18, AOSS19, CZ19]. Bhattacharya, Saranurak and Sukprasert [BSS22] presented a deterministic (and therefore, adaptive-update) algorithm for maintaining $(2k-1)$ spanners with near-optimal size, $O(\log n)$ amortized recourse and $\text{poly}(n)$ worst-case update time. The algorithm of [CKL+22, vdBCK+23] mentioned above also has recourse at most $n^{o(1)}$.

Our first main result answers Question 1.1 in the affirmative, by providing an algorithm for maintaining dynamic spanners that also has a low recourse:

Theorem 1.1. (Dynamic Deterministic Spanners, Informal) *There is a deterministic algorithm, whose input is an n-vertex simple undirected graph G with integral poly-bounded edge lengths, a stretch parameter $512 \leq k \leq (\log n)^{1/49}$, and an additional parameter $1/k \leq \delta \leq 1/400$. Graph G initially has no edges, and it undergoes an online sequence of edge insertions and deletions. The algorithm maintains a spanner H of G with stretch $k^{O(1)} \cdot 2^{O(1/\delta^6)}$ and $|E(H)| \leq O(n^{1+O(1/k)})$. The worst-case update time of the algorithm is $n^{O(\delta)}$, and its recourse is $n^{O(1/k)}$. The algorithm can also be extended to maintain fault-tolerant spanners, connectivity certificates, and low-congestion spanners.*

Stretch	Spanner Size	Update Time	[W/A]	Deterministic?	Citation
$(2k-1)$	$O\left(k \cdot n^{1+1/k}\right)$	$O(m \cdot n^{-1/k})$	[A]	Oblivious	[Elk07]
$(2k-1)$	$O\left(kn^{1+1/k} \log n\right)$	$\min\{O(k^2 \log^2 n), 2^{O(k)}\}$	[A]	Oblivious	[BKS12]
$(2k-1)$	$O\left(n^{1+1/k} \log n\right)$	$O(k \log^2 n)$	[A]	Oblivious	[FG19]
$(2k-1)$	$O\left(k \cdot n^{1+1/k}\right)$	$2^{O(k)} \cdot \log^3 n$	[W]	Oblivious	[BFH19]
$\text{poly} \log n$	$\widetilde{O}(n)$	$\text{poly} \log n$	[A]	Adaptive	[BvdBG+22]
$n^{o(1)}$	$O\left(n^{1+o(1)}\right)$	$n^{o(1)}$	[W]	Adaptive	[BvdBG+22]
$n^{o(1)}$	$O\left(n^{1+o(1)}\right)$	$n^{o(1)}$	[W]	Deterministic	[vdBCK+23]
$\text{poly}(k) \cdot 2^{c/\delta^6}$	$O\left(n^{1+O(1/k)}\right)$	$n^{O(\delta)}$	[W]	Deterministic	**(this paper)**

Table 1: Summary of known algorithms for fully dynamic spanners. Amortized update time is marked by [A] and worst-case update time by [W]. In the penultimate column we mark wherther the algorithm is deterministic, and, if it is randomized, whether it works against adaptive or oblivious adversary. Our result holds for any $512 \leq k \leq (\log n)^{1/49}$ and $1/k \leq \delta \leq 1/400$.

At the heart of our approach is a new algorithmic tool that we believe is of independent interest, that we call a *low-diameter router decomposition*, or just a *router decomposition* for brevity. Given an n-vertex graph G and a parameter k, a router decomposition of G consists of a collection $\mathcal{C}$ of edge-disjoint *router* subgraphs of G, that we call clusters, each of which has a small diameter, such that $\sum_{C \in \mathcal{C}} |V(C)| \leq n^{1+O(1/k)}$ holds, and the vast majority of the edges of G lie in some cluster of $\mathcal{C}$. The notion of routers was introduced in a very recent work of [HHT24, HHL+24], as an object that is closely related to length-constrained expanders of [HRG22, HHG22]. Routers are also closely related to well-connected graphs introduced in [Chu23]; we discuss this connection in more detail in Section 2.4. Intuitively, a graph G is a router, if any "reasonable" multicommodity demand can be routed in G via short paths (the length is typically sublogarithmic in n), and with low congestion. One example of a reasonable demand is where the total demand on every vertex is bounded by its degree.

Before we continue, we need to define the notion of *demands* and their *routing*. A *demand* $\mathcal{D}$ in a graph G consists of a collection Π of pairs of vertices of G, and, for every pair $(a,b) \in \Pi$, a demand value $D(a,b) > 0$. For a vertex $v \in V(G)$, the *demand on* v is the sum of demands $D(u,v)$ over all pairs $(u,v) \in \Pi$ containing v. We say that the demand $\mathcal{D}$ is *unit*, if the demand on every vertex v is bounded by its degree, and we say that it is Δ-bounded, for some value $\Delta > 0$, if the demand on every vertex v is at most Δ. Lastly, we say that demand $\mathcal{D}$ is *h-length* if, for every pair $(a,b) \in \Pi$ of vertices, $\mathsf{dist}_G(u,v) \leq h$. The *routing* of a demand $\mathcal{D}$ is a flow f, where, for every pair $(a,b) \in \Pi$, $D(a,b)$ flow units are sent from a to b. The *congestion* of the routing is the maximum amount of flow through any edge of G. We say that the flow is over paths of length at most d, if every flow-path P with $f(P) > 0$ has length at most d. We say that an algorithm for a static problem has *almost linear* running time, if its running time is $m^{1+o(1)}$, where m is the number of edges in the input graph, and we say that its running time is *near linear*, if it is bounded by $\tilde{O}(m)$; throughout, $\tilde{O}$ is used to hide $\text{poly} \log n$ factors. We now provide an overview of recent work on expander decompositions, routers and related graph notions.

Expanders, Routers and Graph Decompositions. Expanders are a fundamental notion in both graph theory and algorithms. Thanks to a number of powerful algorithmic techniques that were developed for expanders over the years, they now play a central role in the design of efficient algorithms for a wide variety of graph problems, in static, dynamic, and other settings. One of the central properties that make the expanders useful in many applications is that, as shown in the seminal work of Leighton and Rao [LR99], they have strong routing properties, in the sense that any unit demand can be routed across an expander with polylogarithmic congestion, via paths of polylogarithmic length. An almost-linear time algorithm for computing such routing was shown in [GKS17, GL18].

Another central tool in this area is *expander decomposition*, that decomposes the input graph G into vertex-disjoint subgraphs called clusters, such that each cluster is a strong enough expander, and only a small number of edges of G connect vertices that lie in different clusters. Expander decompositions are often used in order to reduce a given optimization problem in general graphs to that on expanders, which, due to the many useful properties of expanders, is often more tractable. Some examples of algorithmic results relying on expander decomposition include graph sketching and sparsification [ACK+16, JS18, CGP+20], dynamic algorithms for minimum spanning

forest [NS17, WN17], and recent breakthrough almost-linear time algorithms for Min-Cost Flow in directed graphs [CKL$^+$22, vdBCK$^+$23], to name just a few.

In their seminal work, Spielman and Teng [ST04] provided the first almost-linear time algorithm that computes a decomposition of a graph into clusters, with somewhat weaker guarantees than the expander decomposition: their decomposition only guarantees that every cluster is contained inside some expander. In other words, if we are given, for every cluster C in the decomposition $\mathcal{C}$, a unit demand $\mathcal{D}_C$, then all these demands can be routed in G simultaneously via short paths and with low congestion, but the routing of the demand $\mathcal{D}_C$ for a cluster C may not be confined to C, and may use edges and vertices that lie outside of C. While their decomposition algorithm found many applications, the use of this weakened notion of expander decomposition makes some of the resulting algorithms quite complex. Moreover, in some applications, this weakened notion of expander decomposition is not sufficient. The work of Saranurak and Wang [SW19] overcomes this shortcoming, by providing a near-linear time randomized algorithm for computing a proper (or strong) expander decomposition. This so-called "user-friendly" decomposition led to simplifying the analysis of existing algorithms, e.g., [ST04, KLOS14, CKP$^+$17, JS18], and to new applications that crucially require the stronger guarantee of the proper decomposition, that include algorithms for dynamic minimum spanning forest [NSWN17, Wul17, NSW17] and short-cycle decomposition [CGP$^+$20]. Deterministic almost-linear time algorithms for computing proper expander decompositions were later developed by [GLN$^+$19, CGL$^+$20, Chu23].

While expanders provide a valuable and powerful tool for the design of algorithms, they have some shortcomings. One such central shortcoming is that the diameter of an n-vertex expander may be as large as logarithmic in n, and moreover, given a unit demand $\mathcal{D}$, one can only guarantee its routing with low congestion via paths of polylogarithmic length, even if all pairs of vertices with non-negative demand are very close to each other. One implication of this shortcoming is that, when relying on expanders for distance-based problems, such as, for example, dynamic All-Pairs Shortest Paths (APSP), distance oracles, and graph spanners, it seems inevitable that one has to settle for super-logarithmic approximation guarantees. To address this challenge, and in order to generalize the successful paradigm of using expanders in distance-based problems to shorter distances, Ghaffari, Haeupler and Räcke [HRG22] introduced the notion *length-constrained (LC) expanders*. We provide here a *routing characterization* of length-constrained expanders, that was shown by [HHT24] to be equivalent, up to constant factors, to the original definition of [HRG22]. Under this definition, a graph G is an (h,s)-length φ-expander, if any h-length unit demand $\mathcal{D}$ can routed in G via paths of length $O(h\cdot s)$, with congestion at most $\widetilde{O}(1/\varphi)$. A somewhat similar notion of well-connected graphs was introduced in [Chu23], with the same motivation. They also designed a number of algorithmic tools for well-connected graphs, that may be thought of as analogues of similar tools for expanders, and that were recently used in [CZ23] to obtain the first adaptive-update algorithm for dynamic APSP with sublogaritmic approximation and amortized update time $n^{o(1)}$. We provide a formal definition of well-connected graphs and discuss their relationship with LC-expanders in Section 2.4.

Due to the success of the expander decompositions as a powerful tool in algorithm design, it is natural to ask whether one can also decompose a given graph into LC-expanders. Ghaffari, Haeupler and Räcke [HRG22] introduced the notion of LC-expander decomposition, that, in turn, relies on the notion of a *moving cut*. Given a graph G with lengths $\ell(e)\geq 0$ on its edges $e\in E(G)$ and integers h and s, an (hs)-length moving cut L assigns an integral *length increase* value $L(e)\in\{1,\ldots,(hs)\}$ to each edge $e\in E(G)$. A graph that is obtained from G by setting the length of every edge e to $\ell(e)+L(e)$ is denoted by $G-L$. The *size* of the (hs)-length moving cut L is $|L|=\frac{1}{hs}\cdot\sum_{e\in E(G)}L(e)$. An (h,s)-length φ-expander decomposition is simply an (hs)-length moving cut L, such that graph $G-L$ is an (h,s)-length φ-expander.

The work of [HRG22] provided a proof that there exists an (h,s)-length φ-expander decomposition of any n-vertex m-edge graph, for $s=O(\log n)$, such that the resulting (hs)-length moving cut has size $\tilde{O}(\varphi m)$. Additionally, they provide an algorithm for computing a weaker notion of an (h,s)-length φ-expander decomposition.

A very recent work by Haeupler, Hershkowitz and Tan [HHT24] provided an algorithm that, given an n-vertex m-edge graph G, with parameters $0<\epsilon<1$, h and φ, computes an (h,s)-length φ-expander decomposition of G for $s=\exp(\mathrm{poly}(1/\epsilon))$, where the size of the resulting moving cut is at most $n^{\epsilon}\cdot m\cdot\varphi$, in time $O(m\cdot n^{\mathrm{poly}(\epsilon)}\cdot\mathrm{poly}(h))$. Their algorithm was recently extended to the dynamic setting by [HLS24]. A key notion underlying these results is that of *routers*. As mentioned already, a graph G is a router if any unit demand can be routed in G via short paths

with low congestion. Equivalently, a router can be thought of as an (h, s)-length φ-expander, whose diameter is bounded by h. In order certify that a graph H is an (hs)-length φ-expander, [HHT24] use a variation of the notion of *neighborhood covers*. In this variation, a neighborhood cover is a collection $\{\mathcal{C}_1, \ldots, \mathcal{C}_r\}$ of clusterings, where, for all $1 \leq i \leq r$, clustering $\mathcal{C}_i$ is a collection of subgraphs of H called clusters, of radius at most h each. Additionally, every pair of clusters in $\mathcal{C}_i$ must be at distance at least hs from each other. Lastly, it is required that every vertex of H belongs to a relatively small number of clusters in $\bigcup_i \mathcal{C}_i$, and, for every vertex v of G, its h-neighborhood $B_H(v, h)$ must be contained in some cluster in $\bigcup_i \mathcal{C}_i$. As shown in [HHT24], in order to certify that a graph H is an (h, s)-length φ-expander, it is sufficient to compute, for every cluster $C \in \bigcup_i \mathcal{C}_i$, a router graph R_C with $V(R_C) = V(C)$, together with an embedding of all such resulting routers R_C into G via short paths, that cause low congestion. Their algorithm then computes a moving cut L for graph G, and certifies that the resulting graph $G' = G - L$ is an (h, s)-length φ-expander by computing the neighborhood cover $\{\mathcal{C}_1, \ldots, \mathcal{C}_r\}$ of G', and then embedding, for every cluster $C \in \bigcup_i \mathcal{C}_i$, the corresponding router R_C into G'. Therefore, the algorithms of [HHT24, HLS24] can also be thought of as computing and dynamically maintaining a weaker notion of router decomposition of the input graph G, similar to the weaker notion of expander decomposition of [ST04]. Specifically, every cluster C in the neighborhood cover is provided with a certificate router graph R_C, with $V(C) = V(R_C)$, and an embedding of R_C into G via short paths. However, this embedding may use edges and vertices that lie outside of C. Our main technical result is an algorithm for maintaining a *proper* router decomposition of a dynamic graph:

THEOREM 1.2. (DYNAMIC ROUTER DECOMPOSITION, INFORMAL) *There is a deterministic algorithm, whose input is an n-vertex undirected graph G, a stretch parameter $512 \leq k \leq (\log n)^{1/49}$, and two additional parameters $1/k \leq \delta \leq 1/400$, and $\Delta \geq n^{20/k}$. Graph G initially has no edges, and undergoes an online sequence of edge insertions and deletions. The algorithm maintains a collection $\mathcal{C}$ of edge-disjoint subgraphs (clusters) of G, and, for every cluster $C \in \mathcal{C}$, a subgraph $C' \subseteq C$ with $|V(C')| = |V(C)|$, such that the following hold at all times: (i) for every cluster $C \in \mathcal{C}$, any unit demand $\mathcal{D}_C$ for C can be routed inside C via paths of length at most $d \leq \text{poly}(k) \cdot 2^{O(1/\delta^6)}$, with congestion at most $\eta = n^{O(1/k)}$; (ii) $\sum_{C \in \mathcal{C}} |V(C)| \leq n^{1+O(1/k)}$; (iii) for every cluster $C \in \mathcal{C}$, any Δ-restricted demand on $V(C)$ can be routed in C' via paths of length at most d, with congestion at most η; (iv) $\sum_{C \in \mathcal{C}} |E(C')| \leq \Delta \cdot n^{1+O(1/k)}$; and (v) the total number of edges $e \in E(G)$ that do not lie in any cluster of $\mathcal{C}$ is at most $\Delta \cdot n^{1+O(1/k)}$. The worst-case update time of the algorithm is $n^{O(\delta)}$.*

In a sense, our decomposition result can be thought of as strengthening the results of [HHT24, HLS24] to a proper router decomposition, in the same manner as the result of [SW19] strengthened the fast algorithm of [ST04] for weak expander decomposition to obtain a proper expander decomposition. While this may appear as a technicality, just as with expander decompositions, a proper router decomposition provides a cleaner structure that is easier to use in applications. In fact some of the applications of our router decomposition described below crucially rely on the fact that the decomposition is proper. Additionally, for every cluster C in the decomposition, we maintain a sparsified subgraph $C' \subseteq C$ of a prescribed density that remains a sufficiently strong router. These subgraphs are useful for computing various sparsifiers of G. We note that, unlike the decompositions of [HHT24, HLS24], our decomposition does not provide an LC-expander decomposition of G. Specifically, if we denote by E^{del} the set of edges that do not lie in any cluster, then our decomposition does not guarantee that $G \setminus E^{\text{del}}$ is a length-constrained expander. This is due to the fact that the clusters in our decomposition do not necessarily define a neighborhood cover in $G \setminus E^{\text{del}}$.

We obtain the result of Theorem 1.1 from Theorem 1.2 as follows: we use the parameter $\Delta = n^{O(1/k)}$; the spanner H is obtained by taking the union of the subgraphs C' for all $C \in \mathcal{C}$, together with all edges of G that do not lie in the clusters of $\mathcal{C}$. We show that H can be maintained with worst-case update time $n^{O(\delta)}$ and recourse $n^{O(1/k)}$. Our results also extend to low-congestion and fault-tolerant spanners, that we discuss next.

Low-Congestion and Fault-Tolerant Spanners. The following general recipe, pioneered by [BvdBG+22], can be used in order to compute spanners via expander decomposition and routing. Initially, the spanner H contains

n vertices and no edges. Next, we compute an expander decomposition $\mathcal{C}$ of G, and, for every cluster $C \in \mathcal{C}$, we compute a sparse subgraph $C' \subseteq C$ that is an expander. For every cluster $C \in \mathcal{C}$ of the decomposition, we then add the edges of C' to H, and we consider the edges of C as "settled". The algorithm then continues recursively with edges that are not settled yet: namely, edges whose endpoints lie in different clusters of $\mathcal{C}$. It is easy to verify that the resulting spanner H has polylogarithmic stretch and that it is sparse. The recent breakthrough almost-linear algorithm of [CKL+22, vdBCK+23] for Min Cost Flow exploits the fact that the spanner H obtained via this approach has additional useful properties. Specifically, they show that one can embed all edges of G into H, such that all the embedding paths are short, and cause low vertex-congestion. This is in contrast to the standard notion of spanners, that only guarantees that, for each edge (u, v) of G, there is a short u-v path in H, though the resulting paths may cause arbitrarily high congestion.

In this work, we formalize this notion by introducing *low-congestion spanners*, and show an algorithm that dynamically maintains such a spanner, by exploiting our result for router decomposition. Formally, let G be a graph, and let H be a subgraph of G. We say that H is a (d, η)-low congestion spanner for G, if there is an embedding of the edges of G into H, such that all embedding paths have length at most d, and cause edge-congestion at most η. We use our router decomposition to show that for any stretch parameter $512 \leq k \leq (\log n)^{1/49}$ and degree threshold Δ, any n-vertex graph G admits a (d, η)-low congestion spanner with $\widetilde{O}(\Delta \cdot n^{1+O(1/k)})$ edges for $d = \text{poly}(k)$ and $\eta = O\left(n^{O(1/k)} \cdot \frac{\Delta_{\max}(G)}{\Delta}\right)$, where $\Delta_{\max}(G)$ is the maximum vertex degree in G. In fact we show that the graph H that is maintained by the algorithm from Theorem 1.1, and which, in turn, relies on the router decomposition from Theorem 1.2, has this property. We believe that low-congestion spanner is an interesting object that may find other applications in the future.

Fault-Tolerant Spanners. An additional useful property satisfied by expander-based spanners is their resilience to edge faults. As many applications of spanners arise in the context of distributed networks, which are inherently prone to failures of edges and vertices, it is especially desirable to obtain robust spanners, that maintain their functionality in the presence of such faults. *Fault-tolerant (FT) spanners* are specifically designed to provide such guarantees: given a graph G and parameters f and t, a subgraph $H \subseteq G$ is an f-FT t-spanner if, for every collection $F \subseteq E(G)$ of at most t failed edges, $H \setminus F$ is a t-spanner for $G \setminus F$. FT-spanners were first introduced in the context of geometric graphs by Levcopoulos et al. [LNS98], and were later considered for general graphs by Chechik et al. [CLPR10]. Recently, Parter [Par22] presented a near-linear time randomized algorithm for constructing an f-FT $(2k-1)$-spanner with $\widetilde{O}(f^{1-1/k} \cdot n^{1+1/k})$ edges, which is also resilient to up to f vertex faults. Bodwin, Dinitz, Robelle [BDR22] provided an existential result for f-FT $(2k-1)$-spanners: for odd values of k, they obtain spanners of size $O(k^2 f^{1/2-1/(2k)} n^{1+1/k} + kfn)$, and for even k, the size is $O(k^2 f^{1/2} n^{1+1/k} + kfn)$. These bounds almost match the $\Omega\left(f^{1/2-1/(2k)} \cdot n^{1+1/k}\right)$ size lower bound that is based on the Erdös Girth Conjecture [BDPW18].

Notice that, if we use the notion of f-FT spanners, that can only tolerate at most f edge failures, the spanner size must be at least fn, even when allowing a large stretch, since it is possible that all of the edge faults are incident to a single vertex. It is possible, however, that one can obtain sparse spanners that can withstand a significantly higher number of edge faults, as long as these faults are distributed evenly in the graph, and are not concentrated around a small number of vertices. With this motivation in mind, a very recent work of Bodwin, Haeupler and Parter [BHP24] introduced a stronger notion of fault-tolerant spanners, called *Faulty-Degree (FD) spanners*. Specifically, suppose we are given an n-vertex graph G, and parameters t and f. An f-FD t-spanner for G is a graph $H \subseteq G$, with the following property: if F is any subset of edges of G, such that every vertex of G is incident to at most f edges of F, then $H \setminus F$ is a t-spanner for $G \setminus F$. Notice that, for $f = 1$, the set F of edges can be any matching, whose size may be as high as linear in n. Bodwin et al. [BHP24] provided two approaches for computing sparse f-FD spanners. The first approach uses the LC-expander decomposition, and obtains f-FD spanners with stretch $k^{O(k)}$ and size $\widetilde{O}(fn^{1+1/k})$ via an algorithm with a high polynomial running time. Combining this approach with the recent time-efficient algorithm for LC-expander decomposition and routing of [HHT24, HLS24] yields an algorithm with running time $O(m \cdot n^{O(1/k)})$, albeit with a slightly higher stretch $2^{O(\text{poly}(k))}$. Their second approach, based on a greedy algorithm, provides an f-FD spanner with stretch $(2k-1)$ and size $\widetilde{O}\left(f^{1-1/k} \cdot n^{1+1/k} \cdot k^{O(k)}\right)$, which is nearly optimal for fixed values of k under the Erdös Girth Conjecture. The main drawback of the latter approach is the large polynomial running time of their algorithm. In their paper, [BHP24] noted that improving the $k^{O(k)}$ stretch bound obtained via the LC-expanders is closely

related to the question of the resilience of LC-expanders to bounded degree faults, which was left open therein (see Theorem 1.14 in [BHP24]). In this work, we build on our results for router decomposition to obtain f-FD spanners with stretch poly(k), that contain $\widetilde{O}(fn^{1+1/k})$ edges. This allows us to address the open problem of [BHP24], only on router graphs (which can viewed as LC-expanders with small diameter), and improve the stretch bound from $k^{O(k)}$ to poly(k). The running time of our algorithm is $n^{1+O(1/k)+o(1)}$ which outperforms the (at least quadratic) runtime of the greedy-based approach of [BHP24]. Moreover, our algorithm maintains such a spanner for a fully dynamic graph G with low recourse and low worst-case update time that are independent on the number of faults f. Our construction of f-FD spanners crucially relies on the fact that our router decomposition is proper.

Connectivity Certificates. Finally, a graph structure closely related to FT-spanners is the f-edge connectivity certificates, introduced by [NI92]. For an integer $f \geq 1$, we say that a graph G is f-edge connected, if, for any subset F of $f-1$ edges of G, graph $G \setminus F$ is connected. An f-edge connectivity certificate for G is a subgraph $H \subseteq G$, that has the following property: H is f-edge connected if and only if G is. Connectivity certificates play a key role in fast algorithms for minimum cuts; see, e.g. [Mat93, CKT93, KM97, GK13, DHNS19, FNY$^+$20, LNP$^+$21, GHN$^+$23]. Nagamochi and Ibaraki [NI92] presented a linear-time static algorithm for computing f-edge connectivity certificates of n-vertex graphs G, with $O(fn)$ edges. In the dynamic setting, Thorup and Karger [TK00] presented a deterministic algorithm for maintaining an f-edge connectivity certificate of a dynamic n-vertex graph, of size $O(fn)$, with $\widetilde{O}(f^2)$ amortized update time. Combining the latter with the deterministic algorithm of [CGL$^+$20] for dynamic minimum spanning forest yields $O(f^2 \cdot n^{o(1)})$ worst-case update time. The dependency of the update time on f can be avoided using randomness: the recent work of Assadi and Shah [AS23] implies a *randomized* algorithm for maintaining an f-connectivity certificate with $O(f \cdot n \log n)$ edges in a fully dynamic n-vertex graph, with $O(\text{poly} \log n)$ worst-case update time. To the best of our knowledge, there is no deterministic dynamic algorithm for maintaining sparse f-connectivity certificate with worst-case update time bound that is independent of f. We show that our algorithm for maintaining a router decomposition can also be used to maintain an f-connectivity certificate with $f \cdot n^{1+o(1)}$ edges (and therefore of almost optimal size) for a fully dynamic graph, with worst-case update time and recourse bounded by $n^{o(1)}$. In fact, we maintain an even stronger version of an f-connectivity certificate recently introduced in [BHP24], called f-FD connectivity certificate, that can withstand the failure of any edge subset F, as long as every vertex of G is incident to at most f edges of F.

1.1 Our Results and Techniques

Our main technical result is an algorithm that maintains a low-diameter router decomposition of a fully dynamic graph with low worst-case update time. Before we state the result, we need to define the notions of demands and routers; formal definitions can be found in Sections 2.2 and 2.3. Consider a graph $G = (V, E)$. A *weighting* of the vertices of G is an assignment $w(v) \geq 0$ of a non-negative weight to every vertex $v \in V$. We also denote such a weighting by an n-dimensional vector $\overline{w} \in \mathbb{R}^n$. We sometimes use two special vertex weightings: in the first weighting, denoted by $\overline{\deg_G}$, the weight of every vertex $v \in V$ is its degree in G. In the second weighting, we are given some value $\Delta > 0$, and the weight of every vertex is set to be Δ; we denote such a weighting by $\overline{\Delta}$. A *demand* $\mathcal{D}$ in G consists of a collection Π of pairs of vertices, and, for every pair $(a, b) \in \Pi$, a value $D(a, b) > 0$, that we refer to as the *demand between a and b*. For a vertex weighting $\overline{w}$, we say that a demand $\mathcal{D} = \left(\Pi, \{D(a,b)\}_{(a,b)\in\Pi}\right)$ is *$\overline{w}$-restricted*, if, for every vertex $v \in V$, the total demand between all pairs in Π that contain v is at most $w(v)$. A *routing* of the demand $\mathcal{D}$ is a flow f, in which, for every pair $(a, b) \in \Pi$, $D(a, b)$ flow units are sent from a to b. The *congestion* of the routing is the maximum amount of flow through any edge of G. We say that the routing is via *flow-paths of length at most d*, if every flow-path with non-zero flow value contains at most d edges.

A central notion that we use in our paper is that of a *router*, that was first defined in [HHT24] as an object that is closely related to length-constrained expanders. Given a graph G with vertex weighting $\overline{w}$, and parameters η and d, we say that G is a $(\overline{w}, d, \eta)$-router[1], if every $\overline{w}$-restricted demand $\mathcal{D}$ can be routed with congestion at most

[1] in Section 2.3 we provide a slightly more general definition of routers, that includes a set S of supported vertices, where the router is only required to route demands defined over the vertices of S. But all our results hold for the case where $S = V(G)$, so the definition above is sufficient in order to state our results.

η in G, via paths of length at most d.

Our main technical result, summarized in the following theorem, is an algorithm that maintains a router decomposition of a fully dynamic graph, with small worst-case update time. The algorithm also maintains a subgraph H of G of a prescribed density that has additional useful properties. In particular, as we show later, graph H is a (low-congestion and fault-tolerant) spanner of G.

THEOREM 1.3. *There is an algorithm, whose input is a graph G with n vertices that initially has no edges, which undergoes an online sequence of at most n^2 edge deletions and insertions, such that G remains a simple graph throughout the update sequence. Additionally, the algorithm is given parameters $512 \leq k \leq (\log n)^{1/49}$, $\frac{1}{k} \leq \delta < \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer, and $\Delta \geq n^{20/k}$. The algorithm maintains the following:*

- *a collection $\mathcal{C}$ of edge-disjoint subgraphs of G called* clusters *with $\sum_{C \in \mathcal{C}} |V(C)| \leq n^{1+O(1/k)}$, such that every cluster $C \in \mathcal{C}$ is a $(\overline{\deg}_C, d, \eta)$-router for $d = k^{19} \cdot 2^{O(1/\delta^6)}$ and $\eta = n^{O(1/k)}$. Moreover, if we denote by $E^{\mathrm{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$, then, at all times, $|E^{\mathrm{del}}| \leq n^{1+O(1/k)} \cdot \Delta$ holds;*
- *for every cluster $C \in \mathcal{C}$, a subgraph $C' \subseteq C$ with $V(C') = V(C)$, that is a $(\overline{\Delta}, d, \eta)$-router, with $\sum_{C \in \mathcal{C}} |E(C')| \leq n^{1+O(1/k)} \cdot \Delta$; and*
- *a graph $H \subseteq G$ with $|E(H)| \leq n^{1+O(1/k)} \cdot \Delta$, that contains all edges of $E^{\mathrm{del}} \cup \left(\bigcup_{C \in \mathcal{C}} E(C')\right)$.*

The worst-case update time of the algorithm is $n^{O(\delta)}$ per operation, and the total number of edge insertions and deletions that graph H undergoes after each update to G is at most $n^{O(1/k)}$.

We note that, if we set $\Delta = n^{20/k}$, then it is guaranteed that $|E(H)| \leq n^{1+O(1/k)}$ holds, and moreover, it is easy to verify that H is a d-spanner for G. In Section 9 we show that, in addition to being a d-spanner, H has many other useful properties; for example, it is also a low-congestion spanner. Moreover, given a fault parameter f, by setting the bound Δ appropriately, we can guarantee that H is an f-FD spanner. Lastly, graph H can also be used as an f-connectivity certificate for G. Interestingly, the worst-case update time to maintain these structures is independent of the parameter f.

We now provide an overview of the techniques that we employ in the proof of Theorem 1.3. Using a standard deamortization technique (see [NSWN17, JS22], and Section 2.6), it is sufficient to provide an algorithm with guarantees similar to those in Theorem 1.3 in the *online-batch dynamic model*. In this model, the initial graph G may contain an arbitrary number of edges, and it undergoes k batches $\pi_1, \ldots, \pi_k$ of updates, where, for all $1 \leq i \leq k$, the ith batch π_i of updates is a collection of edge insertions and deletions for graph G. In order to obtain the desired worst-case update time and recourse bound from Theorem 1.3, it is enough to design an algorithm in the online-batch model, whose initialization time is $m^{1+O(\delta)}$; the time required to process each batch π_i is bounded by $|\pi_i| \cdot n^{O(\delta)}$; and the total number of edge insertions and deletions that graph H undergoes following each batch π_i of updates is bounded by $|\pi_i| \cdot n^{O(1/k)}$. We develop several technical tools in order to solve this problem, that we describe next.

First tool: nice router $W_k^{N,\Delta}$ and its pruning. Our first tool is an explicit construction of a nice well-structured router graph, and an algorithm for its pruning. Suppose we are given parameters N, Δ and k. The corresponding graph $W_k^{N,\Delta}$ (that, for simplicity, we will denote by W_k), consists of N^k vertices, that are partitioned into a set V^c of N^{k-1} *center* vertices, and a set V^ℓ of remaining vertices, called *leaf* vertices. The set of edges of W_k is partitioned into k subsets $E_1, \ldots, E_k$, where, for $1 \leq i \leq k$, we refer to the edges of E_i as *level-i edges*. For all $1 \leq i \leq k$, the subgraph of W_k induced by the edges of E_i can be thought of as consisting of a union of N^{k-1} disjoint star graphs (called *level-i stars*), each of which has $N-1$ leaves, after we replace every edge of every star by a collection of Δ parallel edges (that we refer to as a *bundle*; we also refer to the corresponding star edge as a *superedge*). The center vertices of all stars must lie in V^c, and their leaves must lie in V^ℓ.

The construction of the graph W_k is inductive. In order to obtain the graph W_1, we start with a star S that has $N-1$ leaves. We then replace every edge of S with Δ parallel edges, obtaining the graph W_1. The center

of the star S becomes the unique center vertex of W_1. The edges of W_1 are *level-1 edges*. Assume now that, for an integer $1 < i \leq k$, we have defined the graph W_{i-1}. In order to obtain a graph W_i, we start by constructing N disjoint copies of $C_1, \ldots, C_N$ of the graph W_{i-1}; we refer to these copies as *level-*$(i-1)$ *clusters*. The set of the center vertices of W_i is the union of the sets of center vertices of the graphs $C_1, \ldots, C_N$, and the remaining vertices of W_i are leaf vertices. We connect the clusters $C_1, \ldots, C_N$ to each other by level-i edges, as follows. We construct N^{k-1} disjoint stars $S_1, \ldots, S_{N^{k-1}}$ (level-i stars), so that each star S_j has $N-1$ leaves; each vertex of S_j lies in a distinct cluster of $\{C_1, \ldots, C_N\}$; and the center of the star S_j is a center vertex of W_k. For each such star S_j, we then replace each of its edges with Δ parallel edges, which are referred to as level-i edges.

Next, we define the notion of a *properly pruned subgraph* of W_k. Suppose we are given a subgraph $W \subseteq W_k$, and k collections $U_1, \ldots, U_k$ of vertices of W, with $U_k \subseteq U_{k-1} \subseteq \cdots \subseteq U_1$. We say that W is a *properly pruned subgraph* of W_k with respect to the sets $U_1, \ldots, U_k$ of its vertices, if $V(W) = U_1$, and the graph obeys the following additional rules. First, for each level $1 \leq i \leq k$, for each level-i superedge (u, v), where v is a leaf vertex, if v remains in set U_i, then at least $\Delta/2$ parallel edges (u, v) must remain in W. Second, for each level $1 \leq i \leq k$, and each level-i star S, if the center vertex v of S remains in U_i, then a large fraction of the leaves of S must remain in U_i. Lastly, for all $1 \leq i < k$, for every level-i cluster C, if any vertex of C remains in U_1, then a significant fraction of vertices of C must lie in U_{i+1}. We prove that, if W is a properly pruned subgraph of W_k, then it is a strong router: namely, any Δ-restricted demand can be routed in W via paths of length $O(k^2)$, with congestion $k^{O(k)}$. The routing algorithm uses the star structure of the graph W_k in a natural manner.

Finally, we provide an algorithm for pruning the graph W_k in the online-batch dynamic model. The algorithm is given k batches $\pi_1, \ldots, \pi_k$ of edge deletions from W_k, and maintains a properly pruned decremental subgraph W of W_k throughout the update sequence. The algorithm ensures that the processing time of each batch π_i is bounded by $O(|\pi_i| \cdot k^{O(k)})$, and the number of edges and vertices deleted from W after each batch π_i is small compared to $|\pi_i|$. The pruning algorithm is quite straightforward: whenever any of the above rules are not obeyed, we perform vertex deletions in order to ensure that the current graph W is compliant with the rules.

We note that [HHT24] used the kth power graph of a constant-degree expander as an analogue of a nice router graph. They also provide a pruning algorithm for such routers (see Lemma 5.29 of [HHT24]). To the best of our understanding, their algorithm relies on the expander pruning algorithm of [SW19], by maintaining the kth power graph of the resulting pruned expander. Our algorithm uses a completely different strategy, where we construct the initial router explicitly, define rules for properly pruned subgraphs that guarantee that the resulting graph remains a strong enough router, and design our pruning algorithm to enforce these rules in a rather straightforward manner. One advantage of our construction is that it is easy to maintain a sparsified subgraph W' of $W_k^{N,\Delta}$ of any prescribed density $\Delta' < \Delta$ that remains a strong router, even as the graph undergoes edge deletions and pruning: for every superedge (u, v), we simply include in W' arbitrary Δ' parallel copies of (u, v). This, in turn, allows us to maintain the sparsified routers $C' \subseteq C$ for the clusters C in the router decomposition $\mathcal{C}$ that our algorithm maintains. We believe that this construction and the pruning algorithm are of independent interest.

Second tool: embedding a nice router. Our second tool is an algorithm that can be used in order to embed a nice router W_k into a given graph. Specifically, suppose we are given a graph C, and parameters k and $d = k^{O(1)}$, such that, for some vertex $v \in V(C)$, the number of vertices of C that lie within distance d from v is significantly larger than $|V(C)|^{1-1/k}$. Assume further that $|E(C)| \approx |V(C)| \cdot \Delta \cdot n^{\Theta(1/k)}$. Our algorithm either successfully embeds a graph $W_k^{N,\Delta}$ into C via short paths that cause small congestion, where N^k is quite close to $|V(C)|$; or computes a subset E' of $O(n\Delta)$ edges, so that $C \setminus E'$ becomes *scattered*: that is, for every vertex $u \in V(C)$, the ball of radius d around u in $C \setminus E'$ contains fewer than $|V(C)|^{1-1/k}$ vertices. More specifically, our algorithm uses an additional parameter δ to control the tradeoff between the lengths of the embedding paths and the running time. In the first case, the algorithm computes a relatively small collection F of edges of $W_k^{N,\Delta}$, and an embedding $\mathcal{P}$ of $W_k^{N,\Delta} \setminus F$ into C via paths of length $\mathrm{poly}(k) \cdot 2^{O(1/\delta^6)}$, with congestion at most $n^{O(1/k)}$. The running time of the algorithm is $m^{1+O(\delta)}$. If the algorithm terminates with an embedding of $W_k^{N,\Delta} \setminus F$ into C, then we can employ our pruning algorithm, in order to compute a large enough properly pruned subgraph W of $W_k^{N,\Delta}$ that does not contain any edges of F, together with its embedding $\mathcal{P}'$ via paths whose length and congestion are bounded as before. Moreover, we can compute a large enough subgraph $\hat{C} \subseteq C$ that contains all embedding paths in $\mathcal{P}'$, such that every vertex of $\hat{C}$ participates in a large number of such paths. As we show later, this is sufficient in order to prove that $\hat{C}$ is a $(\overline{\deg}_{\hat{C}}, \tilde{d}, \tilde{\eta})$-router.

The algorithm for computing the embedding of $W_k^{N,\Delta} \setminus F$ (or a subset E' of edges whose removal makes C scattered), employs the algorithmic techniques that were developed in [Chu23] for *well-connected graphs*. Specifically, we start by selecting a large subset T of vertices of C that are close to each other, and then use an algorithm from [Chu23] in order to attempt to embed a well-connected graph, whose vertex set is a large subset of T, into C, via short paths that cause a low congestion. If the algorithm fails to do so, then it computes a small collection E'' of edges, so that, in $C \setminus E''$, many pairs of vertices of T become far from each other. In the latter case, we either compute a new large subset T of vertices of C that remain close to each other and restart the same algorithm; or correctly establish that the graph becomes scattered and terminate the algorithm. Using simple accounting, we show that the number of such iterations in the algorithm is not too large. If we successfully embed a large well-connected graph R into C, we employ the algorithm of [Chu23] for APSP in well-connected graphs, together with the standard Even-Shiloach trees, in an attempt to embed the edges of $W_k^{N,\Delta}$ into C one by one. If we fail to embed a large fraction of the edges of $W_k^{N,\Delta}$, then we again obtain a small subset E'' of edges, such that, in $C \setminus E''$, many pairs of vertices that were originally close to each other, become far from each other. All such sets E'' of edges removed from C over the course of the algorithm are added to a set E', and, since the number of iterations in the algorithm is relatively small, we can ensure that $|E'|$ remains sufficiently small as well.

Third tool: router witness. Suppose we are given a graph W, that is a properly pruned subgraph of a graph $W_k^{N,\Delta}$. Assume further that we are given some graph C, and an embedding of W into C via paths of length d^* that cause congestion at most η^*. Assume further that all vertex degrees in C are at most $\alpha \cdot \Delta$, and every vertex $v \in V(C)$ lies on at least Δ/β of the embedding paths. We prove that, in such a case, C must be a $(\overline{\deg}_C, \tilde{d}, \tilde{\eta})$-router, for $\tilde{d} = d^* \cdot \mathrm{poly}(k)$, and $\tilde{\eta} = \alpha \cdot \beta \cdot d^* \cdot \eta^* \cdot k^{O(k)}$. Therefore, we can view the graph W and its embedding $\mathcal{P}$ into C with the above properties as a *witness* that C is a $(\overline{\deg}_C, \tilde{d}, \tilde{\eta})$-router. In our construction of the router decomposition, we employ this type of witnesses to ensure that all clusters in the decomposition remain strong routers.

Assume now that we are given a target parameter $\Delta^* \ll \Delta$, a cluster C, and a router witness $(W, \mathcal{P})$ as described above. Assume further that our goal is to compute a *sparsified router* $C' \subseteq C$ for C, so that $|E(C')| \leq \Delta^* \cdot |V(C)|$, and C' is a $(\Delta^*, \tilde{d}, \tilde{\eta}')$-router, where $\tilde{d}$ remains the same as before, and $\tilde{\eta}'$ is only slightly higher than $\tilde{\eta}$. We show that such a cluster C' can be constructed as follows. We let $\Delta' < \Delta^*$ be a parameter that is sufficiently close to Δ^*. First, for every superedge (a, b) of W, we select a subset $E'(a, b)$ of Δ' parallel edges (a, b), and we let $\mathcal{Q}' \subseteq \mathcal{P}$ be the set of all embedding paths of all edges in sets $E'(a, b)$, for all superedges (a, b) of W. Next, we construct a collection $\mathcal{Q}'' \subseteq \mathcal{P}$ of paths as follows: every vertex $v \in V(C)$ selects a subset of Δ' paths of $\mathcal{P}$ that contain v, and adds them to $\mathcal{Q}''$; but we require that every leaf vertex of W serves as an endpoint of at most $\gamma \cdot \Delta'$ such paths, for $\gamma = n^{O(1/k)}$. We then let $C' \subseteq C$ be the graph that contains all edges and vertices participating in the paths of $\mathcal{Q}' \cup \mathcal{Q}''$. We prove that such a graph C' must be a $(\Delta^*, \tilde{d}, \tilde{\eta}')$-router, by using the routing properties of properly pruned subgraphs of $W_k^{N,\Delta}$ in a straightforward manner. We also show that $|E(C')| \leq |V(C')| \cdot \Delta^*$.

Fourth tool: decremental low-diameter clustering. Our last tool is a simple algorithm for decremental low-diameter clustering. Suppose we are given a graph G and a parameter $k < \log n$. We say that a subgraph $C \subseteq G$ is a *settled cluster*, if either $|V(C)| \leq n^{1/k}$, or there is a vertex $v \in V(C)$, such that the ball of radius $\mathrm{poly}(k)$ around v contains at least $|V(C)|^{1-1/k}$ vertices. If a subgraph $C \subseteq G$ is not a settled cluster, then we call it *unsettled*.

In the decremental low-diameter clustering problem, the goal is to maintain a decomposition $\mathcal{C}$ of G into edge-disjoint clusters under edge deletions, so that each cluster in $\mathcal{C}$ is settled, and $\sum_{C \in \mathcal{C}} |V(C)| \leq n^{1+O(1/k)}$ holds (for conciseness, we refer to the latter property as a *low overlap*). We note that the settled clusters in the decomposition that we maintain are not guaranteed to have a low diameter; each such cluster either has relatively few vertices, or it has a single vertex with a sufficiently large neighborhood.

Specifically, the algorithm is required to first compute an initial decomposition $\mathcal{C}$ of G into settled clusters with low overlap. The set $\mathcal{C}$ of clusters is then partitioned by the adversary into a set $\mathcal{C}^I$ of *inactive* and a set $\mathcal{C}^A$ of *active* clusters. At the beginning, all clusters are active. Once a cluster becomes inactive, it may not undergo any further updates. Then the algorithm proceeds in iterations. In every iteration, the adversary may (i) move some clusters from $\mathcal{C}^A$ to $\mathcal{C}^I$; and (ii) delete some edges and vertices from clusters $C \in \mathcal{C}^A$, after which each such

cluster must become unsettled. After that the algorithm is required to "fix" the clusters in $\mathcal{C}^A$ via *cluster-splitting* updates: given a cluster $C \in \mathcal{C}$ and a subgraph $C' \subseteq C$, add C' to $\mathcal{C}$ and to the set $\mathcal{C}^A$ of active clusters, delete all edges of C' from C, and delete any resulting isolated vertices from C. The algorithm must perform such cluster splitting operations, until all clusters in $\mathcal{C}^A$ become settled. We provide a simple algorithm for the decremental low-diameter clustering problem that is based on the standard ball-growing technique.

Combining all tools together. We are now ready to describe the initialization algorithm for Theorem 1.3. Assume first that all vertex degrees in the input graph G are close to $\Delta \cdot n^{\Theta(1/k)}$, for some $\Delta \gg \Delta^*$. We use the algorithm for the low-diameter clustering problem, to compute an initial collection $\mathcal{C}$ of edge-disjoint clusters with low overlap, so that all clusters in $\mathcal{C}$ are settled. We set $\mathcal{C}^I = \emptyset$, $\mathcal{C}^A = \mathcal{C}$, and then iterate, as long as $\mathcal{C}^A \neq \emptyset$. In every iteration, we consider every cluster $C \in \mathcal{C}^A$ one by one. For each such cluster C, we attempt to embed a graph $W_k^{N_C,\Delta} \setminus F$ into C, where F is a relatively small set of fake edges, and N_C is chosen so that $\frac{|V(C)|}{n^{\Theta(1/k)}} \leq N_C^k \leq |V(C)|$, via short paths that cause low congestiton. If we fail to do so, then we obtain a relatively small set E_C of edges, such that graph $C \setminus E_C$ is scattered. We delete the edges of E_C from C (and add them to the set E^{del} of deleted edges that we maintain), and notify the algorithm for the low-diameter clustering problem regarding these edge deletions; we show that C now becomes unsettled. If, however, we manage to embed a graph $W_k^{N_C,\Delta} \setminus F$ into C as above, then we move C to the set $\mathcal{C}^I$ of inactive clusters. Using the pruning algorithm, we then compute a large subgraph $\hat{C} \subseteq C$, together with a router witness for $\hat{C}$, so that $\hat{C}$ is $(\overline{\deg}_{\hat{C}}, \tilde{d}, \tilde{\eta})$-router. We also construct a subgraph $\hat{C}' \subseteq \hat{C}$, with $|E(\hat{C}')| \leq \Delta^* \cdot |V(\hat{C})|$, such that $\hat{C}'$ is a $(\overline{\Delta}^*, \tilde{d}, \tilde{\eta})$-router. We then add $\hat{C}$ to the router decomposition that we are constructing.

Once the algorithm terminates, $\mathcal{C}^A = \emptyset$ holds, and so all clusters are inactive. For each such inactive cluster C, we have constructed a large enough router $\hat{C} \subseteq C$. If the number of edges lying outside of the routers that we have constructed so far is too high, then we repeat the algorithm with these remaining edges. However, since our algorithm ensures that a large enough fraction of edges of each cluster C is included in the corresponding router $\hat{C}$, we can show that the number of times that we need to restart the algorithm is relatively small.

Our algorithm for processing each batch π_i of updates to graph G proceeds as follows. Assume first that all updates in π_i are edge deletions. Then we simply use the pruning algorithm for the nice routers that we described above, inside each cluster C of the decomposition, in order to maintain the corresponding router witness.

Lastly, so far we have assumed that all vertex degrees in the graph G are close to each other, and that the update batches π_i only contain edge deletions. In order to handle a general graph G, we partition its edges among at most k subgraphs $G_1, G_2, \ldots$, where, for each j, $|E(G_j)| \leq \Delta^* \cdot n^{1+j/k}$, and all vertex degrees in G_j are at least $\Delta^* \cdot n^{(j-2)/k}$. For each such subgraph G_j, we further perform *splitting* of its high-degree vertices, to ensure that all vertex degrees become close to $\Delta^* \cdot n^{(j-2)/k}$, and the number of vertices in the resulting graph G'_j remains at most $2n$. For all $j > 2$, we initialize the above algorithm on graph G'_j. In fact, the initialization proceeds from higher to lower indices j, and edges that are left over from the initialization algorithm for graph G'_j (that is, edges that do not lie in any of the routers), become the edges of graph G_{j-1}. The set E^{del} of edges is then set to contain all edges of $E(G_1) \cup E(G_2)$. Consider now a batch π_i of updates. We compute the smallest integer j, so that $\Delta^* \cdot n^{1+(j-2)/k} \geq |\pi_i|$. For indices $j' > j$, for each edge e of $G'_{j'}$ that is deleted in π_i, we delete e from $G'_{j'}$, and update the router decomposition of $G'_{j'}$ following these edge deletions; this may result in an additional collection $E_{j'}$ of edges to be deleted from $G_{j'}$, where $|E_{j'}|$ is roughly bounded by the number of edges of $G_{j'}$ that are deleted by π_i. Next, we construct a set E^* of edges, that contains (i) all edge insertions from π_i; (ii) all edges lying in graphs $G_1, \ldots, G_j$; and (iii) all sets $E_{j'}$ of edges that were deleted from graphs $G_{j'}$ for $j' > j$ while the batch π_i was processed. We let the new graph G_j contain all edges of E^*, and we initialize the algorithm for computing the router decomposition on the corresponding vertex-split graph G'_j from scratch. As before, all leftover edges that are not included in the resulting router decomposition are added to the graph G_{j-1}, and we recursively initialize the algorithm for computing the router decomposition on the corresponding graph G'_{j-1}, and so on.

Organization. We start with Preliminaries in Section 2. We formally define router decompositions, and state our main technical result: namely, an algorithm that maintains a router decomposition of a graph that undergoes a small number of batches of updates, in Section 3. We also show that the proof of Theorem 1.3 follow from this main technical result. Then in sections Sections 4–7 we develop our main technical tools for maintaining a router

decomposition. The first tool is a simple construction of a nice router, and an algorithm for its pruning, that are presented in Section 4. The second tool is an algorithm for embedding a nice router into a given graph, and is presented in Section 5. The third tool is a router witness, that is presented in Section 6. Finally, the last tool is a decremental algorithm for low-diameter decomposition, that appears in Section 7. In Section 8, we combine all these tools to complete the proof of the main technical theorem. Lastly, in Section 9, we provide an algorithm for maintaining (low congestion, fault tolerant) spanners for bounded-degree faults in a fully dynamic graph, and connectivity certificates. We defer some of the proofs to the full version of the paper.

2 Preliminaries

All logarithms in this paper are to the base of 2. Throughout the paper, we use the $\tilde{O}(\cdot)$ notation to hide multiplicative factors that are polynomial in $\log m$ and $\log n$, where m and n are the number of edges and vertices, respectively, in the initial input graph. All graphs in this paper are undirected. By default, we allow parallel edges but we do not allow self loops. Graphs in which parallel edges are not allowed are explicitly referred to as *simple* graphs.

We follow standard graph-theoretic notation. Given a graph $G = (V, E)$ and two disjoint subsets A, B of its vertices, we denote by $E_G(A, B)$ the set of all edges with one endpoint in A and another in B, and by $E_G(A)$ the set of all edges with both endpoints in A. We also denote by $\delta_G(A)$ the set of all edges with exactly one endpoint in A. For a vertex $v \in V(G)$, we denote by $\delta_G(v)$ the set of all edges incident to v in G, and by $\deg_G(v)$ the degree of v in G. We also denote by $\overline{\deg}_G$ the n-dimensional vector of all vertex degrees in G. We may omit the subscript G when clear from context. Given a subset $S \subseteq V$ of vertices of G, we denote by $G[S]$ the subgraph of G induced by S.

If G is a graph, and $\mathcal{P}$ is a collection of paths in G, we say that the paths in $\mathcal{P}$ cause *congestion* η, if every edge $e \in E(G)$ participates in at most η paths in $\mathcal{P}$, and some edge participates in exactly η such paths.

Assume now that we are given a graph G with capacities $c(e) > 0$ on edges $e \in E(G)$. A *flow* in G is an assignment of non-negative flow values $f(P)$ to paths P in G. If P is a path whose corresponding flow value $f(P)$ is non-zero, then we say that P is a *flow-path*. A flow f is typically defined by providing a list of flow-paths P together with their corresponding flow value $f(P)$. For an edge $e \in E(G)$, the flow $f(e)$ through e is the total sum of flow values $f(P)$ over all paths P containing e. We say that flow f causes *congestion* η if $\max_{e \in E(G)} \left\{ \frac{f(e)}{c(e)} \right\} = \eta$. Notice that it is possible that $\eta < 1$. If $f(e) \leq c(e)$ holds for every edge $e \in E(G)$, we may sometimes say that f is a *valid flow*, or that it *respects edge capacities*. When the edge capacities $c(e)$ are not explicitly given, we assume that they are unit.

Distances and Balls. Suppose we are given a graph G with lengths $\ell(e) > 0$ on its edges $e \in E(G)$. For a path P in G, we denote its length by $\ell_G(P) = \sum_{e \in E(P)} \ell(e)$. For a pair of vertices $u, v \in V(G)$, we denote by $\mathsf{dist}_G(u, v)$ the *distance* between u and v in G: the smallest length $\ell_G(P)$ of any path P connecting u to v in G. For a pair S, T of subsets of vertices of G, we define the distance between S and T to be $\mathsf{dist}_G(S, T) = \min_{s \in S, t \in T} \{\mathsf{dist}_G(s, t)\}$. For a vertex $v \in V(G)$, and a subset $S \subseteq V(G)$ of vertices, we also define the distance between v and S as $\mathsf{dist}_G(v, S) = \min_{u \in S} \{\mathsf{dist}_G(v, u)\}$. The *diameter* of the graph G, denoted by $\mathsf{diam}(G)$, is the maximum distance between any pair of vertices in G.

Consider now some vertex $v \in V(G)$, and a distance parameter $D \geq 0$. The *ball of radius D around v* is defined as: $B_G(v, D) = \{u \in V(G) \mid \mathsf{dist}_G(u, v) \leq D\}$. Similarly, for a subset $S \subseteq V(G)$ of vertices, we let the ball of radius D around S be $B_G(S, D) = \{u \in V(G) \mid \mathsf{dist}_G(u, S) \leq D\}$. We will sometimes omit the subscript G when clear from context.

Embeddings of Graphs. Let G, X be two graphs with $|V(X)| \leq |V(G)|$. An *embedding* of X into G consists of a mapping $g : V(X) \to V(G)$, such that the vertices of X are mapped to **distinct** vertices of G, and a collection $\mathcal{P} = \{P(e) \mid e \in E(X)\}$ of paths in graph G, such that, for every edge $e = (x, y) \in E(X)$, path $P(e)$ connects $g(x)$ to $g(y)$. The *congestion* of the embedding is the maximum, over all edges $e' \in E(G)$, of the number of paths in $\mathcal{P}$ containing e'. Given an embedding of X into G as above, we will often identify vertices $v \in V(X)$ with the corresponding vertices $g(v) \in V(G)$ to which they are mapped, and so the embedding will only be specified

by the collection $\mathcal{P}$ of embedding paths. Whenever we say that we are given an embedding of a graph X into a graph G, this implicitly means that $|V(X)| \leq |V(G)|$.

2.1 Vertex Weighting, Conductance and Expanders Given an n-vertex graph $G = (V, E)$, a *vertex weighting* $w : V \mapsto \mathbb{Z}_{\geq 0}$ is an assignment of non-negative weight $w(v)$ to every vertex $v \in V(G)$. We denote by $\overline{w}$ the corresponding n-dimensional vector of all vertex weights. We may sometimes consider two special vertex weightings: weighting $\overline{\deg_G}$, where the weight of every vertex v is $\deg_G(v)$; and a weighting $\overline{\Delta}$, where $\Delta > 0$ is an real number, and $\overline{\Delta}$ is an n-dimensional vector with all entries equal to Δ, so the weight of every vertex is Δ.

Consider now a graph $G = (V, E)$ with a vertex weighting $\overline{w}$. For a subset $S \subseteq V$ of vertices, the weight of S is $w(S) = \sum_{v \in S} w(v)$. When $\overline{w} = \overline{\deg_G}$, we may denote $w(S)$ by $\mathrm{Vol}_G(S)$, and refer to it as the *volume of S in G*. If $S, V \setminus S \neq \emptyset$, we sometimes refer to $(S, V \setminus S)$ as a *cut*. Given a cut $(S, V \setminus S)$ with $w(S), w(V \setminus S) > 0$, the *conductance* of the cut with respect to vertex weighting $\overline{w}$ is:

$$\Phi_{G,\overline{w}}(S) = \frac{|\delta_G(S)|}{\min\{w(S), w(V \setminus S)\}} .$$

For simplicity, if $|\delta_G(S)| = 0$, we define $\Phi_{G,\overline{w}}(S) = 0$, and otherwise, if $|\delta_G(S)| > 0$ and $\min\{w(S), w(V \setminus S)\} = 0$, then we set $\Phi_{G,\overline{w}}(S) = \infty$.

The *conductance* of a graph G with respect to vertex weighting $\overline{w}$ is denoted by $\Phi(G, \overline{w})$, and it is defined to be the minimum conductance $\Phi_{G,\overline{w}}(S)$ of any cut $(S, V \setminus S)$.

If no vertex-weighting is explicitly given for a graph $G = (V, E)$, we use the default weighting $\overline{w} = \overline{\deg_G}$. In such a case, the conductance of a cut $(S, V \setminus S)$ is defined with respect to the standard volume $\mathrm{Vol}_G(\cdot)$ notion defined above.

Definition 2.1. (φ-expander) *Let $G = (V, E)$ be a graph, let $\overline{w}$ be a weighting of its vertices, and let $\varphi \geq 0$ be a parameter. We say that G is a φ-expander with respect to $\overline{w}$, if $\Phi(G, \overline{w}) \geq \varphi$. If G is φ-expander with respect to vertex weighting $\overline{w} = \overline{\deg_G}$, then we may simply say that G is a φ-expander.*

2.2 Demands and their Routing

Definition 2.2. ($\overline{w}$-restricted demand) *Let $G = (V, E)$ be a graph, and let $\overline{w}$ be a weighting of its vertices. A* demand *in G consists of a collection Π of pairs of vertices, and, for every pair $(a, b) \in \Pi$, a value $D(a, b) > 0$, that we refer to as the* demand between a and b. *We say that a demand $\mathcal{D} = \left(\Pi, \{D(a,b)\}_{(a,b)\in\Pi}\right)$ is* $\overline{w}$-restricted, *if, for every vertex $v \in V$, the total demand between all pairs in Π that contain v is at most $w(v)$. When $\overline{w} = \overline{\Delta}$ for an integer Δ, we may sometimes refer to a $\overline{w}$-resticted demand as a Δ-restricted demand.*

The support *of the demand $\mathcal{D} = \left(\Pi, \{D(a,b)\}_{(a,b)\in\Pi}\right)$ is the set of all vertices participating in the pairs in Π. We say that the demand is between two pairs $A, B \subseteq V$ of subsets of vertices of G, if $\Pi \subseteq A \times B$.*

Definition 2.3. (Routing of a demand) *Let $G = (V, E)$ be a graph, and let $\mathcal{D} = \left(\Pi, \{D(a,b)\}_{(a,b)\in\Pi}\right)$ be a demand in G. For every pair $(a, b) \in \Pi$, let $\mathcal{P}_{a,b}$ denote the collection of all a-b paths in G. A* fractional routing, *or just a routing, of the demand $\mathcal{D}$, is a flow f, that assigns non-zero values $f(P) > 0$ only to flow-paths $P \in \bigcup_{(a,b)\in\Pi} \mathcal{P}_{a,b}$, such that, for all $(a, b) \in \Pi$, $\sum_{P\in\mathcal{P}_{a,b}} f(P) = D(a, b)$. The routing is defined by only specifying values $f(P)$ that are non-zero. The* congestion *of the routing is the congestion caused by the flow f in G. We say that the routing is* over paths of length at most d, *if every flow-path P with $f(P) > 0$ has length at most d. If, for every path P, $f(P) \in \{0, 1\}$, then we say that the routing is* integral. *In this case, we may represent the routing as a collection of paths $\mathcal{Q} = \{P \mid f(P) = 1\}$*

2.3 Routers A central object that we use is routers. Routers were introduced in [HHT24] as an object that is closely related to bounded-hop expanders. As we show below, routers are also closely related to the well-connected graphs, that were introduced in [Chu23]. The definition below slightly generalizes the definition of [HHT24].

DEFINITION 2.4. ($(\overline{w}, d, \eta)$-ROUTER.) *Let $G = (V, E)$ be a graph, let $\overline{w}$ be a weighting of its vertices, and let $S \subseteq V$ be a subset of vertices of G. Additionally, let $d, \eta \geq 1$ be parameters. We say that G is a* $(\overline{w}, d, \eta)$-router *for the set S of supported vertices, if every $\overline{w}$-restricted demand $\mathcal{D}$ defined over the vertices of S can be routed with congestion at most η in G, via paths of length at most d.*

We note that the inclusion of the set S of supported vertices in the above definition may seem redundant, since we could simply set the weight of each such vertex to 0, without the need to include the vertices of S in the definition. However, supported vertices will be useful when we consider special vertex weighings, such as $\overline{\Delta}$ for $\Delta \in \mathbb{R}^{>0}$, and $\overline{\deg_G}$.

2.4 Well-Connected Graphs and their Relationship to Routers We employ well-connected graphs, that were introduced in [Chu23]. We start with a formal definition of such graphs, which is identical to that from [Chu23].

DEFINITION 2.5. (WELL-CONNECTED GRAPH) *Given an n-vertex graph G, a set S of its vertices called* supported vertices*, and parameters $\eta, d > 0$, we say that graph G is (d, η)-well-connected with respect to S, if, for every pair $A, B \subseteq S$ of disjoint equal-cardinality subsets of supported vertices, there is a collection $\mathcal{P}$ of paths in G, that connect every vertex of A to a distinct vertex of B, such that the length of each path in $\mathcal{P}$ is at most d, and every edge of G participates in at most η paths in $\mathcal{P}$.*

For intuition, it may be convenient to think of $d = 2^{\mathrm{poly}(1/\epsilon)}$, $\eta = n^{O(\epsilon)}$, and $|S| \geq |V(G)| - n^{1-\epsilon}$, for some precision parameter $0 < \epsilon < 1$.

2.4.1 The Relationship between Routers and Well-Connected Graphs The notion of routers is closely related to the notion of well-connected graphs. It is immediate to see that, if G is a $(\overline{w}, d, \eta)$-router with respect to any set S of supported vertices, for any vertex weighting $\overline{w}$ where $w(v) \geq 1$ for all $v \in V(G) \setminus S$, then G is (d, η)-well-connected with respect to S.

For the connection in the opposite direction, at the first glance, it appears that the notion of well-connected graphs is weaker, for two reasons. The first reason is that, from the definition, well-connected graphs only provide routings between pairs A, B of vertex subsets, and it appears that one cannot control the specific pairs that are being routed. The second reason is that they only guarantee the routing of $|A| = |B|$ flow units from A to B, which roughly corresponds to routing with vertex weighting $\overline{\Delta}$ where $\Delta = 1$. We first address the former issue, and show, in the following claim, that in fact a (d, η)-well-connected graph is a sufficiently strong router. We do not use this claim directly, but we feel that it is useful in that it illuminates the relationship between these two objects. The proof of the claim is deferred to the full version of the paper.

CLAIM 2.1. *Let G be an n-vertex graph, and assume that it is a (d, η)-well-connected graph with respect to a set S of supported vertices. Then for all $1 \leq k \leq \log n$, G is a $(1, d', \eta')$-router with respect to S, for $d' = 512k \cdot d$ and $\eta' = \left(2^{17} \cdot kdn^{1/k} \log^2 n\right) \cdot \eta$.*

There are three approaches to address the second issue that is raised above – namely, that the routing guarantees provided by well-connected graphs correspond to vertex weighting $w(v) = 1$ for all vertices v. The first approach has to do with the manner in which well-connected graphs are used. Most if not all applications of such graphs known today construct a well-connected graph W that is embedded into a given input graph G via the *distanced-matching game*, that can be thought of as the analogue of the cut-matching game for constructing a well-connected graph. From the construction, graph W has a rather low degree, so it can be thought of as providing routing

for the weighting $\overline{\deg}_W$ of its vertices. Moreover, in the typical application of well-connected graphs as described above, the embedding of W into G is usually then used as a certificate that G itself is a well-connected graph, and is thus a router with respect to vertex weighting $w(v) = 1$ for all vertices v, from Claim 2.1. In order to obtain a router certificate for a general weighting $\overline{w}$ of the vertices of G, we can construct a larger graph W, containing $\sum_{v \in V(G)} w(v)$ vertices, and then embed it into G via the distanced-matching game, where, for every vertex $v \in V(G)$, we embed $w(v)$ vertices of W into v. Such an embedding would then serve as a certificate that G is a router for the vertex weighting $\overline{w}$. The second approach is that, given any graph H with arbitrary integral vertex weighing $\overline{w}$, we can obtain a new graph H' via the following simple transformation: for every vertex $v \in V(H)$, add a new collection $T(v)$ of $w(v)$ vertices to the graph, connecting them to v with new edges. Let $S = \bigcup_{v \in V(H)} T(v)$. Then if graph H' is well-connected with respect to the set S of supported vertices, the original graph H is a router with respect to vertex weighing $\overline{w}$. Lastly, the third approach is to generalize the definition of well-connected graphs directly to arbitrary vertex weighting. In view of the first two approaches, it is not clear to us if such a generalization provides an additional value.

2.5 Length-Constrained Expanders Length-constrained expanders were initially introduced in [HRG22] as a generalization of the standard notion of expanders, that allows routing via paths of sublogarithmic length; this was also the main motivation behind the well-connected graphs introduced in [Chu23]. We do not use length-constrained expanders directly (except that we use routers that can be viewed as bounded-diameter length-constrained expaners). We provide a short discussion here for completeness.

Consider any graph G, and a demand $\mathcal{D} = \left(\Pi, \{D(a,b)\}_{(a,b) \in \Pi}\right)$ for G. We say that the demand is *h-length* if, for every pair $(a,b) \in \Pi$ of vertices, $\mathsf{dist}_G(a,b) \leq h$. The following description of length-constrained expanders follows the summary in [HHG22]. The formal definition of (h,s)-length φ-expanders relies on the notion of so-called "moving cuts", and can be found in [HRG22]. Instead, as in prior work on graph spanners (e.g., [HHT23] and [BHP24]), we use the more convenient *routing characterization* of length-constrained expanders. Informally, up to a constant factor in the length parameter, and a poly-logarithmic factor in the congestion parameter, a graph G is an (h,s)-hop φ-expander for a vertex weighing $\overline{w}$, if any h-length $\overline{w}$-bounded demand $\mathcal{D}$ can be routed in G via paths of length at most $(s \cdot h)$, with congestion at most roughly $\frac{1}{\varphi}$. The precise characterization of length-constrained expanders in terms of routing appers in the following theorem.

THEOREM 2.1. (THEOREM 5.17 OF [HHT24]) *Let G be a graph, let $\overline{w}$ be a weighting of its vertices, and let $h \geq 1$, $\varphi < 1$, and $s \geq 1$ be parameters. Then:*

1. *if G is an (h,s)-length φ-expander for vertex weighting $\overline{w}$, then every h-length $\overline{w}$-restricted demand $\mathcal{D}$ can be routed in G via paths of length at most $(s \cdot h)$, with congestion $O\left(\frac{\log(n)}{\varphi}\right)$; and*
2. *if G is not an (h,s)-length φ-expander for vertex weighting $\overline{w}$, then there is some h-length $\overline{w}$-bounded demand $\mathcal{D}$ that cannot be routed in G via paths of length at most $(\frac{s}{2} \cdot h)$ with congestion at most $\frac{1}{2\varphi}$.*

The next claim summarizes the connection between routers and length-constrained expanders that was shown by [HRG22, HHT24], and also follows from Theorem 2.1.

CLAIM 2.2. (IMMEDIATE FROM LEMMA 5.24 IN [HHT24]) *If G is an (h,s)-length φ-expander for vertex weighting $\overline{w}$ with $\mathsf{diam}(G) \leq h$, then G is a $\left(\overline{w}, hs, O\left(\frac{\log n}{\varphi}\right)\right)$-router for the set $S = V(G)$ of supported vertices. Conversely, if an G is a graph of diameter at most h that is not an (h,s)-length φ-expander for vertex weighting $\overline{w}$, then G is not a $\left(\overline{w}, \frac{hs}{2}, O\left(\frac{1}{2\varphi}\right)\right)$-router.*

To summarize, one can intuitively think of a $(\overline{\deg}, d, \eta)$-router as being roughly equivalent to a bounded-diameter length-constrained φ-expander, where $\varphi = \tilde{\Theta}(1/\eta)$, and the bound on the diameter is roughly d.

2.6 The Online-Batch Dynamic Model and Deamortization Our algorithm uses the by now standard online-batch dynamic model, and a reduction from this model to the standard dynamic model with bounded worst-case update time, that was formalized in [JS22]. Much of the exposition in this subsection is borrowed from Section 10.1 of [JS22]

We assume that we are given two main parameters: batch number σ and sensitivity parameter w. We let DS be a data stucture that we would like to maintain for an input graph G undergoing updates. A dynamic algorithm for maintaining DS in the online-batch dynamic model consists of $\sigma+1$ phases $\Phi_0,\ldots,\Phi_\sigma$, where Φ_0 is the *preprocessing*, or the *initialization* phase, and $\Phi_1,\ldots,\Phi_\sigma$ are *regular* phases. For all $1\leq i\leq\sigma$, at the beginning of Phase Φ_i, the algorithm is given the ith batch Π_i of updates to graph G; in our case, all updates are insertions and deletions of edges to and from G. The number of such updates in each batch Π_i must be bounded by w. The algorithm is required to maintain the data structure DS, so that, at the end of the 0th phase, the data structure DS that it obtains correctly corresponds to the initial graph $G^{(0)}$. Additionally, for all $1\leq i\leq\sigma$, the data structure DS that the algorithm obtains at the end of Phase Φ_i must correctly correspond to the graph $G^{(i)}$, that is obtained from the initial graph $G^{(0)}$, after the sequence $\Pi_1\circ\cdots\circ\Pi_i$ of updates is applied to it. The algorithm has amortized time t, if, for all $1\leq i\leq\sigma$, the running time of the algorithm during Phase Φ_i is bounded by $t\cdot|\Pi_i|$. Here, t is allowed to be some function of the graph parameters, such as, for example, n^ϵ, where $n=|V(G)|$, and ϵ a given parameter.

The following lemma was formally proved in [JS22], and is implied by the previous work of [NSWN17]. We provide a slightly simplified version of the lemma that is sufficient for us.

Lemma 2.1. (Lemma 10.1 in [JS22]; see also Section 5 in [NSWN17]) *Let G be a graph undergoing batch updates, and let $\sigma>0$ and $w\geq 2\cdot 6^\sigma$ be parameters. Assume that there is an algorithm in the batch update model with batch number σ and sensitivity parameter w, that maintains some data structure DS for G, whose preprocessing time is T_0, and amortized update time is T', where both T_0 and T' are functions that map some graph measure (e.g. $|E(G^{(0)})|$ or $|V(G)|$) to non-negative numbers.*

Then there is a dynamic algorithm, with preprocessing time $O\left(2^{O(\sigma)}\cdot T_0\right)$, and worst case update time $O\left(4^\sigma\cdot\left(\frac{T_0}{w}+w^{1/\sigma}\cdot T'\right)\right)$, that maintains a set of $O(2^{O(\sigma)})$ instances of the data structure DS, such that, after each update, the algorithm indicates one of the maintained instances of the data structure as the "current" one. The current instance of data structure DS satisfies the following conditions:

- *it is up-to-date with respect to the current graph; and*
- *the online-batch update algorithm was executed at most σ times on it, with each update batch size at most w.*

3 Low-Diameter Router Decomposition

In this section we formally define one of our main combinatorial objects – a Low-Diameter Router Decomposition, that, for conciseness, we refer to as "Router Decomposition". We then provide the statement of our main technical result, namely, an algorithm that maintains such a decomposition in the online-batch dynamic model, and show that the proof of Theorem 1.3 follows from it. We complete the proof of the main technical result in Sections 4–8. We start with a formal definition of a router decomposition.

Definition 3.1. (Router Decomposition of a Graph) *Let G be a simple graph, and let $\Delta^*,\tilde{d},\tilde{\eta}$ and ρ be non-negative parameters. A* router decomosition *of G with parameters $\Delta^*,\tilde{d},\tilde{\eta}$ and ρ consists of the following ingredients:*

- *a collection $\mathcal{C}$ of subgraphs of G called* clusters *with $\sum_{C\in\mathcal{C}}|V(C)|\leq\rho\cdot|V(G)|$, such that every cluster $C\in\mathcal{C}$ is a $(\overline{\deg}_C,\tilde{d},\tilde{\eta})$-router with respect to the set $V(C)$ of supported vertices, and every edge of G lies in at most one cluster $C\in\mathcal{C}$; and*

- *for every cluster $C \in \mathcal{C}$, a subgraph $C' \subseteq C$ with $V(C') = V(C)$, such that C' is a $(\overline{\Delta}^*, \tilde{d}, \tilde{\eta})$-router for the set $V(C)$ of supported vertices, and $\sum_{C \in \mathcal{C}} |E(C')| \leq \rho \cdot \Delta^* \cdot |V(G)|$.*

We denote by $E^{\mathrm{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$ the set of all edges of G that do not lie in any cluster of $\mathcal{C}$.

We note that generally we will ensure that $|E^{\mathrm{del}}|$ is sufficiently small in a router decomposition. The following theorem summarizes our main technical result, and a significant part of the paper is devoted to proving it.

THEOREM 3.1. *There is a deterministic algorithm, that receives as input a parameter n that is greater than a large enough constant, and additional parameters $512 \leq k \leq (\log n)^{1/49}$, $\frac{1}{k} \leq \delta < \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer, Δ^*, and Δ, such that $n^{18/k} \leq \Delta^* \leq \frac{\Delta}{n^{8/k^2}} \leq n$ holds. The algorithm is also given as input a simple graph G with $|V(G)| \leq n$, such that, initially, all vertex degrees in G are at most $\Delta \cdot n^{16/k}$. Lastly, the algorithm receives k online batches $\pi_1, \ldots, \pi_k$ of updates to graph G, where for $1 \leq i \leq k$, the ith batch π_i is a collection of at most $\Delta \cdot n$ edges that are deleted from G.*

The algorithm maintains a router decomposition $\left(\mathcal{C}, \{C'\}_{C \in \mathcal{C}}\right)$ of the graph G, for parameters Δ^*, $\tilde{d} = k^{19} \cdot 2^{O(1/\delta^6)}$, $\tilde{\eta} = n^{17/k}$, and $\rho = n^{20/k}$. After the initialization, $|E^{\mathrm{del}}| \leq |V(G)| \cdot \Delta \cdot n^{2/k^2}$ holds, and, for all $1 \leq i \leq k$, at most $|\pi_i| \cdot n^{7/k^2}$ edges may be added to E^{del} while processing the batch π_i. For all $1 \leq i \leq k$, after the ith batch of updates is received, the clusters $C \in \mathcal{C}$ may only be updated by removing some of their edges (which are then added to E^{del}) and some of their vertices. The corresponding graphs $C' \subseteq C$ may be updated by removing or inserting edges into them, and by removing vertices from them. Additionally, clusters of $\mathcal{C}$ that become empty are removed from $\mathcal{C}$. These are the only updates to the router decomposition. The initialization time of the algorithm is $O\left((n\Delta)^{1+O(\delta)}\right)$, and for all $1 \leq i \leq k$, the time required to process the ith update batch π_i is bounded by $O\left(|\pi_i| \cdot n^{O(1/k^2)}\right)$. The total number of edges inserted into or deleted from all subgraphs in $\{C' \mid C \in \mathcal{C}\}$ while batch π_i is processed is at most $O\left(|\pi_i| \cdot n^{O(1/k^2)}\right)$.

We will prove Theorem 3.1 in Sections 4–8. In the remainder of this section, we complete the proof of Theorem 1.3 using it. We start with the following corollary of Theorem 3.1, that can be thought of as an analogue of our main technical result – Theorem 1.3, for the online-batch dynamic model. The proof of the corollary is deferred to the full version of the paper.

COROLLARY 3.1. *There is a deterministic algorithm, that receives as input a simple n-vertex graph G, where n is greater than a large enough constant, and additional parameters $512 \leq k \leq (\log n)^{1/49}$, $\frac{1}{k} \leq \delta < \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer, and $\Delta^* \geq n^{19/k}$. Lastly, the algorithm receives k online batches $\pi_1, \ldots, \pi_k$ of updates to graph G, where for $1 \leq i \leq k$, the ith batch π_i is a collection of edge insertions and deletions for G; graph G is guaranteed to remain simple throughout the update sequence.*

The algorithm maintains a router decomposition $\left(\mathcal{C}, \{C'\}_{C \in \mathcal{C}}\right)$ of the graph G, for parameters Δ^*, $\tilde{d} = k^{19} \cdot 2^{O(1/\delta^6)}$, $\tilde{\eta} = n^{19/k}$, and $\rho = n^{O(1/k)}$, such that $|E^{\mathrm{del}}| \leq n^{1+O(1/k)} \cdot \Delta^*$ holds. The algorithm also maintains a graph H with $V(H) = V(G)$ and $E(H) = E^{\mathrm{del}} \cup \left(\bigcup_{C \in \mathcal{C}} E(C')\right)$, with $|E(H)| \leq n^{1+O(1/k)} \cdot \Delta^*$. The initialization time of the algorithm is $O\left(m^{1+O(\delta)}\right)$, where m is the initial number of edges and vertices in G. For all $1 \leq i \leq k$, the time required to process the ith update batch π_i is bounded by $O\left(|\pi_i| \cdot n^{O(\delta)}\right)$. The number of edge insertions and deletions that graph H undergoes while batch π_i is processed is bounded by $O\left(|\pi_i| \cdot n^{O(1/k)}\right)$.

We are now ready to complete the proof of Theorem 1.3. Let G be an initial graph with n vertices and no edges, that undergoes an online sequence of at most n^2 edge deletions and insertions, such that G remains a simple graph throughout the update sequence. We can assume w.l.o.g. that n is greater than a sufficiently large constant, since, if it is not the case, we can simply increase it, and add some dummy vertices to G. We also assume that we are given parameters $512 \leq k \leq (\log n)^{1/49}$, $\frac{1}{k} \leq \delta < \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer, and $\Delta^* \geq n^{19/k}$ (if n was initially too small, then Δ^* may need to be increased to accommodate the increase in n).

Notice that the algorithm from Corollary 3.1 allows us to maintain the collection $\mathcal{C}$ of edge-disjoint clusters of G, the subgraphs $C' \subseteq C$ for all clusters $C \in \mathcal{C}$, and the graph H with all required properties, in the online-batch dynamic model, with batch number k and sensitivity parameter $w = n^2$. The initialization time of the algorithm is bounded by $O(n^{2+O(\delta)})$, and, for all $1 \leq i \leq k$, the time required to process the ith batch π_i of updates is bounded by $O(|\pi_i|) \cdot n^{O(\delta)}$. We denote the data structure maintained by the algorithm from Corollary 3.1 by DS.

From Lemma 2.1, we then obtain an algorithm that maintains the collection $\mathcal{C}$ of clusters, and, for every cluster $C \in \mathcal{C}$, a subgraph $C' \subseteq C$ with all required properties, whose worst-case update time is $O\left(2^{O(k)} \cdot \left(n^{O(\delta+1/k)}\right)\right) \leq O\left(n^{O(\delta)}\right)$ (since $2^{O(k)} \leq n^{O(1/k)}$ as $k \leq (\log n)^{1/49}$, and $\delta \geq 1/k$).

The only subtlety is in maintaining the graph H, so that, after each update to graph G, graph H undergoes at most $n^{O(1/k)}$ updates. The reason is that the algorithm from Lemma 2.1 can be thought of as maintaining, at all times, 3^k different instances of data structure DS, each of which is associated with a different graph H. At any given time, the algorithm indicates which of the data structures is up to date. However, following a single update to a graph G, the algorithm may decide to switch the designated current data structure, which may lead to a large number of changes in the graph H. In order to overcome this difficulty, we will simply include in our final sparsifier H^* the edges of all graphs H that are maintained by all copies of data structure DS, but we explicitly exclude edges that have been deleted from G.

We now provide more details of the algorithm from Lemma 2.1, and the algorithm for maintaining the final graph H^*. It is convenient to think of the algorithm from Lemma 2.1 as consisting of $k+1$ levels, which are associated with a hierarchical partition of the time line into intervals. Let T be the time horizon of the algorithm. We first partition T into consecutive intervals of length w each, and denote the resulting collection of intervals by $\mathcal{I}_0$; we refer to these intervals as *level-0 intervals*. We can think of the level-0 algorithm as having 3^{k+1} copies of data structure DS from Corollary 3.1, that are partitioned into 3 groups $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$. Each group $\mathcal{G}_r$ contains 3^k identical copies of data structure DS. For each interval $I \in \mathcal{I}_0$, every data structure DS is in one of the following three states: (i) preparation: given some initial graph G' at the beginning of interval I, the data structure needs to initialize the algorithm from Corollary 3.1 for graph G', during the first half of interval I; or (ii) active: the data structure will be used, during time interval I, by data structures from levels $1, \ldots, k$; or (iii) rollback: we can think of this part as "undoing" all computation that has been done during the preparation step. If data structure DS is active during interval I, then it is in preparation state during the preceding interval, and in the rollback state during the following interval. Moreover, all data structures in a single group $\mathcal{G}_r$ are always in the same state, and exactly one group is active during each interval $I \in \mathcal{I}_0$. Recall that the time to initialize the data structure DS is bounded by $n^{2+O(\delta)}$, and during this time the initial graph H_{DS}, that contains all edges of E^{del} and $\bigcup_{C \in \mathcal{C}} E(C')$ is constructed. We can think of the construction of the graph H_{DS} as being stretched over the course of the first half of a single level-0 interval when the data structure is in the preparation state, where we start with H_{DS} containing no edges, and then insert at most $n^{O(1/k)}$ edges into H_{DS} per time unit. Then during the rollback state, we will delete the edges from H_{DS}, at the rate of $n^{O(1/k)}$ edge deletions per time unit, until no edges remain in H_{DS}. We define the level-0 graph H^0 to contain the edges of all graphs H_{DS}, for all level-0 data structures DS, excluding the edges that no longer lie in G. Note that graph H^0 undergoes at most $n^{O(1/k)} \cdot 2^{O(k)} \leq n^{O(1/k)}$ updates per time unit.

Consider now some integer $1 \leq j \leq k$, and some level-$(j-1)$ interval I. This interval is partitioned into $\lceil n^{2/k} \rceil$ level-j intervals of identical length; we denote the length of each such interval by L_j. Moreover, there are 3^{k+1-j} identical copies of data structure DS that are active during time interval I. Each of these copies is obtained by applying the algorithm from Corollary 3.1 to some initial graph G, which then undergoes $j-1$ batches of updates. These copies are in turn partitioned into 3 groups $\mathcal{G}_1^j, \mathcal{G}_2^j, \mathcal{G}_3^j$, each of which contains 3^{k-j} identical copes of the data structure. For each level-j interval $I' \subseteq I$, each such data structure is in one of the following three states: (i) preparation: the data structure is processing the jth batch of updates, whose size is at most $4L_j \cdot n^{2/k}$, and the processing needs to complete by the middle of interval I'; (ii) active: the data structure is ready to be used by levels $j+1, \ldots k$; and (iii) rollback: we undo the changes that the data structure underwent during the preparation state. As before, if data structure DS is active during a level-j interval I, then it is in preparation state in the preceding interval, and in rollback state during the subsequent interval. Consider now any copy DS of the data structure, and suppose it is in preparation state during the level-j interval I'. Recall that the length of the interval is denoted by L_j, and the length of the update batch to the data structure is at most $4L_j \cdot n^{2/k}$.

During this update sequence, the graph H_{DS} associated with the data structure undergoes at most $L_j \cdot n^{O(1/k)}$ edge insertions and deletions. We think of these updates as being spread over the course of interval I, so graph H_{DS} undergoes at most $n^{O(1/k)}$ updates per time unit. During the following rollback stage, we undo the changes that graph H_{DS} underwent during the preparation stage, with the graph undergoing at most $n^{O(1/k)}$ updates per time unit. We now define the level-j graph H^j to be the union of all graphs H_{DS} for all data structures $\mathsf{DS} \in \mathcal{G}_1^j \cup \mathcal{D}_2^j \cup \mathcal{D}_3^j$, but we do not include in H^j edges that have been deleted from G. As before, graph H^j undergoes at most $n^{O(1/k)} \cdot 2^{O(k)} \leq n^{O(1/k)}$ updates per time unit.

We let $j^* \leq k-1$ be smallest integer, such that level-j^* time intervals have length at most $n^{2/k}$ each. In this case, we think of level-(j^*+1) time intervals as consisting of a single time slot each. For each level-(j^*+1) time interval $I = \{\tau\}$, there is at least one data structure DS, that is up to date with respect to the graph G at time τ. This is the data structure that the algorithm from Lemma 2.1 considers as current. The corresponding graph H_{DS} must then be contained in the graph $H^* = H^0 \cup \cdots \cup H^{j^*+1}$ that our algorithm maintains. From our discussion, graph H^* undergoes at most $n^{O(1/k)}$ edge insertions or deletions per time unit. Since the algorithm from Corollary 3.1 ensures that each graph H maintained by a copy of the data structure contains at most $n^{1+O(1/k)} \cdot \Delta^*$ edges, and since, at all times, our graph H^* is contained in the union of at most $2^{O(k)}$ such graphs, we get that $|E(H^*)| \leq 2^{O(k)} \cdot n^{1+O(1/k)} \cdot \Delta^* \leq n^{1+O(1/k)} \cdot \Delta^*$ holds. This completes the proof of Theorem 1.3.

In the remainder of this paper we prove Theorem 3.1. We start by providing the main tools that we use in the proof of the theorem.

4 First Tool: a Nice Router and its Pruning

In this section we provide an explicit construction of a router, and provide an algorithm for pruning this router. We show that the graph remains a router even after pruning.

4.1 Construction of a Nice Router Graph

We assume that we are given three integral parameters: $k \geq 256$, $N \geq k^{3k}$, and $\Delta \geq 4k^2$. We sometimes refer to Δ the *target degree*. We now describe a construction of a corresponding router, that we denote by $W_k^{N,\Delta}$. To avoid clatter, in the remainder of this section, we assume that N and Δ are fixed, and we do not include them in superscripts, so in this section we denote graph $W_k^{N,\Delta}$ by W_k. The construction of graph W_k is recursive. We start by constructing a graph W_1, and then, for $1 < i \leq k$, we construct graph W_i by combining N copies of graph W_{i-1}. Parameters N and Δ remain unchanged for these graphs.

Graph W_i – a High Level View. Consider an integer $1 \leq i \leq k$. We now provide a high-level description of graph $W_i = (V_i, E_i)$. The set V_i contains exactly N^i vertices, and it is partitioned into two subsets: set V_i^c of N^{i-1} vertices called *center* vertices, and set $V_i^\ell = V_i \setminus V_i^c$ of remaining vertices, called *leaf* vertices. The set $E(W_i)$ of edges of the graph is partitioned into i subsets $E_1^i, \ldots, E_i^i$, where for $1 \leq z \leq i$, we refer to the edges of E_z^i as *level-z edges*. If we let $H_z = G_i[E_z^i]$ be the subgraph of G_i induced by the set E_z^i of level-z edges, and denote by $\mathcal{R}_z$ the collection of the connected components of H_z, then each such connected component $C \in \mathcal{R}_z$ contains exactly N vertices, out of which exactly one is a center vertex. Additionally, if we let S be a star defined over the vertices of C, whose center is the unique vertex of $V(C) \cap V_i^c$, then C can be obtained from S by replacing every edge of S with Δ parallel copies (we refer to these copies as a *bundle*; the edges of S are referred to as *superedges*). We denote by $\mathcal{S}_z$ a collection of star graphs that contains, for every connected component $C \in \mathcal{R}_z$, the corresponding star graph S on $V(C)$ (without the parallel edges). Note that every edge in W_i connects a leaf vertex to a center vertex. We now describe the construction of the graph W_i in more detail. The construction is recursive, from smaller to larger values of i.

Graph W_1. The set of vertices of the graph W_1 is $V_1 = \{v_1, \ldots, v_N\}$, where vertex v_1 is the *center* vertex, and vertices $v_2, \ldots, v_N$ are *leaf vertices*. In other words, $V_1^c = \{v_1\}$, and $V_1^\ell = \{v_2, \ldots, v_N\}$. Let S be a star defined on the vertices of V_1, whose center is vertex v_1, and leaves are $v_2, \ldots, v_N$. Graph W_1 is obtained from S by replacing every edge of S with a collection of Δ parallel edges. All edges of W_1 belong to level 1, so $E(W_1) = E_1^1$. The collection $\mathcal{S}_1$ of level-1 stars contains a single star S.

Graph W_i for $i > 1$. We assume that we are given an integer $i > 1$, and that we have already defined the graph W_{i-1}. We now describe the construction of graph W_i.

We start by creating N copies of graph W_{i-1}, that we denote by $C_1, \ldots, C_N$. We refer to $C_1, \ldots, C_N$ as *level-*$(i-1)$ *clusters*. We let the set V_i of vertices of W_i be $\bigcup_{j=1}^{N} V(C_j)$. For all $1 \leq j \leq N$, if $v \in V(C_j)$ is a center vertex for C_j, then it becomes a center vertex for W_i, and otherwise it becomes a leaf vertex for W_i. Recall that, for all $1 \leq j \leq N$, $|V(C_j)| = N^{i-1}$, and the number of center vertices in C_j is N^{i-2}. Therefore, we get that $|V_i| = N^i$ and the number of center vertices in W_i is N^{i-1}, as required.

For all $1 \leq i' < i$, the set $E_{i'}^i$ of level-i' edges of W_i is the union of the sets of level-i' edges of $C_1, \ldots, C_N$. It now remains to define the set E_i^i of level-i edges for graph W_i. In order to do so, we construct a collection $\mathcal{S}_i$ of N^{i-1} disjoint stars, where each star $S \in \mathcal{S}_i$ contains exactly one vertex from each of the clusters $C_1, \ldots, C_N$, and the center of the star lies in V_i^c. The set E_i^i of level-i edges is then obtained by replacing each edge $e \in \bigcup_{S \in \mathcal{S}_i} E(S)$ with Δ parallel edges. We omit the details of the construction of the collection $\mathcal{S}_i$ of stars with the above properties; they can be found in the full version of the paper.

Consider now the level-k graph W_k. For all $1 \leq i < k$, graph W_k contains N^{k-i} copies of the level-i graph W_i. We let $\mathcal{C}_i$ denote the partition of the vertices of W_k defined by these copies of W_i. We refer to $\mathcal{C}_i$ as *level-i clustering*, and we refer to each subgraph $C \in \mathcal{C}_i$ as a *level-i cluster*. Notice that for all $1 \leq i < k$, each level-i cluster is contained in a single level-$(i+1)$ cluster, and moreover, each level-$(i+1)$ cluster contains the union of exactly N level-i clusters. There is a single level-k cluster, containing all vertices of W_k. From our construction, it is easy to verify that, for all $1 \leq i \leq k$, if we denote by $H_i = W_k[E_i^k]$ the subgraph of W_k induced by the level-i edges, then there is a collection $\mathcal{S}_i$ of N^{k-1} disjoint star graphs, where each star $S \in \mathcal{S}_i$ has N vertices, and the center of S lies in W_k^c, such that the set E_i^k of edges can be obtained from $E_i' = \bigcup_{S \in \mathcal{S}_i} E(S)$, by replacing each edge $e \in E_i'$ with Δ parallel edges. We refer to the set $\mathcal{S}_i$ of stars as *level-i stars*. Lastly, note that for all $1 \leq i \leq k$, the number of level-i superedges in W_k is at most N^k, and $|E_i^k| \leq \Delta \cdot N^k$. Overall, W_k contains at most $k \cdot N^k$ superedges, and $|E(W_k)| \leq k \cdot \Delta \cdot N^k$. Moreover, the degree of every vertex in W_k is at least $\Delta \cdot k$, and the degree of every leaf vertex is exactly $\Delta \cdot k$.

We do not directly prove that the graph W_k is a router. Instead, we first define *properly pruned* subgraphs of W_k, and we prove that each such graph is a router.

4.2 Properly Pruned Subgraphs of W_k

For all $1 \leq i \leq k$, we now define the notion of a *properly pruned subgraph* of W_i. We then show that any Δ-restricted demand can be routed with low congestion on paths of length $O(k^2)$ in a properly pruned subgraph of W_k.

DEFINITION 4.1. *For $1 \leq i \leq k$, a subgraph $W \subseteq W_i$ is a* properly pruned *subgraph of W_i with respect to sets $U_1, U_2, \ldots, U_i \subseteq V(W_i)$ of its vertices, where $U_i \subseteq U_{i-1} \subseteq \cdots \subseteq U_1$, if the following hold:*

- *P1. for every level $1 \leq i' \leq i$, for every leaf vertex $v \in V_i^\ell$:*
  - *if $v \in U_{i'}$, then at least $\lceil \Delta/2 \rceil$ level-i' edges that are incident to v in W_i lie in W (recall that all these edges must be parallel edges that correspond to the unique level-i' superedge that is incident to v);*
  - *otherwise, W_i contains no level-i' edges that are incident to v;*
- *P2. for each level $1 \leq i' \leq i$, for every center vertex $v \in V_i^c$, if we denote by $S \in \mathcal{S}_{i'}$ the unique level-i' star for which v serves as a center, then:*
  - *if $v \in U_{i'}$, then at least $N \cdot \left(1 - \frac{1}{k}\right)$ leaf vertices of S lie in $U_{i'}$ (and we say that star S* survives *in this case);*
  - *otherwise, no leaf vertices of S lie in $U_{i'}$ (and we say that star S is* destroyed*);*
- *P3. $V(W) = U_1$; and*
- *P4. for every level $1 \leq i' < i$, for every level-i' cluster $C \in \mathcal{C}_{i'}$ of W_i, either $V(C) \cap U_1 = \emptyset$ (in which case we say that cluster C is* destroyed*), or at least $\frac{|V(C)|}{k^{3k}}$ vertices of C lie in $U_{i'+1}$ (and we say that cluster C* survives*).*

4.3 Routing in Properly Pruned Subgraphs of W_k The following theorem, whose proof is deferred to the full version of the paper, shows that a properly pruned subgraph of W_k is a good router.

THEOREM 4.1. *Let W be a properly pruned subgraph of W_k, with respect to sets $U_1, \ldots, U_k$ of its vertices, and let $\mathcal{D} = \left(\Pi, \{D(a,b)\}_{(a,b)\in\Pi}\right)$ be a $\frac{\Delta}{k^{4k}}$-restricted demand for W. Then there is a fractional routing of $\mathcal{D}$ in W with no edge-congestion, such that every flow-path P with $f(P) > 0$ has length at most $20k^2$. Moreover, there is an algorithm, that, given the graph W, the sets $U_1, \ldots, U_k$ of its vertices, and the demand $\mathcal{D}$, computes such a routing in time $O\left(k^2 \cdot \Delta \cdot N \cdot (N^k + |\Pi|)\right)$.*

4.4 Efficient Pruning of Graph W_k In this subsection we provide an algorithm to efficiently prune the graph $W_k^{N,\Delta}$, as it undergoes an online sequence of edge deletions. In order to motivate the model that we use in this section, we note that our approach for proving Theorem 3.1 uses the online-batch dynamic update model (see Section 2.6). Intuitively, during initialization, we will construct an initial clustering $\mathcal{C}$ of the input graph G, and, for every cluster $C \in \mathcal{C}$, we will compute an embedding of a graph $W_k^{N,\Delta}$, for suitably chosen parameters N and Δ, into C. In order to ensure that C has strong routing properties, we need to ensure that every vertex of C participates in the embeddings of many edges of $W_k^{N,\Delta}$. Recall that in the online-batch dynamic update model, the algorithm receives updates in σ batches $\pi_1, \ldots, \pi_\sigma$. In our case, we will use $\sigma = k$, and the only type of updates we will need to process is the deletion of edges from the input graph. Consider now some cluster C, and let E' be the set of edges deleted from C in a single batch π_i. Then every edge $e \in E(W_k^{N,\Delta})$ whose embedding path contains an edge from E' needs to be deleted from $W_k^{N,\Delta}$. In order to ensure that we obtain a properly pruned subgraph $W \subseteq W_k^{N,\Delta}$, we may delete some additional edges from W. As the result of these deletions, it is possible that, for some vertices $v \in V(C)$, the number of edges $e \in E(W)$ whose embedding path contains v becomes too small. When this happens, we need to delete v from C, and we need to further delete all edges $e \in E(W)$, whose embedding path uses v, from W. This, in turn, may trigger another round of pruning, and so on. Therefore, even though our algorithm for maintaining a router decomposition works in the online-batch dynamic model, the input to the pruning algorithm does not arrive in batches. Instead, we partition the timeline into phases $\Phi_0, \ldots, \Phi_k$. Phase Φ_0 is special and is only used during initialization, while phases $\Phi_1, \ldots, \Phi_k$ can be thought of as corresponding to the batches $\pi_1, \ldots, \pi_k$ of updates to the router decomposition algorithm. For all $1 \leq \tau \leq k$, the pruning algorithm receives as input an online sequence E'_τ of edges that are deleted from $W_k^{N,\Delta}$ over the course of the phase Φ_τ. We emphasize that these edges do not arrive simultaneously, but rather their arrival occurs over the course of the phase Φ_τ. Intuitively, the algorithm needs to maintain subsets $U_1, \ldots, U_k$ of vertices of $W_k^{N,\Delta}$ with $U_k \subseteq \cdots \subseteq \cdots U_1$, and a subgraph $W \subseteq W_k^{N,\Delta}$, such that, at all times, W is a properly pruned subgraph of $W_k^{N,\Delta}$ with respect to $U_1, \ldots, U_k$. We require that, for all $0 \leq \tau \leq k$, the total number of vertices deleted from U_k over the course of the phase is not much larger than $|E'_\tau|/\Delta$, and the total number of edges deleted from W over the course of the phase is not much larger than $|E'_\tau|$. We now define a new problem, that we call NiceRouterPruning, which abstracts our pruning algorithm.

DEFINITION 4.2. (THE NiceRouterPruning PROBLEM) *The input to the* NiceRouterPruning *problem consists of parameters $k \geq 256$, $N \geq k^{3k}$ and $\Delta \geq 4k^2$, and an online sequence $e_1, \ldots, e_r$ of edge deletions from the corresponding graph $W_k = W_k^{N,\Delta}$. We denote the resulting dynamic graph by H, so $H^{(0)} = W_k$, and, for $t > 0$, $H^{(t)} = H^{(0)} \setminus \{e_1, \ldots, e_t\}$.*

The algorithm for the problem is required to maintain subsets $U_1, \ldots, U_k$ of vertices of H with $U_k \subseteq \cdots \subseteq U_1$, and a subgraph $W \subseteq H$ for which the following properties hold:

- *for all $1 \leq i \leq k$, at the beginning of the algorithm, $U_i = V(W_k)$, and, as the algorithm progresses, vertices may be deleted from U_i, but no vertices may be inserted into U_i;*
- *at all times, W must be a properly pruned subgraph of W_k with respect to the current sets $U_1, \ldots, U_k$; and*
- *the only changes that W undergoes over time is edge- and vertex-deletions.*

The algorithm's timeline is partitioned into $k+1$ phases $\Phi_0, \Phi_1, \ldots, \Phi_k$, and the algorithm is notified when a new phase starts. For all $1 \leq \tau \leq k$, if we denote by E'_τ the sequence of edge deletions from W_k that algorithm receives as input during the phase Φ_τ, then it is guaranteed that $|E'_0| \leq \frac{N^k \cdot \Delta}{2k^{4k}}$, and, for all $1 \leq \tau \leq k$, $|E'_\tau| \leq \frac{N^k \cdot \Delta}{k^{8k}}$. We require that, for all $0 \leq \tau \leq k$, the total number of vertices deleted from U_k over the course of phase Φ_τ is at most $\frac{|E'_\tau|}{\Delta} \cdot k^{4k}$, and the total number of edges deleted from W over the course of phase Φ_τ is at most $|E'_\tau| \cdot k^{7k+3}$.

The following theorem, whose proof is deferred to the full version of the paper, provides an efficient algorithm for the NiceRouterPruning problem.

THEOREM 4.2. (ALGORITHM FOR NiceRouterPruning) *There is a deterministic algorithm for the* NiceRouterPruning *problem, whose initialization time is $O(k^2 \cdot N^k)$, and the running time of each phase Φ_τ, for $0 \leq \tau \leq k$, is bounded by $O\left(|E'_\tau| \cdot k^{O(k)}\right)$.*

5 Second Tool: Embedding a Nice Router

One of the main notions that we use in this section is that of a scattered graph, that we define next.

DEFINITION 5.1. (SCATTERED GRAPH.) *Let G be a graph, and let $d \geq 1$ and $0 < \epsilon < 1$ be parameters. We say that G is (d, ϵ)-*scattered *if, for every vertex $v \in V(G)$, $|B_G(v, d)| \leq |V(G)|^{1-\epsilon}$.*

Next, we provide a simple algorithm, that, given a graph G and parameters d and ϵ, either correctly establishes that G is (d, ϵ)-scattered, or computes a vertex $v \in V(G)$, such that $|B_G(v, 4d/\epsilon)| \geq |V(G)|^{1-\epsilon}$. The proof follows standard techniques and is deferred to the full version of the paper.

CLAIM 5.1. *There is an algorithm, that, given an n-vertex graph G and parameters $d \geq 1$ and $0 < \epsilon < 1$, either correctly establishes that G is (d, ϵ)-scattered, or computes a vertex $v \in V(G)$, such that $|B_G(v, 4d/\epsilon)| \geq |V(G)|^{1-\epsilon}$. The running time of the algorithm is $O(m^{1+\epsilon})$, where $m = |E(G)|$.*

Next, we define a new problem, that we call EmbedOrScatter. The input to the problem consists of an n-vertex graph G, and parameters Δ and k. Additionally, we are given a distance parameter d, and a congestion parameter η. The goal of the algorithm is to either (i) compute an embedding of graph $W_k^{N,\Delta}$ into G via paths whose length is not much larger than d, that cause congestion at most η, where parameter N is chosen so that $N^k \geq n^{1-2/k}$; or (ii) compute a set E' of at most $2\Delta n$ edges of G, such that $G \setminus E'$ is $(d, 1/k)$-scattered. In the former case, we also allow the algorithm to compute a small subset $F \subseteq E(W_k^{N,\Delta})$ of edges, that we call *fake edges*, for which the algorithm does not need to provide an embedding. In addition to the above input, the algorithm is given a parameter $0 < \delta < 1$, that will determine the tradeoff between the lengths of the embedding paths (which will be bounded by $d \cdot k \cdot 2^{O(1/\delta^6)}$), and the running time of the algorithm (that will depend on m^δ, where $m = |E(G)|$). We now provide a formal definition of the EmbedOrScatter problem.

DEFINITION 5.2. (EmbedOrScatter PROBLEM) *The input to the* EmbedOrScatter *problem is a simple n-vertex graph G with no isolated vertices, and parameters $16 \leq \Delta \leq n$, $d \geq 1$, $2 \leq k < (\log n)^{1/3}$, and $\frac{2}{(\log n)^{1/24}} < \delta < \frac{1}{400}$, such that $1/\delta$ is an integer. We require that n is sufficiently large, so that $\sqrt{\log n} > 100 \log\log n$ holds. Let $N = \left\lfloor n^{1/k - 1/k^2} \right\rfloor$ and let $\eta = n^{1/k} \cdot d \cdot k^{ck} \cdot 2^{c/\delta^6}$ for a large enough constant c. The output of the problem is one of the following:*

- *either a collection $F \subseteq E(W_k^{N,\Delta})$ of at most $\frac{N^k \cdot \Delta}{(512k)^{4k}}$ edges of $W_k^{N,\Delta}$ (that we call* fake edges*), together with an embedding of $W_k^{N,\Delta} \setminus F$ into G via paths of length at most $d \cdot k \cdot 2^{O(1/\delta^6)}$ that cause congestion at most η; or*
- *a subset E' of at most $2n\Delta$ edges, such that $G \setminus E'$ is $(d, 1/k)$-scattered.*

In the following theorem, whose proof is deferred to the full version of the problem, we provide an algorithm for the EmbedOrScatter problem.

THEOREM 5.1. *There is a deterministic algorithm for the* EmbedOrScatter *problem, whose running time on input* $(G, \Delta, k, d, \eta, \delta)$ *is at most* $O\left(m^{1+O(\delta+1/k)} \cdot d^2 \cdot 2^{O(1/\delta^6)}\right)$, *where* $m = |E(G)|$.

6 Third Tool: Router Witness

In this section we show that an embedding of a nice router $W_k^{N,\Delta}$ into a graph C can be used to certify that the graph C is a strong router, and also provide an algorithm to compute a sparse subgraph of C that is a strong enough router. We start with the following lemma, whose proof is deferred to the full version of the paper.

LEMMA 6.1. *Let* C *be a graph, and let* $\alpha \geq 1, \Delta \geq 1$ *be parameters such that, for every vertex* $v \in V(C)$, $\deg_C(v) \leq \alpha \cdot \Delta$. *Assume that we are given a graph* W, *that is a properly pruned subgraph of* $W_k^{N,\Delta}$, *for some parameters* N *and* k, *and an embedding* $\mathcal{P}$ *of* W *into* C *via paths of length at most* d^*, *that cause congestion at most* η^*. *Assume further that every vertex* $v \in V(C)$ *lies on at least* $\frac{\Delta}{\beta}$ *paths in* $\mathcal{P}$. *Then* C *is a* $(\overline{\deg}_C, \tilde{d}, \tilde{\eta})$*-router with respect to the set* $V(C)$ *of its vertices, where* $\tilde{d} = 22d^* \cdot k^4$, *and* $\tilde{\eta} = 2\alpha \cdot \beta \cdot k^{4k+1} \cdot d^* \cdot \eta^*$.

Recall that a Router Decomposition requires that, for every cluster $C \in \mathcal{C}$, we provide a subgraph $C' \subseteq C$ with $V(C') = V(C)$ and $|E(C')| \leq \Delta^* \cdot |V(C)|$, such that C' is a $(\overline{\Delta}^*, \tilde{d}, \tilde{\eta})$-router for the set $V(C)$ of supported vertices. We next describe how C' is constructed, and how we certify that it is indeed a router with required properties. We will call C' a *sparsified router*.

6.1 A Sparsified Router

In this subsection we assume that we are given a simple graph C, together with parameters $1 \leq \Delta^* \leq \Delta$. Assume further that we are given another graph W, that is a properly pruned subgraph of $W_k^{N,\Delta}$, for some parameters N and k, together with an embedding $\mathcal{P}$ of W into C via paths of length at most d^*, that cause congestion at most η^*, such that every vertex $v \in V(C)$ lies on at least $\left\lfloor \frac{\Delta^*}{2kd^*} \right\rfloor$ paths in $\mathcal{P}$. Finally, assume that $\Delta^* \geq 2kd^*$.

We use a parameter $\Delta' = \left\lfloor \frac{\Delta^*}{2kd^*} \right\rfloor$. In order to construct a sparsified router C' for C, we select, for every superedge (u, v) of W, an arbitrary collection $E'(u, v) \subseteq E(W)$ of Δ' parallel edges (recall that, from Definition 4.1, graph W' must contain at least $\frac{\Delta}{2} \geq \frac{\Delta^*}{2} \geq \Delta'$ such parallel edges). Let $W' \subseteq W$ be the graph obtained from W by including, for every superedge (u, v) of W, only the edges of $E'(u, v)$. Additionally, for every vertex $v \in V(C)$, we select an arbitrary subset $\mathcal{P}(v) \subseteq \mathcal{P}$ of Δ' paths that contain v, such that every leaf vertex $u \in V(W)$ serves as an endpoint in at most $\gamma \cdot \Delta'$ paths in the multiset $\bigcup_{v \in V(C)} \mathcal{P}(v)$, for some parameter γ.

We let $C' \subseteq C$ be the simple graph obtained by taking the union of all paths in $\bigcup_{v \in V(C)} \mathcal{P}(v)$, and all paths of $\mathcal{P}$ that serve as embedding paths for the edges in $E(W')$. The following observation summarizes the properties of C'. The proof is deferred to the full version of the paper.

OBSERVATION 6.1. *Graph* C' *is a* $(\overline{\Delta}^*, \tilde{d}, \tilde{\eta}')$*-router with respect to the set* $V(C')$ *of supported vertices, where* $\tilde{d} = 22d^* \cdot k^4$, *and* $\tilde{\eta}' = 8\gamma(d^*)^2 \cdot \eta^* \cdot k^{4k+1}$. *Moreover,* $|E(C')| \leq |V(C)| \cdot \Delta^*$.

7 Fourth Tool: Decremental Low-Diameter Clustering

7.1 Problem Definition

We assume that we are given an undirected n-vertex graph G that undergoes an online adversarial sequence of edge deletions. We are also given a parameter $1 < k < \log n$, and we use another parameter $d = 4k^3$. We will sometimes refer to subgraphs of G as *clusters*. Next, we need the definition of a settled cluster, and of a valid clustering.

DEFINITION 7.1. (SETTLED CLUSTER) *We say that a cluster* $C \subseteq G$ *is* settled, *if either* $|V(C)| \leq n^{1/k}$; *or there is a vertex* $v \in V(C)$, *such that* $|B_C(v, d)| \geq |V(C)|^{1-1/k}$. *A cluster that is not settled is called* unsettled.

Definition 7.2. (Valid Clustering) *A valid clustering is a collection $\mathcal{C}$ of clusters of G, such that:*

- *every cluster $C \in \mathcal{C}$ is settled; and*
- *every edge in the current graph G lies in exactly one of the sets $\{E(C)\}_{C\in\mathcal{C}}$.*

We are now ready to define the Decremental Low-Diameter Clustering problem.

Definition 7.3. (Low-Diameter Clustering problem) *In the Decremental Low-Diameter Clustering problem, the input is a simple n-vertex graph G, and a parameter $1 < k < \log n$. Let $d = 4k^3$. At the beginning of the algorithm, an initial valid clustering $\mathcal{C}$ of the initial graph G must be produced by the algorithm. Over the course of the algorithm, the adversary maintains a partition of the set $\mathcal{C}$ of clusters into two subsets: set $\mathcal{C}^I$ of inactive clusters, and set $\mathcal{C}^A$ of active clusters. Once a cluster $C \in \mathcal{C}$ is added to $\mathcal{C}^I$, it may no longer be modified, and it remains in $\mathcal{C}^I$ until the end of the algorithm. Immediately after the initialization, $\mathcal{C}^A = \mathcal{C}$ and $\mathcal{C}^I = \emptyset$ holds.*

The algorithm proceeds over a number of phases. At the beginning of every phase, the adversary may delete some edges and vertices from some of the clusters in $\mathcal{C}^A$, and move some clusters from $\mathcal{C}^A$ to $\mathcal{C}^I$, after which every cluster that remains in $\mathcal{C}^A$ is guaranteed to become unsettled.

The algorithm is then required to update the current clustering, via a sequence of cluster splitting operations*, defined as follows. In every cluster splitting operation, we start with a cluster $C \in \mathcal{C}^A$, and a vertex-induced subgraph $C' \subseteq C$ with $|V(C')| \leq |V(C)|^{1-1/k}$. Cluster C' is then added to $\mathcal{C}$ and $\mathcal{C}^A$ as a new cluster, and the edges of $E(C')$ are deleted from C. Additionally, some vertices that have become isolated in C can be deleted from C. At the end of the phase, all clusters in $\mathcal{C}^A$ must be settled.*

For every vertex v, let n_v be the total number of clusters to which v ever belonged over the course of the algorithm. Then we additionally require that $\sum_{v\in V} n_v \leq 2n^{1+1/k}$ holds.

The main result of the current section is an algorithm for the Low-Diameter Clustering problem, that is summarized in the following theorem; we note that the algorithm relies on the standard ball-growing technique, and a standard approach for computing neighborhood covers. The proof of the theorem is deferred to the full version of the paper.

Theorem 7.1. *There is a deterministic algorithm for the Low-Diameter Clustering problem, such that, if m is the number of edges in the initial graph G, then the running time of the initialization, and the running time of each phase is bounded by $O(m^{1+1/k^3})$.*

8 Combining All Tools: the Proof of Theorem 3.1

In this section we complete the prooof of Theorem 3.1 by combining the tools presented in Sections 4–7.

In order to simplify the notation in this subsection, we call a graph G and parameters n, k, δ, Δ and Δ^* that satisfy the conditions of Theorem 3.1 (except for the constraint on vertex degrees in G) as "valid input graph and parameters". For completeness we define them below.

Definition 8.1. (Valid input graph and parameters) *Assume that we are given a parameter n, that is greater than a large enough constant, and additional parameters $512 \leq k \leq (\log n)^{1/49}$, $\frac{1}{k} \leq \delta \leq \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer, Δ^*, and Δ, such that $n^{18/k} \leq \Delta^* \leq \frac{\Delta}{n^{8/k^2}} \leq n$ holds. Lastly, assume that we are given a simple graph G with $|V(G)| \leq n$. We refer to a graph G and parameters $n, k, \delta, \Delta, \Delta^*$ that satisfy these constraints as a* valid input graph G with valid input parameters n, k, δ, Δ and Δ^*.

Throughout, we use a large enough constant c from from the definition of the EmbedOrScatter problem (see Definition 5.2). Given a valid input graph G with valid input parameters n, k, δ, Δ and Δ^*, we define additional

parameters $d^* = k^{10} \cdot 2^{c/\delta^6}$, $\eta^* = 2^{c/\delta^6} \cdot k^{4ck^2} \cdot n^{1/k^2}$, and $\hat{k} = k^2$. Note that, if n, k, δ, Δ and Δ^* are valid input parameters, then, since $k \leq (\log n)^{1/49}$ holds, we get that $k^{29} \leq \frac{\log n}{k^{20}}$, and so:

$$2^{k^{29}} \leq n^{1/k^{20}}. \tag{8.1}$$

Additionally, since n is sufficiently large, and $k^{49} \leq \log n$ holds, we can assume that $ck^{40} \leq c(\log n)^{40/49} \leq \log n$ holds, and so $2^{ck^{40}} \leq n$. Therefore:

$$2^{ck^{10}} \leq n^{1/k^{10}}. \tag{8.2}$$

Lastly, since $\delta \geq 1/k$, and from the definitions $d^* = k^{10} \cdot 2^{c/\delta^6}$ and $\eta^* = 2^{c/\delta^6} \cdot k^{4ck^2} \cdot n^{1/k^2}$, we get that:

$$\eta^* \cdot d^* = 2^{2c/\delta^6} \cdot k^{8ck^2} \cdot n^{1/k^2} \leq 2^{4ck^6} \cdot n^{1/k^2} \leq n^{2/k^2}. \tag{8.3}$$

The proof of Theorem 3.1 consits of two parts. In the first part, we provide an algorithm that maintains the collection $\mathcal{C}$ of edge-disjoint clusters of G, such that every cluster $C \in \mathcal{C}$ is a $(\overline{\deg}_C, \tilde{d}, \tilde{\eta})$-router with respect to the set $V(C)$ of supported vertices. The second part provides an algorithm that, for every cluster $C \in \mathcal{C}$, maintains a subgraph $C' \subseteq C$ with $V(C') = V(C)$, such that C' is a $(\overline{\Delta}^*, \tilde{d}, \tilde{\eta})$-router for the set $V(C)$ of supported vertices, and $|E(C')| \leq \Delta^* \cdot |V(C)|$.

8.1 Part 1: Maintaining the Clustering $\mathcal{C}$ In this subsection we provide an algorithm that maintains a collection $\mathcal{C}$ of edge-disjoint clusters of G, such that every cluster $C \in \mathcal{C}$ is a $(\overline{\deg}_C, d^*, \eta^*)$-router with respect to the set $V(C)$ of supported vertices. A central notion that we use in this part is a *router certificate*, that we define next.

Definition 8.2. (Router Certificate) *Let G be a valid input graph, and let n, k, δ, Δ and Δ^* be the corresponding valid input parameters. Let $d^* = k^{10} \cdot 2^{c/\delta^6}$, $\eta^* = 2^{c/\delta^6} \cdot k^{4ck^2} \cdot n^{1/k^2}$, and $\hat{k} = k^2$. For a given subgraph (cluster) $C \subseteq G$, a* router certificate *for C consists of:*

- *a graph W_C, that is a properly pruned subgraph of $W_{\hat{k}}^{N_C,\Delta}$ for some parameter $\frac{1}{2} \cdot |V(C)|^{1/\hat{k}-1/\hat{k}^2} \leq N_C \leq (2|V(C)|)^{1/\hat{k}}$, and*
- *an embedding $\mathcal{P}_C$ of W_C into C via paths of length at most d^* that cause congestion at most η^*, such that every edge $e \in E(C)$ lies on some path in $\mathcal{P}_C$, and every vertex $v \in V(C)$ participates in at least $\frac{\Delta}{n^{4/k^2}}$ paths in $\mathcal{P}_C$.*

Moreover, graph W_C must be obtained from $W_{\hat{k}}^{N_C,\Delta}$ by applying the algorithm from Theorem 4.2 for the NiceRouterPruning *problem to it, after a single phase Φ_0 of updates, consisting of at most $\frac{N_C^{\hat{k}} \cdot \Delta}{2\hat{k}^{4\hat{k}}}$ edge deletions.*

Next, we describe the initialization step, in which we construct an initial clustering $\mathcal{C}$, and, for every cluster $C \in \mathcal{C}$, we construct a router certificate $\left(W_{\hat{k}}^{N_C,\Delta}, W_C, \mathcal{P}_C\right)$. Intuitively, from Lemma 6.1, the existence of the router certificate for each such cluster C proves that C is a $(\overline{\deg}_C, \tilde{d}, \tilde{\eta})$-router, for appropriately chosen parameters $\tilde{d}$ and $\tilde{\eta}$. Later, we describe an algorithm that maintains the clusters in $\mathcal{C}$ as the graph G undergoes k batches of edge deletions. For each cluster $C \in \mathcal{C}$, the algorithm will also maintain a subgraph $\hat{W}_C \subseteq W_C$, such that, for every vertex v that remains in C, the number of the embedding paths paths in $\left\{P(e) \in \mathcal{P}_C \mid e \in E(\hat{W}_C)\right\}$ that contain v is sufficiently large. Again, from Lemma 6.1, this is sufficient in order to ensure that each cluster $C \in \mathcal{C}$ remains a $(\overline{\deg}_C, \tilde{d}, \tilde{\eta})$-router, for appropriately chosen parameters $\tilde{d}$ and $\tilde{\eta}$. We now provide the algorithm to initialize the clustering $\mathcal{C}$, and to compute an initial router certificate for every cluster $C \in \mathcal{C}$.

8.1.1 Initialization: Constructing the Clustering and Router Certificates The algorithm for the first part of the initialization is summarized in the following theorem, whose proof is deferred to the full version of the paper.

THEOREM 8.1. *There is a deterministic algorithm, that receives as input a valid input graph G together with valid input parameters n, k, δ, Δ and Δ^*, such that, for every vertex $v \in V(G)$, $\deg_G(v) \leq \Delta \cdot n^{16/k}$. The algorithm computes a collection $\mathcal{C}$ of subgraphs of G called* clusters *that are disjoint in their edges but may share vertices, with $\sum_{C \in \mathcal{C}} |V(C)| \leq n^{1+20/k}$. For every cluster $C \in \mathcal{C}$, it also computes a router certificate $\left(W_{k^2}^{N_C,\Delta}, W_C, \mathcal{P}_C\right)$. If we denote by $E^{\mathrm{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$, then the algorithm guarantees that $|E^{\mathrm{del}}| \leq |V(G)| \cdot \Delta \cdot n^{2/k^2}$. The running time of the algorithm is $O\left((n\Delta)^{1+O(\delta)}\right)$.*

Recall that, from the definition of a router certificate, for every cluster $C \in \mathcal{C}$, our algorithm initializes the algorithm from Theorem 4.2 for the NiceRouterPruning problem on graph $W_C = W_{\hat{k}}^{N_C,\Delta}$, that we denote by $\mathcal{A}_C$. During the phase Φ_0, Algorithm $\mathcal{A}_C$ performs an initial pruning of W_C, and the resulting graph serves as the router certificate for C. After the initialization step, as our algorithm receives batches $\pi_1, \ldots, \pi_k$ of edge deletions for graph G, we may delete additional edges from W_C, over the course of k phases $\Phi_1, \ldots, \Phi_k$, that correspond to the above batches of updates to G. We will employ the same algorithm $\mathcal{A}_C$, that was initialized as part of the algorithm in Theorem 8.1, in order to implement the additional pruning of graph W_C over the course of the phases $\Phi_1, \ldots, \Phi_k$.

8.1.2 Updating the Clusters and the Router Certificates In this subsection we provide an algorithm for maintaining the clustering $\mathcal{C}$. For every cluster $C \in \mathcal{C}$, for all $1 \leq i \leq k$, the algorithm updates the cluster C and the corresponding graph W_C following the batch $\pi_i^C = \pi_i \cap E(C)$ of edge deletions from C. The algorithm for updating a single cluster $C \in \mathcal{C}$ is summarized in the following theorem, whose proof is deferred to the full version of the paper. For simplicity, in the theorem, we denote π_i^C by π_i. We note that the theorem statement requires that $|\pi_i^C|$ is sufficiently small; whenever this is not the case, we will destroy the cluster completely, and move all its edges to E^{del}.

THEOREM 8.2. *There is a deterministic algorithm, whose input consists of:*

- *a valid input graph G together with valid input parameters n, k, δ, Δ and Δ^*, such that all vertex degrees in G are at most $\Delta \cdot n^{16/k}$;*
- *a subgraph $C \subseteq G$ called a* cluster*, together with a router certificate $\left(W_{\hat{k}}^{N_C,\Delta}, W_C, \mathcal{P}_C\right)$ for C;*
- *for every edge $e \in E(C)$, a list $L(e)$ of all edges $\hat{e} \in E(W_C)$, whose embedding path $P(\hat{e}) \in \mathcal{P}_C$ contains e; and*
- *for every vertex $v \in V(C)$, the number n_v of paths in $\mathcal{P}_C$ containing v.*

Recall that, from the definition of the router certiciate, graph W_C must be obtained from $W_{\hat{k}}^{N_C,\Delta}$ by applying the algorithm $\mathcal{A}_C$ from Theorem 4.2 for the NiceRouterPruning *problem to it, after a single phase Φ_0 of updates, consisting of at most $\frac{N_C^{\hat{k}} \cdot \Delta}{2\hat{k}^{4\hat{k}}}$ edge deletions. We assume that our algorithm is also given access to Algorithm $\mathcal{A}_C$, including its inner state, at the end of Phase Φ_0. The algorithm also receives as input k online batches $\pi_1, \ldots, \pi_k$ of edge deletions from C, where, for all $1 \leq i \leq k$, $|\pi_i| \leq \frac{|V(C)| \cdot \Delta}{n^{5/k^2}}$.*

The algorithm maintains, at all times, a graph $\hat{C} \subseteq C$ and a graph $\hat{W}_C \subseteq W_C$, that is a properly pruned subgraph of $W_{\hat{k}}^{N_C,\Delta}$. Initially, $\hat{C} = C$ and $\hat{W}_C = W_C$ hold, and the only changes that graphs $\hat{C}$ and $\hat{W}_C$ undergo is the deletion of some of their edges and vertices over time. Throughout, we denote by $\mathcal{P}_C = \left\{P(e) \mid e \in E(\hat{W}_C)\right\}$ the embedding of the current graph $\hat{W}_C$ into C, where the embedding paths $P(e)$ do not change over the course of the

algorithm. The algorithm guarantees that, at all times, for every edge $e \in E(\hat{W}_C)$, its embedding path $P(e) \in \mathcal{P}_C$ is contained in the current graph $\hat{C}$; for every edge $e' \in E(\hat{C})$, some path in $\mathcal{P}_C$ contains e; and for every vertex $v \in V(\hat{C})$, at least $\frac{\Delta}{n^{6/k^2}}$ paths in $\mathcal{P}_C$ contain v. Lastly, the algorithm ensures that, for all $1 \leq i \leq k$, at most $|\pi_i| \cdot n^{5/k^2}$ edges are deleted from $\hat{C}$, and at most $|\pi_i| \cdot n^{3/k^2}$ edges are deleted from $\hat{W}_C$ while processing batch π_i. For all $1 \leq i \leq k$, the time required to process batch π_i is at most $O\left(|\pi_i| \cdot n^{O(1/k^2)}\right)$.

8.1.3 Part 1: Summary The following corollary easily follows from Theorem 8.1 and Theorem 8.2, and it summarizes our algorithm for maintaining the clustering $\mathcal{C}$, and, for every cluster $C \in \mathcal{C}$, the corresponding graph W_C, along with its embedding into C. We defer the proof to the full version of the paper.

COROLLARY 8.1. *There is a deterministic algorithm, that receives as input a valid input graph G together with valid input parameters n, k, δ, Δ and Δ^*, such that, for every vertex $v \in V(G)$, $\deg_G(v) \leq \Delta \cdot n^{16/k}$. The algorithm also receives k online batches $\pi_1, \ldots, \pi_k$ of updates to graph G, where for $1 \leq i \leq k$, the ith batch π_i is a collection of at most $\Delta \cdot n$ edges that are deleted from G.*

The algorithm maintains a collection $\mathcal{C}$ of subgraphs of G called clusters*, with $\sum_{C \in \mathcal{C}} |V(C)| \leq n^{1+20/k}$, such that every cluster $C \in \mathcal{C}$ is a $\left(\overline{\deg}_C, \tilde{d}, \tilde{\eta}\right)$-router with respect to the set $V(C)$ of supported vertices, for $\tilde{d} \leq 2k^{18} \cdot 2^{c/\delta^6}$ and $\tilde{\eta} \leq n^{17/k}$, and every edge of G lies in at most one cluster $C \in \mathcal{C}$. The algorithm also maintains, for every cluster $C \in \mathcal{C}$, a graph W_C, that is a properly pruned subgraph of $W_{k^2}^{N_C,\Delta}$, for some parameter N_C, and an embedding $\mathcal{P}_C$ of W_C into C via paths of length at most d^* that cause congestion at most η^*, such that every vertex of C belongs to at least $\frac{\Delta}{n^{6/k^2}}$ of the paths in $\mathcal{P}_C$ at all times. After the clustering $\mathcal{C}$ is initialized, for every cluster $C \in \mathcal{C}$, the only changes that C and W_C may undergo are the deletions of some of their edges and vertices; for every edge e that remains in W_C, its embedding path $P(e) \in \mathcal{P}_C$ cannot change after the initialization. Moreover, for all $1 \leq i \leq k$, if we denote by $\pi_i^C = \pi_i \cap E(C)$, then the number of edges deleted from C while processing the batch π_i is at most $|\pi_i^C| \cdot n^{7/k^2}$. The only additional change that the clustering $\mathcal{C}$ may undergo after its initializaition is the deletion of empty clusters from $\mathcal{C}$.*

Throughout, we denote by $E^{\text{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$. At the beginning of the algorithm, after the initialiation, $|E^{\text{del}}| \leq |V(G)| \cdot \Delta \cdot n^{2/k^2}$ holds, and after that edges may only be added to E^{del}. For all $1 \leq i \leq k$, the number of edges added to E^{del} while processing batch π_i is bounded by $|\pi_i| \cdot n^{7/k^2}$.

The initialization time of the algorithm is $O\left((n\Delta)^{1+O(\delta)}\right)$, and for all $1 \leq i \leq k$, the time required to process the ith update batch π_i is bounded by $O\left(|\pi_i| \cdot n^{O(1/k^2)}\right)$.

8.2 Part 2 of the Algorithm: Maintaining the Subgraphs $C' \subseteq C$ In order to complete the proof of Theorem 3.1, it remains to provide an algorithm that maintains, for every cluster $C \in \mathcal{C}$, a subgraph $C' \subseteq C$ with $V(C') = V(C)$ and $|E(C')| \leq \Delta^* \cdot |V(C)|$, such that C' is a $\left(\overline{\Delta}^*, \tilde{d}, \tilde{\eta}\right)$-router for the set $V(C)$ of supported edges.

Throughout, we use the parameter $\Delta' = \left\lfloor \frac{\Delta^*}{2k^2 d^*} \right\rfloor$. Recall that the algorithm from Corollary 8.1 maintains, for every cluster $C \in \mathcal{C}$, a graph W_C, that is a properly pruned subgraph of $W_{k^2}^{N_C,\Delta}$, for some parameter N_C, and an embedding $\mathcal{P}_C$ of W_C into C via paths of length at most d^* that cause congestion at most η^*, such that every vertex of C belongs to at least $\frac{\Delta}{n^{6/k^2}}$ of the paths in $\mathcal{P}_C$ at all times. Notice that:

$$\Delta' = \left\lfloor \frac{\Delta^*}{2k^2 d^*} \right\rfloor \leq \frac{\Delta}{n^{8/k^2}} \leq \frac{\Delta}{2^k \cdot n^{6/k^2}},$$

from the definition of valid input parameters, and since $2^k \leq n^{1/k^2}$.

Consider now some cluster $C \in \mathcal{C}$. For every vertex $v \in V(C)$, let $\mathcal{P}_C(v) \subseteq \mathcal{P}_C$ denote the set of all embedding paths that contain v. Consider now some path $P \in \mathcal{P}_C(v)$, and recall that P must be an embedding path of some

edge $e \in E(W_C)$, so $P = P(e)$. Recall that one of the endpoints of e must be a leaf vertex of W_C; we call it the *distinguished endpoint of* P. We denote by $U_C(v)$ the multiset of leaf vertices of W_C that serve as distiguished endpoints of the paths in $\mathcal{P}_C(v)$. Lastly, consider the bipartite graph $R_C = (A, B, \tilde{E})$, where $A = V(C)$, and B contains all leaf vertices of W_C (so a vertex $v \in V(C)$ may lie in both A and B; we will refer to these vertices as "the copy of v in A" and "the copy of v in B".). For every vertex $v \in V(C)$, for each vertex u multiset $U_C(v)$, we include an edge connecting a copy of v in A to a copy of u in B in $\tilde{E}$. Recall that, for every vertex $u \in U_C(v)$, there is a distinct path $P \in \mathcal{P}_C(v)$, such that u is the distinguished endpoint of P. We will say that the new edge *represents* the path P.

Additional Data Structures. We fix again a cluster $C \in \mathcal{C}$. Throughout the algorithm, for every vertex $v \in V(C)$, we maintain the set $U_C(v)$ of leaf vertices of W_C explicitly. Additionally, we maintain the set $\mathcal{P}_C(v)$ of paths implicitly, as follows: for every vertex $u \in U_C(v)$, we maintain a pointer from the copy of u in $U_C(v)$ to the edge $e \in E(W_C)$, such that u serves as the distinguished endpoint of the path $P(e) \in \mathcal{P}_C(v)$ (if several copies of u lie in $U_C(v)$, then we associate each copy with a distinct path in $\mathcal{P}_C(v)$, and maintain a pointer from each copy of u in $U_C(v)$ to the corresponding edge of W_C). We also explicitly maintain the bipartite graph R_C.

Recall that our algorithm from Part 1 maintains, for every large cluster $C \in \mathcal{C}$ and vertex $v \in V(C)$, the counter $n_v = |\mathcal{P}_C(v)|$. The counter is maintained in a straightforward manner: during the initialization step, for every edge $e \in E(W_C)$, we inspect all vertices lying on path $P(e)$ and update their counters n_v. At this time, we can also construct the sets $U_C(v)$ of vertices for all $v \in V(C)$, together with the pointers from every vertex $u \in U_C(v)$ to the corresponding edge of W_C, without increasing the asymptotic running time of the initialization algorithm. We can also initialize the graph R_C within this running time.

During the update steps, whenever an edge e is deleted from graph W_C, we inspect every vertex $v \in V(P(e))$, and we decrease the corresponding counter n_v. During this time, we can also update the list $U_C(v)$ for each such vertex v with the deletion of this edge, and delete the edge connecting v to the distinguished endpoint of $P(e)$ from graph R_C. All this can be done without increasing the asymptotic running time of the algorithm for processing each batch π_i^C of updates to cluster C. The main result of the current subsection is summarized in the following claim, whose proof is deferred to the full version of the paper.

Claim 8.1. *There is a deterministic algorithm, that, given an access to the adjacency-list representation of the graph R_C, maintains, for every vertex $v \in V(C)$, a subset $U'_C(v) \subseteq U_C(v)$ of vertices, whose cardinality is between Δ' and $\Delta' \cdot n^{11/k^2}$ at all times, such that, for every leaf vertex x of W_C, the total number of times that x appears in the sets $\{U'_C(v) \mid v \in V(C)\}$ is at most $\gamma \cdot \Delta'$, where $\gamma = n^{16/k}$. The initialization time of the algorithm is $O(|V(C)| \cdot \Delta \cdot n^{O(1/k)})$, and, for all $1 \leq i \leq k$, the time required for updates following the processing of batch π_i^C of edge deletions to graph C is bounded by $O\left(|\pi_i^C| \cdot n^{O(1/k^2)}\right)$.*

We are now ready to comlete the proof of Theorem 3.1. We fix a cluster $C \in \mathcal{C}$, and provide an algorithm for maintaining the corresponding graph $C' \subseteq C$. We first define the graph C' and prove that it must be a $\left(\overline{\Delta}^*, \tilde{d}, \tilde{\eta}\right)$-router for the set $V(C)$ of supported edges, and that $|E(C')| \leq \Delta^* \cdot |V(C)|$. Later, we provide an algorithm for maintaining C'.

Definition of Graph C'. Recall that the algorithm from Claim 8.1 maintains, for every vertex $v \in V(C)$, a subset $U'_C(v) \subseteq U_C(v)$ of vertices, whose cardinality is between Δ' and $\Delta' \cdot n^{11/k^2}$ at all times. We let $U''_C(v) \subseteq U'_C(v)$ be an arbitray subset of Δ' such vertices, and we let $\mathcal{Q}_C(v) \subseteq \mathcal{P}_C(v)$ be the set of paths that correspond to these vertices, so that $|\mathcal{Q}_C(v)| = \Delta'$. Let $\mathcal{Q}_C = \bigcup_{v \in V(C)} \mathcal{Q}_C(v)$. Notice that, from Claim 8.1, every leaf vertex of W_C serves as an endpoint in at most $\gamma \cdot \Delta'$ paths in $\mathcal{Q}_C$. Additionally, for every superedge $e = (a, b)$ of W_C, we select an arbitrary collection $E'(a, b)$ of Δ' parallel edges (a, b) (since W_C remains a properly pruned subgraph of $W_{k^2}^{N_C, \Delta}$, the number of parallel edges (a, b) must be at least $\left\lceil \frac{\Delta}{2} \right\rceil \geq \Delta'$). We let $\mathcal{Q}'_C$ be the set of paths that contains, for every superedge (a, b) of W_C, for every edge $e \in E'(a, b)$, the embedding path $P(e) \in \mathcal{P}_C$. We let C'' be the multigraph, obtained by taking the union of all paths in set $\mathcal{Q}_C \cup \mathcal{Q}'_C$, and we let $C' \subseteq C$ be the corresponding simple graph, obtained from C'' by deleting parallel edges. Recall that, from the definition of valid input parameters, $\Delta^* \geq n^{18/k} \geq 2k^2 d^*$ (from Inequality 8.3) must hold. From Observation 6.1, we conclude that $|E(C')| \leq |V(C)| \cdot \Delta^*$, and that graph C' is a $(\overline{\Delta}^*, \tilde{d}, \tilde{\eta}')$-router with respect to the set $V(C')$ of supported vertices,

where:

$$\tilde{d} = 22d^* \cdot k^8 \leq k^{19} \cdot 2^{c/\delta^6},$$

since $d^* = k^{10} \cdot 2^{c/\delta^6}$; and

$$\tilde{\eta} = 8\gamma(d^*)^2 \cdot \eta^* \cdot k^{k^3} \leq n^{17/k},$$

since $\gamma = n^{16/k}$, $\eta^* \cdot d^* \leq n^{2/k^2}$ from Inequality 8.3; and $k^{k^3} \leq n^{1/k^2}$ from Inequality 8.1. Since $\sum_{C \in \mathcal{C}} |V(C)| \leq n^{1+20/k}$, we get that $\sum_{C \in \mathcal{C}} |E(C')| \leq \sum_{C \in \mathcal{C}} |V(C)| \cdot \Delta^* \leq n^{1+20/k} \cdot \Delta^*$, as required. It is now enough to provide an algorithm for maintaining the graph C'.

Maintaining the Graph C'. Recall that the algorithm from Claim 8.1 maintains, for every vertex $v \in V(C)$, a subset $U'_C(v) \subseteq U_C(v)$ of vertices, whose cardinality is between Δ' and $\Delta' \cdot n^{11/k^2}$ at all times, such that, for every leaf vertex x of W_C, the total number of times that x appears in the sets $\{U'_C(v) \mid v \in V(C)\}$ is at most $\gamma \cdot \Delta'$. Once the subset $U'_C(v)$ is initialized, the algorithm may modify it arbitrarily, that is, vertices may be both added to, and deleted from the set $U'_C(v)$. We assume w.l.o.g. that the algorithm maintains an ordered list $L_C(v)$ of all vertices in $U'_C(v)$, so that, when a new vertex is added to $U'_C(v)$, it is added to the end of the list. Throughout the algorithm, we then let $U''_C(v)$ be the collection of the first Δ' vertices that lie in list $L_C(v)$. As before, we denote by $\mathcal{Q}_C(v) \subseteq \mathcal{P}_C(v)$ all paths corresponding to the vertices of $U''_C(v)$, and we denote by $\mathcal{Q}_C = \bigcup_{v \in V(C)} \mathcal{Q}_C(v)$. For every superedge (u, v) of W_C, we also assume that the parallel edges of W_C corresponding to (u, v) are stored in an ordered list $L'_C(u, v)$. Recall that, as the algorithm progresses, edges may only be deleted from $L'_C(u, v)$. Throughout the algorithm, we will let $E'(u, v)$ be the set of the first Δ' edges in list $L'_C(u, v)$. As before, we denote by $\mathcal{Q}'_C$ the set of paths that contains, for each superedge (a, b) of W_C, for every edge $e \in E'(a, b)$, the embedding path $\mathcal{P}(e) \in \mathcal{P}_C$. As before, multigraph graph C'' is obtained by taking the union of all paths in $\mathcal{Q}_C \cup \mathcal{Q}'_C$ while keeping parallel edges, and graph C' is the simple graph obtained from C'' by removing parallel edges. We denote by $U^*_C = \bigcup_{v \in V(C)} U''_C(v)$, and by E^*_C the union of the sets $E'(a, b)$ of edges of W_C, for all superedges (a, b) of W_C.

At the beginning of the algorithm, once the data structures from Part 1 and from Claim 8.1 are initialized, we can compute, for every cluster $C \in \mathcal{C}$, the initial sets $U''_C(v)$ of vertices for all $v \in V(C)$, and sets $E'(a, b)$ of edges for each superedge (a, b) of the initial graph W_C, in time that is asymptotically bounded by the time required to initialize the data structures from Part 1 (that include graph W_C), and the data structures from Claim 8.1 (that include sets $U'_C(v)$ of vertices for all $v \in V(C)$). The time required for this computation is then bounded by:

$$T = O\left((n\Delta)^{1+O(\delta)} + \sum_{C \in \mathcal{C}} |V(C)| \cdot \Delta \cdot n^{O(1/k)}\right) \leq O\left((n\Delta)^{1+O(\delta+1/k)}\right) \leq O\left((n\Delta)^{1+O(\delta)}\right),$$

since $\sum_{C \in \mathcal{C}} |V(C)| \leq |V(G)|^{1+20/k}$ must hold, and $\delta \geq 1/k$. Since, for every cluster $C \in \mathcal{C}$, the length of every path in $\mathcal{P}_C$ is bounded by $d^* \leq n^{2/k^2}$ (from Inequality 8.3), the time required to compute the graphs C'' and C', for all $C \in \mathcal{C}$ is then bounded by:

$$O\left(d^* \cdot (|U^*(C)| + |E^*_C|)\right) \leq O\left(T \cdot d^*\right) \leq O\left((n\Delta)^{1+O(\delta+1/k)}\right) \leq O\left((n\Delta)^{1+O(\delta)}\right).$$

Therefore, the initialiation time of both Part 1 and Part 2 of our algorithm is bounded by $O\left((n\Delta)^{1+O(\delta)}\right)$.

Consider now some integer $1 \leq i \leq k$, and the batch π_i of edge deletions. Once the algorithm from Part 1 finishes processing the batch π_i, for some clusters $C \in \mathcal{C}$, some of the edges may be deleted from graph W_C. If (a, b) is a superedge of graph W_C, and if the number of parallel edges (a, b) deleted from W_C that lie in set $E'(a, b)$ is $m_{a,b}$, then we add $m_{a,b}$ new edges from the list $L'_C(a, b)$, to set $E'(a, b)$, so that $E'(a, b)$ contains the first Δ' edges in list $L'_C(a, b)$. For each edge e of the original set $E'(a, b)$ that was deleted from W_C, for every edge $e' = (x, y) \in E(P(e))$, we delete e' from C'', and, if no edges (x, y) remain in C'', then we delete edge (x, y) from

C' as well. Next, for every edge e that was just added to $E'(a,b)$, we consider every edge $e'=(x,y)\in P(e)$. We add e' to graph C'', and, if no other edges (x,y) currently lie in C'', then we add edge (x,y) to C' as well. It is easy to verify that, if T_i is the time that the algorithm from Part 1 spends on processing batch π_i, then all the above updates, for all clusters $C\in\mathcal{C}$, can be done in time $O(T_i\cdot d^*)\leq O(T_i\cdot n^{2/k^2})$.

Consider now a cluster $C\in\mathcal{C}$, and denote by $\pi_i^C=E(C)\cap\pi_i$. Once the algorithm from Claim 8.1 processes the batch π_i^C, we consider every vertex $v\in V(C)$, for which at least one vertex that lies in $U_C''(v)$ was deleted from $U_C'(v)$. Let $n_C(v)$ be the number of such vertices. First, for every vertex $u\in U_C''(v)$ that was deleted from $U_C'(v)$, we delete u from $U_C''(v)$ as well. We then consider the path Q of $\mathcal{Q}_C(v)$ that corresponds to vertex u. For every edge $e'=(x,y)$ of Q, we delete e' from C'', and, if no other edges (x,y) remain in C'', then we delete edge (x,y) from C' as well. Next, we add $n_C(v)$ vertices from list $L_C(v)$ to $U_C''(v)$, so that $U_C''(v)$ contains the first Δ' vertices from list $L_C(v)$. For each such newly added vertex u, we consider the path Q of $\mathcal{Q}_C(v)$ corresponding to u. For every edge $e'=(x,y)$ of Q, we add e' to C'', and, if no other edges (x,y) lie in C'', then we add edge (x,y) to C' as well. Note that the time required to perform these updates a cluster $C\in\mathcal{C}$ is bounded by:

$$O\left(\sum_{v\in V(C)}n_C(v)\cdot d^*\right).$$

For every cluster $C\in\mathcal{C}$, the algoritm from Claim 8.1 must spend at least $\Omega\left(\sum_{v\in V(C)}n_C(v)\right)$ time on updating the sets $U_C'(v)$ of vertices for all $v\in V(C)$. Therefore, if T_i^C denotes the time that the algorithm from Claim 8.1 spends on processing the batch π_i^C of updates, then the time required to update the sets $U_C''(v)$ of vertices for all $v\in V(C)$, and to update the graph C' accordingly, is bounded by $O\left(T_i^C\cdot d^*\right)\leq O\left(T_i^C\cdot n^{2/k^2}\right)$, from Inequality 8.3. Since, from Claim 8.1, for every cluster $C\in\mathcal{C}$, $T_i^C\leq O\left(|\pi_i^C|\cdot n^{O(1/k^2)}\right)$, and since the time required for the algorithm from Part 1 to process the batch π_i of updates is $O\left(|\pi_i|\cdot n^{O(1/k^2)}\right)$, we get that the total time required for processing the batch π_i of updates, for both parts of our algorithm, is bounded by:

$$O\left(|\pi_i|\cdot n^{O(1/k^2)}\right)+\sum_{C\in\mathcal{C}}O\left(|\pi_i^C|\cdot n^{O(1/k^2)}\right)\leq O\left(|\pi_i|\cdot n^{O(1/k^2)}\right).$$

It is easy to verify that the total number of all edges that were deleted from or inserted into the graphs in $\{C'\mid C\in\mathcal{C}\}$ is also bounded by $O\left(|\pi_i|\cdot n^{O(1/k^2)}\right)$.

9 Applications to (Fault-Tolerant) Graph Sparsification

Consider a simple graph G, and a router decomosition $\left(\mathcal{C},\{C'\}_{C\in\mathcal{C}}\right)$ of G with parameters $\Delta^*,\tilde{d},\tilde{\eta}$ and ρ (see Definition 3.1). Recall that the decomposition ensures that, for every cluster $C\in\mathcal{C}$, the corresponding subgraph $C'\subseteq C$ is a $(\overline{\Delta}^*,\tilde{d},\tilde{\eta})$-router, for the set $V(C)$ of supported vertices. Also recall that we have denoted by $E^{\text{del}}=E(G)\setminus\left(\bigcup_{C\in\mathcal{C}}E(C)\right)$ – the set of edges that do not lie in any cluster of the decomposition. Consider the graph H', whose vertex set is $V(G)$, and edge set is $E^{\text{del}}\cup\left(\bigcup_{C\in\mathcal{C}}E(C')\right)$, and recall that the algorithm from Theorem 1.3 maintains a graph H with $H'\subseteq H$. In this section, we explore various properties of the graph H', that imply that the graph H itself has many useful properties as well. This, in turn, implies our results for constructing and maintaining spanners, fault-tolerant spanners with respect to bounded-degree faults, low-congestion spanners, and connectivity certificates. We start with discussing general dynamic spanners and low-congestion spanners.

9.1 Dynamic Spanners and Low-Congestion Spanners

Dynamic Spanners. Let G be a simple n-vertex graph, and let $\mathcal{R}=(\mathcal{C},\{C'\}_{C\in\mathcal{C}})$ be a router decomposition of G with parameters $\Delta^*,\tilde{d},\tilde{\eta}$ and ρ. Consider the corresponding graph H' whose vertex set is $V(G)$, and edge set

is $E^{\text{del}} \cup \left(\bigcup_{C \in \mathcal{C}} E(C')\right)$, where $E^{\text{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$. In the following observation, whose proof is deferred to the full version of the paper, we show that H' is a $\tilde{d}$-spanner.

OBSERVATION 9.1. *The graph H' is $\tilde{d}$-spanner.*

By combining Theorem 1.3 with Observation 9.1, we obtain the following immediate corollary, whose proof is deferred to the full version of the paper.

COROLLARY 9.1. *There is a deterministic algorithm, whose input is a graph G with n vertices and no edges, that undergoes an online sequence of at most n^2 edge deletions and insertions, such that G remains a simple graph throughout the update sequence. Each edge inserted into G has an integral length $\ell(e) \in \{1, \ldots, L\}$. Additionally, the algorithm is given parameters $512 \leq k \leq (\log n)^{1/49}$ and $\frac{1}{k} \leq \delta \leq \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer. The algorithm maintains a subgraph $H \subseteq G$ with $V(H) = V(G)$ and $|E(H)| \leq O\left(n^{1+O(1/k)} \log L\right)$, such that H is a k'-spanner for G, for $k' = k^{19} \cdot 2^{O(1/\delta^6)}$. The worst-case update time of the algorithm is $n^{O(\delta)}$ per operation, and the number of edge insertions and deletions that graph H undergoes after each update to G is bounded by $n^{O(1/k)}$.*

Low-Congestion Spanners. We formalize the notion of *low-congestion spanners*, that was implicitly introduced by Chen et al. [CKL+22]. The new notion views a spanner H of G as a sparse subgraphs of G, with the additional property that G can be embedded into H with low congestion and path length bounds. Notice that, if we use the standard notion of a t-spanner H of G, then G can be embedded into H via paths of length at most t, but the congestion of the embedding may be as large as $|E(G)|$. Low-congestion spanners aim at optimizing the length and the congestion parameters of this embedding simultaneously. We now provide a formal definition of low-congestion spanners.

DEFINITION 9.1. (LOW-CONGESTION SPANNERS) *Let G be a graph, let $H \subseteq G$ be a subgraph of G with $V(H) = V(G)$, and let $d, \eta > 0$ be parameters. We say that H is a (d, η)-low congestion spanner for G, if there is an embedding $\mathcal{Q} = \{Q(e) \mid e \in E(G)\}$ of G into H, such that the length of every path in $\mathcal{R}$ is a most d, and the paths in $\mathcal{R}$ cause congestion at most η.*

We note that low-congestion spanners can also be defined with respect to vertex congestion, and it is the spanners of the latter kind that were considered by [vdBCK+23]. Their algorithm maintains such a spanner H for a dynamic n-vertex graph G, with $|E(H)| \leq O(n^{1+o(1)})$, together with the embedding $\mathcal{Q}$ of G into H, such that the lengths of the embedding paths are bounded by $d = n^{o(1)}$, and vertex-congestion of the embedding $\mathcal{Q}$ is bounded by $\eta = n^{o(1)} \cdot \max_{v \in V(G)} \{\deg_G(v)\}$. The amortized update time and recourse bounds are $n^{o(1)}$.

We use the following simple lemma, whose proof is deferred to the full version of the paper.

LEMMA 9.1. *Let G be an n-vertex graph, and let $\mathcal{R} = (\mathcal{C}, \{C'\}_{C \in \mathcal{C}})$ be a router decomposition of G with parameters $\Delta^*, \tilde{d}, \tilde{\eta}$ and ρ. Then for every cluster $C \in \mathcal{C}$, there is an embedding $\mathcal{Q}_C$ of C into C' via paths of length at most $\tilde{d}$, that cause congestion at most $\eta = \max\left\{\frac{16\tilde{\eta} \cdot \Delta_{\max}(C)}{\Delta^*}, 16 \log n\right\}$, where $\Delta_{\max}(C)$ is the maximum vertex degree in C.*

We obtain the following immediate corollary of Lemma 9.1, whose proof is deferred to the full version of the paper.

COROLLARY 9.2. *Let G be an n-vertex graph, and let $\mathcal{R} = (\mathcal{C}, \{C'\}_{C \in \mathcal{C}})$ be a router decomposition of G with parameters $\Delta^*, \tilde{d}, \tilde{\eta}$ and ρ. Consider the corresponding graph H' whose vertex set is $V(G)$, and edge set is $E^{\text{del}} \cup \left(\bigcup_{C \in \mathcal{C}} E(C')\right)$, where $E^{\text{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$. Then graph H' is an $(\tilde{d}, \eta)$-low congestion spanner for G, where $\eta = \max\left\{\frac{16\tilde{\eta} \cdot \Delta_{\max}(G)}{\Delta^*}, 16 \log n\right\}$, and $\Delta_{\max}(G)$ is the maximum vertex degree in G.*

Finally, by combining Theorem 1.3 with Corollary 9.2, we obtain the following immediate corollary; the proof is essentially identical to the proof of Corollary 9.1 and is omitted here.

COROLLARY 9.3. *There is a deterministic algorithm, whose input is a graph G with n vertices and no edges, that undergoes an online sequence of at most n^2 edge deletions and insertions, such that G remains a simple graph throughout the update sequence. Each edge inserted into G has an integral length $\ell(e) \in \{1, \ldots, L\}$. Additionally, the algorithm is given parameters $512 \leq k \leq (\log n)^{1/49}$, $\frac{1}{k} \leq \delta \leq \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer, and $\Delta \geq n^{20/k}$. The algorithm maintains a subgraph $H \subseteq G$ with $V(H) = V(G)$ and $|E(H)| \leq O\left(n^{1+O(1/k)} \cdot \Delta \cdot \log L\right)$, such that H is a (k', η)-low-congestion spanner for G, for $k' = k^{19} \cdot 2^{O(1/\delta^6)}$ and $\eta \leq n^{O(1/k)} \cdot \frac{\Delta_{\max}(G)}{\Delta}$, where $\Delta_{\max}(G)$ is the maximum degree in G. The worst-case update time of the algorithm is $n^{O(\delta)}$ per operation, and the number of edge insertions and deletions that graph H undergoes after each update to G is bounded by $n^{O(1/k)}$.*

9.2 Dynamic Fault-Tolerant Spanners and Connectivity Certificates In this subsection we provide a key observation that routers with a sufficiently large minimum degree are inherently resilient to edge failures. We then show that, by setting the parameter Δ in Theorem 1.3 appropriately, we can ensure that the resulting graph H is resilient to bounded-degree faults. Lastly, we prove that H can be used as a connectivity certificate.

9.2.1 Resilience of Routers to Bounded-Degree Faults Let G be a graph, and let F be a subset of edges of G (edge faults). We denote by $\deg(F)$ the maximum, over all vertices of v, of the number of edges in F that are incident to v. We say that a set F of edges is an f-BD faulty set, if $\deg(F) \leq f$. Recall that a graph G is a $(\overline{\Delta}, d, \eta)$-router for the set $V(G)$ of supported vertices, if every Δ-restricted demand $\mathcal{D}$ over $V(G)$ can be routed with congestion at most η in G, via paths of length at most d. In this subsection we show that routers are resilient to bounded-degree faults, in the sense that $G \setminus F$ remains a router, albeit with slightly weaker parameters. The main result of this subsection is summarized in the following theorem, whose proof is deferred to the full version of the paper.

THEOREM 9.1. *Let G be an n-vertex graph, and let $f, k, d > 0$, $1 \leq \eta < n^2$ and $\Delta \geq 32f \cdot n^{1/k} \cdot \eta$ be parameters. Assume further that G is a $(\overline{\Delta}, d, \eta)$-router with respect to the set $V(G)$ of supported vertices. Then for every f-BD faulty set $F \subseteq E(G)$ of edges, graph $G \setminus F$ is a $(\overline{\Delta}, O(k) \cdot d, O(k) \cdot \eta)$-router for the set $V(G)$ of supported vertices.*

The above theorem should be compared to the resilience of length-constrained expanders against f-BD faults, that was established in [BDR22]. Specifically, they show in Theorem 1.2 of [BDR22] that, if G is an n-vertex (h, s)-length φ-expander with minimum degree $\Omega(fn^{\epsilon}/\varphi)$, then, for any f-BD faulty set $F \subseteq E(G)$ of edges, the graph $G \setminus F$ is an $(hs)^{1/\epsilon}$-spanner of G; in other words, the distances in $G \setminus F$ may grow by at most factor $(hs)^{1/\epsilon}$. It then follows that the f-FD spanners obtained by [BDR22] via LC-expanders have size $\widetilde{O}(f \cdot n^{1+1/k})$ and stretch $t = k^{O(k)}$. The question of improving the exponential dependence of the stretch on k, while preserving the size of the spanner was left open in [BDR22] (see Theorem 1.14 therein), and this improvement would immediately imply stronger bounds for f-FD spanners. We resolve this open problem in Theorem 9.1, for the case of where the LC-expander has a bounded diameter (that is, it is a router).

9.2.2 From Router Decomposition to Fault Tolerant Spanners

Fault-Tolerant Spanners. Recall that for a graph G, a stretch parameter $k \geq 1$, and an integer fault parameter $f \geq 1$, an f-FT k-spanner of G is a subgraph $H \subseteq G$ satisfying that, for every subset $F \subseteq E(G)$ of at most f edges, $H \setminus F$ is a k-spanner for $G \setminus F$.

Very recently, Bodwin, Haeupler and Parter [BHP24] introduced a considerably stronger notion of fault-tolerance, where only the degree of the faulty edge set, rather than its cardinality, is restricted. For a subset $F \subseteq E(G)$ of faulty edges, its *faulty-degree* $\deg(F)$ is the maximum, over all vertices v, of the number of edges of F incident to v. For example, if F is a matching, then $\deg(F) = 1$, while $|F|$ may be as large as $n/2$. This naturally leads to the definition of faulty-degree spanners.

Definition 9.2. (Faulty-Degree Spanners) *[BHP24] Let G be a graph and let $H \subseteq G$ be a subgraph of G with $V(H) = V(G)$. For parameters f and $k > 0$, we say that H is an f-faulty-degree (FD) k-spanner for G if, for every subset $F \subseteq E(G)$ of edges with $\deg(F) \leq f$, for every pair u, v of vertices of G, $\mathsf{dist}_{H \setminus F}(u, v) \leq k \cdot \mathsf{dist}_{G \setminus F}(u, v)$ holds.*

In the following observation, whose proof is deferred to the full version of the paper, we show that, if H' is a graph obtained from a router decomposition of the graph G as before, then it is an f-FD k'-spanner, for appropriately chosen parameters f and k'.

Observation 9.2. (From Router Decomposition to f-FD Spanners) *Let G be an n-vertex graph, and let $k' > 0$ and $f \in \{1, \ldots, n\}$ be parameters. Let $\mathcal{R} = (\mathcal{C}, \{C'\}_{C \in \mathcal{C}})$ be a router decomposition of G with parameters $\tilde{d}$, $1 \leq \tilde{\eta} \leq n^2$, $\Delta^* \geq 32 \cdot f \cdot \tilde{\eta} \cdot n^{1/k'}$, and ρ. Consider the corresponding graph H' whose vertex set is $V(G)$, and edge set is $E^{\mathrm{del}} \cup \left(\bigcup_{C \in \mathcal{C}} E(C')\right)$, where $E^{\mathrm{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$. Then H' is an f-FD $O(\tilde{d} \cdot k')$-spanner.*

Fault-Tolerant Low-Congestion Spanners. Finally, we show the strongest property of subgraph H maintained by the algorithm from Theorem 1.3. We will use the fact that routers retain their routing properties (with a small loss), to show that H is a fault-tolerant low-congestion spanner even against bounded degree faults. We start by formally defining the latter notion.

Definition 9.3. (Fault-Tolerant Low-Congestion Spanners) *Let G be a graph, let $H \subseteq G$ be a subgraph of G, and let f, d and η be parameters. We say that H is an f-FT (d, η)-low congestion spanner if, for every subset $F \subseteq E(G)$ of edges with $|F| \leq f$, graph $G \setminus F$ embeds into $H \setminus F$ via paths of length at most d, with congestion at most η. Similarly, H is an f-FD (d, η)-low congestion spanner if the above holds for every subset $F \subseteq E(G)$ of edges with $\deg(F) \leq f$.*

In the following observation, whose proof is deferred to the full version of the paper, we show that the graph H' obtained from a router decomposition of G with an appropriately chosen parameter f must be an f-FD low-congestion spanner.

Observation 9.3. (From Router Decomposition to f-FD Low-Congestion Spanners) *Let G be an n-vertex graph, and let $k' > 0$ and $f \in \{1, \ldots, n\}$ be parameters. Let $\mathcal{R} = (\mathcal{C}, \{C'\}_{C \in \mathcal{C}})$ be a router decomposition of G with parameters $\tilde{d}$, $1 \leq \tilde{\eta} \leq n^2$, $\Delta^* \geq 32 \cdot f \cdot \tilde{\eta} \cdot n^{1/k'}$, and ρ. Consider the corresponding graph H' whose vertex set is $V(G)$, and edge set is $E^{\mathrm{del}} \cup \left(\bigcup_{C \in \mathcal{C}} E(C')\right)$, where $E^{\mathrm{del}} = E(G) \setminus \left(\bigcup_{C \in \mathcal{C}} E(C)\right)$. Then H' is an f-FD (d, η)-low congestion spanner for $d = O(k' \cdot \tilde{d})$ and $\eta = \max\left\{O\left(\frac{k' \cdot \tilde{\eta} \cdot \Delta_{\max}(G)}{\Delta^*}\right), 16 \log n\right\}$, where $\Delta_{\max}(G)$ is maximum vertex degree in G.*

Finally, we obtain the following analogue of Corollaries 9.1 and 9.3 for f-FD low-congestion spanners.

Corollary 9.4. *There is a deterministic algorithm, whose input is a graph G with n vertices and no edges, that undergoes an online sequence of at most n^2 edge deletions and insertions, such that G remains a simple graph throughout the update sequence. Each edge inserted into G has an integral length $\ell(e) \in \{1, \ldots, L\}$. Additionally, the algorithm is given parameters $512 \leq k \leq (\log n)^{1/49}$, $\frac{1}{k} \leq \delta \leq \frac{1}{400}$ such that $\frac{1}{\delta}$ is an integer, $f \in \{1, \ldots, n\}$, and $\Delta \geq f \cdot n^{c/k}$, for a large enough constant c. The algorithm maintains a subgraph $H \subseteq G$ with $V(H) = V(G)$ and $|E(H)| \leq O\left(n^{1+O(1/k)} \cdot \Delta \cdot \log L\right)$, such that H is an a f-FD (d, η)-low congestion spanner for G, for $d \leq \mathrm{poly}(k) \cdot 2^{O(1/\delta^6)}$ and $\eta \leq n^{O(1/k)} \cdot \frac{\Delta_{\max}(G)}{\Delta}$, where $\Delta_{\max}(G)$ is the maximum degree in G. The worst-case update time of the algorithm is $n^{O(\delta)}$ per operation, and the number of edge insertions and deletions that graph H undergoes after each update to G is bounded by $n^{O(1/k)}$.*

In order to prove the corollary, we use the algorithm from Corollary 9.3. By using Observation 9.3 with parameter k' replaced with k, it is easy to verify that the graph H that the algorithm maintains has all required properties.

Dynamic Connectivity Certificates. The notion of connectivity certificates was introduced by [NI92]. For a graph $G = (V, E)$ and integer $f \in \{1, \ldots, n\}$, an f-connectivity certificate is a subgraph $H \subseteq G$ with the following property: for every pair $u, v \in V$ of vertices, for every subset $F \subseteq E$ of at most f edges, u and v are connected in $H \setminus F$ if and only if they are connected in $G \setminus F$. The recent work of [BHP24] introduced a stronger notion of f-FD connectivity certificate: a subgraph $H \subseteq G$ that satisfies the above requirement for any subset $F \subseteq E$ of edges $\deg(F) \leq f$. It is well known that any n-vertex graph admits an f-connectivity certificate containing $O(fn)$ edges, and such certificates can be computed in linear time in the static setting. Using expander decompositions and expander routing, [BHP24] presented an algorithm that constructs an f-FD connectivity certificates with $\widetilde{O}(fn)$ edges, which is nearly tight, in polynomial time.

Clearly, if a graph H is an f-FD (d, η)-low congestion spanner for G, for any parameters d and η, then H is an f-FD connectivity certificate an and f-connectivity certificate for G as well. Therefore, by using the algorithm from Corollary 9.4 with a parameter $\Delta = f \cdot n^{O(1/k)}$ and $\delta = \frac{1}{k}$, we get that the graph H that the algorithm maintains has $|E(H)| \leq O\left(f \cdot n^{1+O(1/k)}\right)$, and it is an f-FD connectivity certificate for G. The algorithm has $n^{O(1/k)}$ worst-case update time and $n^{O(1/k)}$ recourse. Note that both of the latter quantities are independent of f. To the best of our knowledge, all previous dynamic algorithms for (the classical notion of) f-connectivity certificates had a worst-case update time with a polynomial dependency on f. Therefore, for high values of the parameter f, we obtain a faster algorithm.

References

[ACK+16] Alexandr Andoni, Jiecao Chen, Robert Krauthgamer, Bo Qin, David P Woodruff, and Qin Zhang. On sketching quadratic forms. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 311–319, 2016.

[ADF+07] Giorgio Ausiello, Camil Demetrescu, Paolo Giulio Franciosa, Giuseppe F. Italiano, and Andrea Ribichini. Small stretch spanners in the streaming model: New algorithms and experiments. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, volume 4698 of *Lecture Notes in Computer Science*, pages 605–617. Springer, 2007.

[AOSS19] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1919–1936. SIAM, 2019.

[AS23] Sepehr Assadi and Vihan Shah. Tight bounds for vertex connectivity in dynamic streams. In Telikepalli Kavitha and Kurt Mehlhorn, editors, *2023 Symposium on Simplicity in Algorithms, SOSA 2023, Florence, Italy, January 23-25, 2023*, pages 213–227. SIAM, 2023.

[Bas06] Surender Baswana. Dynamic algorithms for graph spanners. In Yossi Azar and Thomas Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2006.

[Bas08] Surender Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Inf. Process. Lett.*, 106(3):110–114, 2008.

[BC18] Aaron Bernstein and Shiri Chechik. Incremental topological sort and cycle detection in expected total time. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 21–34. SIAM, 2018.

[BDPW18] Greg Bodwin, Michael Dinitz, Merav Parter, and Virginia Vassilevska Williams. Optimal vertex fault tolerant spanners (for fixed stretch). In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1884–1900. SIAM, 2018.

[BDR22] Greg Bodwin, Michael Dinitz, and Caleb Robelle. Partially optimal edge fault-tolerant spanners. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 3272–3286. SIAM, 2022.

[BFH19] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1899–1918. SIAM, 2019.

[BHP24] Greg Bodwin, Bernhard Haeupler, and Merav Parter. Fault-tolerant spanners against bounded-degree edge failures: Linearly more faults, almost for free. In David P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM*

Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024, pages 2609–2642. SIAM, 2024.

[BK06] Surender Baswana and Telikepalli Kavitha. Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 591–602. IEEE Computer Society, 2006.

[BK16] Greg Bodwin and Sebastian Krinninger. Fully dynamic spanners with worst-case update time. In Piotr Sankowski and Christos D. Zaroliagis, editors, *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012.

[BS07] Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007.

[BSS22] Sayan Bhattacharya, Thatchaphol Saranurak, and Pattara Sukprasert. Simple dynamic spanners with near-optimal recourse against an adaptive adversary. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[BvdBG+22] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 20:1–20:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[CGL+20] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1158–1167. IEEE, 2020.

[CGP+20] Timothy Chu, Yu Gao, Richard Peng, Sushant Sachdeva, Saurabh Sawlani, and Junxing Wang. Graph sparsification, spectral sketches, and faster resistance computation via short cycle decompositions. *SIAM Journal on Computing*, 52(6):FOCS18–85, 2020.

[CHK16] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 217–226. ACM, 2016.

[Chu23] Julia Chuzhoy. A distanced matching game, decremental apsp in expanders, and faster deterministic algorithms for graph cut problems. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2122–2213. SIAM, 2023.

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 612–623. IEEE, 2022.

[CKP+17] Michael B. Cohen, Jonathan A. Kelner, John Peebles, Richard Peng, Anup B. Rao, Aaron Sidford, and Adrian Vladu. Almost-linear-time algorithms for markov chains and new spectral primitives for directed graphs. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 410–419. ACM, 2017.

[CKT93] Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. Scan-first search and sparse certificates: An improved parallel algorithms for k-vertex connectivity. *SIAM J. Comput.*, 22(1):157–174, 1993.

[CLPR10] Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. Fault tolerant spanners for general graphs. *SIAM J. Comput.*, 39(7):3403–3423, 2010.

[CZ19] Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 370–381. IEEE Computer Society, 2019.

[CZ23] Julia Chuzhoy and Ruimin Zhang. A new deterministic algorithm for fully dynamic all-pairs shortest paths. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 1159–1172. ACM, 2023.

[DFKL21] Michal Dory, Orr Fischer, Seri Khoury, and Dean Leitersdorf. Constant-round spanners and shortest paths in congested clique and MPC. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 223–233. ACM, 2021.

[DGPV08] Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 273–282. ACM, 2008.

[DHNS19] Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed edge connectivity in sublinear time. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 343–354. ACM, 2019.

[Elk07] Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 716–727. Springer, 2007.

[EP01] Michael Elkin and David Peleg. Approximating k-spanner problems for k>2. In Karen I. Aardal and Bert Gerards, editors, *Integer Programming and Combinatorial Optimization, 8th International IPCO Conference, Utrecht, The Netherlands, June 13-15, 2001, Proceedings*, volume 2081 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2001.

[Erd64] P. Erdös. Extremal problems in graph theory. *Theory of Graphs and its Applications (Proc. Symp. Smolenice, 1963)*, pages 29–36, 1964.

[EZ04] Michael Elkin and Jian Zhang. Efficient algorithms for constructing (1+, varepsilon;, beta)-spanners in the distributed and streaming models. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 160–168. ACM, 2004.

[FG19] Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 377–388. ACM, 2019.

[FNY+20] Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2046–2065. SIAM, 2020.

[GHN+23] Gramoz Goranci, Monika Henzinger, Danupon Nanongkai, Thatchaphol Saranurak, Mikkel Thorup, and Christian Wulff-Nilsen. Fully dynamic exact edge connectivity in sublinear time. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 70–86. SIAM, 2023.

[GK13] Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[GK18] Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 29:1–29:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[GKS17] Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed MST and routing in almost mixing time. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 131–140. ACM, 2017.

[GL18] Mohsen Ghaffari and Jason Li. New distributed algorithms in almost mixing time via transformations from parallel algorithms. *arXiv preprint arXiv:1805.04764*, 2018.

[GLN+19] Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Deterministic graph cuts in subquadratic time: Sparse, balanced, and k-vertex. *arXiv preprint arXiv:1910.07950*, 2019.

[HHG22] Bernhard Haeupler, Jonas Huebotter, and Mohsen Ghaffari. A cut-matching game for constant-hop expanders. *arXiv preprint arXiv:2211.11726*, 2022.

[HHL+24] Bernhard Haeupler, D. Ellis Hershkowitz, Jason Li, Antti Roeyskoe, and Thatchaphol Saranurak. Low-step multi-commodity flow emulators. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 71–82. ACM, 2024.

[HHT23] Bernhard Haeupler, D. Ellis Hershkowitz, and Zihan Tan. Parallel greedy spanners. *CoRR*, abs/2304.08892, 2023.

[HHT24] Bernhard Haeupler, D Ellis Hershkowitz, and Zihan Tan. New structures and algorithms for length-constrained expander decompositions. *CoRR*, abs/2404.13446, 2024. FOCS 2024, to appear.

[HLS24] Bernhard Haeupler, Yaowei Long, and Thatchaphol Saranurak. Dynamic deterministic constant-approximate distance oracles with n^{ϵ} worst-case update time. *CoRR*, abs/2402.18541, 2024. FOCS 2024, to appear.

[HRG22] Bernhard Haeupler, Harald Räcke, and Mohsen Ghaffari. Hop-constrained expander decompositions, oblivious routing, and distributed universal optimality. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory*

of Computing, pages 1325–1338, 2022.

[JS18] Arun Jambulapati and Aaron Sidford. Efficient õ (n/epsilon) spectral sketches for the laplacian and its pseudoinverse. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2487–2503. SIAM, 2018.

[JS22] Wenyu Jin and Xiaorui Sun. Fully dynamic st edge connectivity in subpolynomial time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 861–872. IEEE, 2022.

[KLOS14] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 217–226, 2014.

[KM97] David R. Karger and Rajeev Motwani. An NC algorithm for minimum cuts. *SIAM J. Comput.*, 26(1):255–272, 1997.

[KMPG24] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. A dynamic shortest paths toolbox: Low-congestion vertex sparsifiers and their applications. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 1174–1183, 2024.

[KP12] Michael Kapralov and Rina Panigrahy. Spectral sparsification via random spanners. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 393–398. ACM, 2012.

[LNP+21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 317–329. ACM, 2021.

[LNS98] Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 186–195. ACM, 1998.

[LR99] F. T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46:787–832, 1999.

[Mat93] David W. Matula. A linear time 2+epsilon approximation algorithm for edge connectivity. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 500–504. ACM/SIAM, 1993.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o (n1/2-ε)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1122–1129, 2017.

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 950–961, 2017.

[NSWN17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 950–961. IEEE, 2017.

[Par22] Merav Parter. Nearly optimal vertex fault-tolerant spanners in optimal time: sequential, distributed, and parallel. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1080–1092. ACM, 2022.

[PS89] David Peleg and Alejandro A. Schäffer. Graph spanners. *J. Graph Theory*, 13(1):99–116, 1989.

[PU87] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In Fred B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 77–85. ACM, 1987.

[PU89] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[RTZ05] Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 261–272. Springer, 2005.

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 81–90, 2004.

[SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2616–2635. SIAM, 2019.

[TK00] Mikkel Thorup and David R. Karger. Dynamic graph algorithms with applications. In Magnús M. Halldórsson, editor, *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, volume 1851 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2000.

[TZ01] M. Thorup and U. Zwick. Approximate distance oracles. *Annual ACM Symposium on Theory of Computing*, 2001.

[vdBCK+23] Jan van den Brand, Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *64th IEEE Annual Symposium on Foundations of Computer Science (FOCS), (to appear)*, 2023.

[WN17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1130–1143. ACM, 2017. Full version at arXiv:1611.02864.

[Wul17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143. ACM, 2017.