Better Decremental and Fully Dynamic Sensitivity Oracles for Subgraph Connectivity

Yaowei Long
University of Michigan, Ann Arbor, MI, USA

Yunfan Wang
Tsinghua University, Beijing, China

Abstract

We study the *sensitivity oracles problem for subgraph connectivity* in the *decremental* and *fully dynamic* settings. In the fully dynamic setting, we preprocess an n-vertices m-edges undirected graph G with n_{off} deactivated vertices initially and the others are activated. Then we receive a single update $D \subseteq V(G)$ of size $|D| = d \le d_\star$, representing vertices whose states will be switched. Finally, we get a sequence of queries, each of which asks the connectivity of two given vertices u and v in the activated subgraph. The decremental setting is a special case when there is no deactivated vertex initially, and it is also known as the *vertex-failure connectivity oracles* problem.

We present a better deterministic vertex-failure connectivity oracle with $\widehat{O}(d_\star m)$ preprocessing time, $\widetilde{O}(m)$ space, $\widetilde{O}(d^2)$ update time and $O(d)$ query time, which improves the update time of the previous almost-optimal oracle [14] from $\widehat{O}(d^2)$ to $\widetilde{O}(d^2)$.

We also present a better deterministic fully dynamic sensitivity oracle for subgraph connectivity with $\widehat{O}(\min\{m(n_{\text{off}} + d_\star), n^\omega\})$ preprocessing time, $\widetilde{O}(\min\{m(n_{\text{off}} + d_\star), n^2\})$ space, $\widetilde{O}(d^2)$ update time and $O(d)$ query time, which significantly improves the update time of the state of the art [9] from $\widetilde{O}(d^4)$ to $\widetilde{O}(d^2)$. Furthermore, our solution is even almost-optimal assuming popular fine-grained complexity conjectures.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases connectivity, sensitivity

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.109

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2402.09150

Acknowledgements We thank Thatchaphol Saranurak for helpful discussions.

1 Introduction

We study the *sensitivity oracles problem for subgraph connectivity* in the *decremental* and *fully dynamic* settings, which is one of the fundamental dynamic graph problems in undirected graphs. In the fully dynamic setting, this problem has three phases. In the preprocessing phase, given an integer $d_\star$, we preprocess an n-vertices m-edges undirected graph $G = (V, E)$ in which some vertices are *activated*, called *on-vertices* and denoted by V_{on}, while the others are *deactivated*, called *off-vertices* and denoted by V_{off}. We let $n_{\text{on}} = |V_{\text{on}}|$ and $n_{\text{off}} = |V_{\text{off}}|$ denote the number of initial on-vertices and off-vertices respectively. In the update phase, we will receive a set $D \subseteq V$ with $|D| = d \le d_\star$, representing vertices whose states will be switched, and we update the oracle. In the subsequent query phase, let $V_{\text{new}} = (V_{\text{on}} \setminus D) \cup (V_{\text{off}} \cap D)$ denote the activated vertices after the update. Each query will give a pair of vertices $u, v \in V_{\text{new}}$ and ask the connectivity of u and v in the new activated subgraph $G[V_{\text{new}}]$. The decremental setting is a special case where there is no off-vertices initially, i.e. V_{off} is empty.

The decremental version of this problem is also called the *vertex-failure connectivity oracles* problem, which has been studied extensively, e.g. [4, 5, 18, 17, 14, 13], and its complexity was well-understood up to subpolynomial factors. Specifically, a line of works by

Duan and Pettie [4, 5] started the study on vertex-failure connectivity oracles for general $d_\star$, and they finally gave a deterministic oracle with $\widetilde{O}(mn)$ preprocessing time, $\widetilde{O}(md_\star)$ space, $\widetilde{O}(d^3)$ update time and $O(d)$ query time. Following [4, 5], Long and Saranurak [14] presented an improved solution with $\widehat{O}(m) + \widetilde{O}(md_\star)$ preprocessing time, $\widetilde{O}(m)$ space, $\widehat{O}(d^2)$ update time and $O(d)$ query time[1], which is optimal up to subpolynomial factors because matching (conditional) lower bounds for all the four complexity measurements were shown in [7, 5, 14]. We refer to Table 1 for more solutions to this problem (for example, Kosinas [13] proposed a simple and practical algorithm using a conceptually different approach). However, there are still relatively large subpolynomial overheads on the current almost-optimal upper bounds (i.e., on the preprocessing time and update time of the LS-oracle), so a natural question is whether we can improve them:

Can we design an almost-optimal deterministic vertex-failure connectivity oracle with only polylogarithmic overheads on the update time, or even on all complexity bounds?

The fully dynamic sensitivity oracles problem for subgraph connectivity was studied by [8, 9]. Henzinger and Neumann [8] showed a black-box reduction from the decremental setting. Plugging in the almost-optimal decremental algorithm of [14], this reduction leads to a fully dynamic sensitivity oracle with $\widehat{O}(n_{\text{off}}^2 m) + \widetilde{O}(d_\star n_{\text{off}}^2 m)$ preprocessing time, $\widetilde{O}(n_{\text{off}}^2 m)$ space, $\widehat{O}(d^4)$ update time and $O(d^2)$. Hu, Kosinas and Polak [9] studied this problem from an equivalent but different perspective called *connectivity oracles for predictable vertex failures*, which gave a solution with $\widetilde{O}((n_{\text{off}} + d_\star)m)$ preprocessing time, $\widetilde{O}((n_{\text{off}} + d_\star)m)$ space, $\widetilde{O}(d^4)$ update time and $O(d)$ query time[2]. Despite the efforts, there are still gaps between the upper bounds of the fully dynamic setting and the decremental setting (except the query time). Naturally, one may have the following question:

Can we match the upper bounds in the fully dynamic and decremental settings or show separations between them for all the four measurements?

Notably, [9] showed a conditional lower bound $\widehat{\Omega}((n_{\text{off}} + d_\star)m)$ on the preprocessing time[3], which separated two settings at the preprocessing time aspect. However, the right complexity bounds are still not clear for the space and the update time. In particular, given that two different approaches [8, 9] both showed upper bounds around d^4 for update time, it is interesting to identify if this is indeed a barrier or it just happened accidentally. Furthermore, we note that it seems hard to improve the update time following either of these two approaches, because the black-box reduction by [8] has been plugged in the almost-optimal decremental oracle, and the fully dynamic oracle by [9] generalizes the decremental oracle by [13], where the latter already has $\widetilde{O}(d^4)$ update time.

1.1 Our Results

We give a partially affirmative answer to the first question and answer the second question affirmatively by the following results.

[1] Throughout the paper, we use $\widetilde{O}(\cdot)$ to hide a polylog(n) factor and use $\widehat{O}(\cdot)$ to hide a $n^{o(1)}$ factor.
[2] In [9], they modeled the problem using slightly different parameters, and here we describe their bounds using our parameters. Basically, they defined a parameter d' (named d in their paper) in the preprocessing phase and η in the update phase. Their η is equivalent to our d. Besides, fixing $d_\star$ and n_{off}, our input instances can be reduced to theirs with d' at most $n_{\text{off}} + d_\star$. In the other direction, fixing d', their input instances can be reduced to ours with $d_\star + n_{\text{off}}$ at most $3d'$. Furthermore, their space complexity was not specified, but to our best knowledge, it should be roughly proportional to their preprocessing time.
[3] This lower bound is obtained from input instances with $n_{\text{off}} = \Theta(n)$ and m is roughly linear to n.

The Decremental Setting

We show a better vertex-failure connectivity oracle by improving the $\widehat{O}(d^2)$ update time of the current almost-optimal solution [14] to $\widetilde{O}(d^2)$. See Corollary 8 for a detailed version of Theorem 1.

▶ **Theorem 1.** *There exists a deterministic vertex-failure connectivity oracle with $\widehat{O}(m) + \widetilde{O}(d_\star m)$ preprocessing time, $\widetilde{O}(m)$ space, $\widetilde{O}(d^2)$ update time and $O(d)$ query time.*

Same as all the previous vertex-failure connectivity oracles, we can substitute all the m factors in Theorem 1 with $\bar{m} = \min\{m, n(d_\star + 1)\}$ using a standard sparsification by Nagamochi and Ibaraki [15] at a cost of an additional $O(m)$ preprocessing time.

Table 1 Complexity of known vertex-failure connectivity oracles. All the m factors can be replaced by $\bar{m} = \min\{m, n(d_\star + 1)\}$ at a cost of an additional $O(m)$ preprocessing time. The randomized algorithms are all Monte Carlo. The notation $\bar{O}(\cdot)$ hides a polyloglog(n) factor.

	Det./ Rand.	Space	Preprocessing	Update	Query
Block trees, SQRT trees, and [10] only when $d_\star \le 3$	Det.	$O(n)$	$\tilde{O}(m)$	$O(1)$	$O(1)$
Duan & Pettie [4] for $c \ge 1$	Det.	linear in preprocessing time	$\tilde{O}(md_\star^{1-\frac{2}{c}} n^{\frac{1}{c}-\frac{1}{c\log(2d_\star)}})$	$\tilde{O}(d^{2c+4})$	$O(d)$
Duan & Pettie [5]	Det.	$O(md_\star \log n)$	$O(mn \log n)$	$O(d^3 \log^3 n)$	$O(d)$
	Rand.	$O(m \log^6 n)$	$O(mn \log n)$	$\bar{O}(d^2 \log^3 n)$ w.h.p.	$O(d)$
Brand & Saranurak [18]	Rand.	$O(n^2)$	$O(n^\omega)$	$O(d^\omega)$	$O(d^2)$
Pilipczuk et al. [17]	Det.	$m2^{2^{O(d_\star)}}$	$mn^2 2^{2^{O(d_\star)}}$	$2^{2^{O(d_\star)}}$	$2^{2^{O(d_\star)}}$
	Det.	$n^2\text{poly}(d_\star)$	$\text{poly}(n)2^{O(d_\star \log d_\star)}$	$\text{poly}(d_\star)$	$\text{poly}(d_\star)$
Long & Saranurak [14]	Det.	$O(m \log^3 n)$	$O(mn \log n)$	$\bar{O}(d^2 \log^3 n \log^4 d)$	$O(d)$
	Det.	$O(m \log^* n)$	$\hat{O}(m) + \tilde{O}(d_\star m)$	$\hat{O}(d^2)$	$O(d)$
Kosinas [13]	Det.	$O(d_\star m \log n)$	$O(d_\star m \log n)$	$O(d^4 \log n)$	$O(d)$
This paper	Det.	$O(m \log^3 n)$	$\widehat{O}(m) + O(d_\star m \log^3 n)$	$O(d^2(\log^7 n + \log^5 n \log^4 d))$	$O(d)$

We emphasize that our result is a strict improvement on [14]. In addition to the improvement on update time, our algorithm also improves the hidden subpolynomial overheads on the preprocessing time. We achieve this by giving a new construction algorithm of the *low degree hierarchy*, a graph decomposition technique widely used in this area [5, 14, 16]. Roughly speaking, the previous almost-linear-time construction [14] relies on modern graph techniques including vertex expander decomposition and approximate vertex capacitated maxflow algorithm, which are highly complicated and will bring relatively large subpolynomial overheads. Our new construction bypasses the vertex expander decomposition to obtain improvement on both efficiency and quality, which is also considerably simpler. Finally, we point out that the subpolynomial factors in our preprocessing time still comes from the construction of the low degree hierarchy, which can be traced back to the subpolynomial overheads of the current approximate vertex capacitated maxflow algorithm [2].

The Fully Dynamic Setting

We also show a better fully dynamic sensitivity oracle for subgraph connectivity, with update time and query time matching the decremental bounds up to polylogarithmic factors. See Theorem 17 for a detailed version of Theorem 2. The first upper bound $\widehat{O}(m)+\widetilde{O}(m(n_{\text{off}}+d_\star))$ is obtained from a combinatorial algorithm[4].

[4] *Combinatorial* algorithms [1] are algorithms that do not use fast matrix multiplication.

▶ **Theorem 2.** *There exists a deterministic fully dynamic sensitivity oracle for subgraph connectivity with $\widehat{O}(m) + \widetilde{O}(\min\{m(n_{\text{off}} + d_\star), n^\omega\})$ preprocessing time, $\widetilde{O}(\min\{m(n_{\text{off}} + d_\star), n^2\})$ space, $\widetilde{O}(d^2)$ update time and $O(d)$ query time, where ω is the exponent of matrix multiplication.*

Table 2 Complexity of known fully dynamic sensitivity oracles for subgraph connectivity.

	Det./ Rand.	Space	Preprocessing	Update	Query
Henzinger & Neumann [8]	Det.	$\widetilde{O}(n_{\text{off}}^2 m)$	$\widehat{O}(n_{\text{off}}^2 m) + \widetilde{O}(d_\star n_{\text{off}}^2 m)$	$\widehat{O}(d^4)$	$O(d^2)$
Hu, Kosinas & Polak [9]	Det.	$\widetilde{O}((n_{\text{off}} + d_\star)m)$	$\widetilde{O}((n_{\text{off}} + d_\star)m)$	$\widetilde{O}(d^4)$	$O(d)$
This paper	Det.	$O(\min\{(n_{\text{off}} + d_\star)m \log^2 n, n^2\})$	$\widehat{O}(m)+$ $O(\min\{(n_{\text{off}} + d_\star)m, n^\omega\} \log^2 n)$	$O(d^2 \log^7 n)$	$O(d)$

We also show conditional lower bounds on the preprocessing time and the space, which separate the fully dynamic and decremental settings. Furthermore, combining our new lower bounds and the existing ones, our solution in Theorem 2 is optimal up to subpolynomial factors.

▶ **Theorem 3.** *Let $\mathcal{A}$ be a fully dynamic sensitivity oracle for subgraph connectivity with S space, t_p preprocessing time, t_u update time and t_q query time upper bounds. Assuming popular conjectures, we have the following:*

1. *If $t_u + t_p = f(d) \cdot n^{o(1)}$, then $S = \widehat{\Omega}(n^2)$. (See Lemma 7.10 in the full version)*
2. *If $t_u + t_p = f(d) \cdot n^{o(1)}$, then $t_u = \widehat{\Omega}((n_{\text{off}} + d)m)$ (See [9])*
3. *If $t_u + t_p = f(d) \cdot n^{o(1)}$, then $t_u = \widehat{\Omega}(n^{\omega_{\text{bool}}})$ (See Lemma 7.3 in the full version)*
4. *If $t_p = \text{poly}(n)$, then $t_u + t_q = \widehat{\Omega}(d^2)$. (See [14])*
5. *If $t_p = \text{poly}(n)$ and $t_u = \text{poly}(dn^{o(1)})$, then $t_q = \widehat{\Omega}(d)$. (See [7])*

The $f(d)$ above can be an arbitrary growing function, and ω_{bool} is the exponent of Boolean matrix multiplication.

We make some additional remarks here. When discussing lower bounds, we assume $d = d_\star$ for each update. The lower bound on the space (item 1) holds even when the input graphs are sparse, so it naturally holds for input graphs with general density. The lower bounds on the preprocessing time (items 2 and 3) are not contradictory, because item 2 is obtained from sparse graphs while item 3 is obtained from dense graphs. The lower bounds on the update time and query time (items 4 and 5) are from those in the decremental setting. See Section 7 in the full version for the omitted proofs and more discussions.

1.2 Organization

In Section 2, we give an overview of our techniques. In Section 3, we give the preliminaries. In Section 4, we introduce our new construction of the low degree hierarchy and obtain a better vertex-failure connectivity oracle as a corollary. In Sections 5 and 6, we describe the preprocessing, update and query algorithms of our fully dynamic sensitivity oracle for subgraph connectivity. Due to space constraints, some proofs are omitted and can be found in the full version. In particular

2 Technical Overview

Better Vertex-Failure Connectivity Oracles

Our main contribution is a new construction of the *low degree hierarchy*. Then we obtain a better vertex-failure connectivity oracle as a corollary by combining the new construction of the low degree hierarchy and the remaining part in [14].

It is known that the construction of the low degree hierarchy can be reduced to $O(\log n)$ calls to the *low-degree Steiner forest decomposition* [5, 14]. Basically, for an input graph G with terminal set $U \subseteq V(G)$, we say a forest $F \subseteq E(G)$ is a *Steiner forest* of U in G if F spans the whole U (may also span some additional non-U vertices) and for each $u, v \in U$, u, v are connected in F if and only if they are connected in G. We propose a new almost-linear time low-degree Steiner forest decomposition algorithm as shown in Lemma 4, which improves the degree parameter Δ from $n^{o(1)}$ to $O(\log^2 n)$ compared to the previous one by [14]. This will leads to an improvement to the quality of the low degree hierarchy, and finally reflects on the update time.

▶ **Lemma 4** (Lemma 12, Informal). *Let G be an undirected graph with terminals $U \subseteq V(G)$. There is an almost-linear-time algorithm that computes a* separator *$|X| \subseteq V(G)$ of size $|X| \leq |U|/2$, and a low-degree Steiner forest of $U \setminus X$ in $G \setminus X$ with maximum degree $\Delta = O(\log^2 n)$.*

In the following discussion, we assume $U = V(G)$ for simplicity (hence spanning trees/-forests and Steiner trees/forests are now interchangable). To obtain Lemma 4, our starting point is that it is *not* necessary to perform a vertex expander decomposition (which will bring large $n^{o(1)}$ overheads to Δ) to get a low-degree Steiner forest decomposition. Basically, in [14], they obtain a fast low-degree Steiner forest decomposition by first proving that any vertex expander admits a low-degree spanning tree, so then it suffices to perform the stronger vertex expander decomposition. The way they prove the former is to argue that for any vertex expander H, one can embed another expander W into H with low vertex-congestion, which implies that H has a low-degree subgraph including all vertices in $V(H)$.

The key observation is that, to make the above argument work, W does not need to be an expander and W can be an arbitrary *connected graph*. This inspires us to design the following subroutine Lemma 5. Then Lemma 4 can be shown by invoking Lemma 10 using a standard divide-and-conquer framework.

▶ **Lemma 5** (Lemma 10, Informal). *Let G be an undirected graph. There is an almost-linear-time algorithm that computes either*

- *a balanced sparse vertex cut (L, S, R) with $|R| \geq |L| \geq |V(G)|/12$ and $|S| \leq 1/(100 \log n) \cdot |L|$.*
- *a large subset $V' \subseteq V(G)$ with $|V'| \geq 3|U|/4$ s.t. we can embed a connected graph W' with $V(W') = V'$ into $G[V']$ with vertex congestion $O(\log^2 n)$, which implies a spanning tree in $G[V']$ with maximum degree $O(\log^2 n)$.*

We design the algorithm in Lemma 5 using a simplified *cut-matching game*. The original cut-matching game [12, 11] can be used to embed an expander into a graph with low congestion (or produce a balanced sparse cut). To embed a connected graph, consider the following procedure. Assume a standard matching player (i.e. Lemma 9) which, given a graph G and a balanced partition (A, B) of $V(G)$, either embeds a large matching between A and B into G with low vertex-congestion or outputs a balanced sparse vertex cut in almost-linear time. Start with a graph W with $V(W) = V(G)$ but no edge and perform several rounds. At each round, we (as the cut player) partition the connected components of W into two parts with balanced sizes, and feed the partition to the matching player. If the matching player gives a matching, we add it to W and go to the next round. The game stops once a giant connected component (of size at least $3|V(G)|/4$) appears in W, which will roughly serve as W'. Roughly speaking, the game will stop in $O(\log n)$ rounds because at each round, there exists a large fraction of vertices, s.t. for each of them (say vertex v), the component containing v has its size doubled.

Fully Dynamic Sensitivity Oracles for Subgraph Connectivity

Our fully dynamic oracle is actually a generalization of a simplified version of the decremental oracle in [14].

Initially, we construct a low degree hierarchy on the activated subgraph $G_{\text{on}} := G[V_{\text{on}}]$. As mentioned in [14], the hierarchy will roughly reduce G_{on} to the following *semi-bipartite* form. First, V_{on} can be partitioned into L_{on} and R_{on}, called *left on-vertices* and *right on-vertices* respectively, s.t. there is no edge connecting two vertices in R_{on}. Second, L_{on} is spanned by a known path τ. Therefore, we assume the original graph G has a semi-bipartite G_{on} from now.

When there is no off-vertices initially (i.e. the decremental setting), the properties of a semi-bipartite G_{on} naturally support the following update and query strategy. In the update phase, removing vertices in D will break the path τ into at most $d+1$ *intervals*, and we will somehow (we will not explain this in the overview) recompute the connectivity of these intervals in the graph $G_{\text{on}} \setminus D$. Then, for each query of $u, v \in V_{\text{on}} \setminus D$, it suffices to find two intervals I_u, I_v connecting with u, v in $G_{\text{on}} \setminus D$ respectively. When u, v are left on-vertices, I_u, I_v can be found trivially. When u, v are right on-vertices, we just need to scan at most $d+1$ neighbors of each of u, v, which takes $O(d)$ time. Note that removing D will generate at most $d+1$ intervals is a crucial point to achieve fast update time.

Back to the fully dynamic setting, for an update D, in addition to removing vertices $D_{\text{on}} := D \cap V_{\text{on}}$ from G_{on}, we will also add vertices $D_{\text{off}} := D \cap V_{\text{off}}$ and their incident edges. The key observation is that $G[V_{\text{on}} \cup D_{\text{off}}]$ is still roughly a semi-bipartite graph. The first property will still hold if we put the newly activated vertices D_{off} into the left side. The second property may not hold because we do not have a path spanning the new left vertices $L_{\text{on}} \cup D_{\text{off}}$. However, this will not hurt because we can still partition $L_{\text{on}} \cup D_{\text{off}}$ into $O(d)$ connected parts after removing D_{on} from $G[V_{\text{on}} \cup D_{\text{off}}]$, i.e. at most $d+1$ intervals covering $L_{\text{on}} \setminus D_{\text{on}}$, and at most d vertices in D_{off}.

Giving this key observation, it is quite natural to adapt the decremental algorithm to the fully dynamic setting. Using the ideas of *adding artificial edges* (intuitively, substituting each right vertex and its incident edges with an artificial clique on its left neighbors) and applying *2D range counting structure*, we can design an update algorithm with $\widetilde{O}(d^3)$ update time [5]. To improve the update time to $\widetilde{O}(d^2)$, we can use a *Borůvka's styled* update algorithm and implement it by considering *batched adjacency queries* on intervals [14].

3 Preliminaries

Throughout the paper, we use the standard graph theoretic notation. For any graph, we use $V(\cdot)$ and $E(\cdot)$ to denote its vertex set and edge set respectively. If there is no other specification, we use G to denote the original graph on which we will build the oracle, and we let $n = |V(G)|$ and $m = |E(G)|$. Initially, the vertices $V(G)$ in the original graph are partitioned into *on-vertices* V_{on} and *off-vertices* V_{off}, and we let $n_{\text{off}} = |V_{\text{off}}|$. For a graph H and any $S \subseteq V(H)$, we let $H[S]$ denote the subgraph induced by vertices S. Also, for any $S \subseteq V(H)$, we use $H \setminus S$ to denote the graph after removing vertices in S and edges incident to them. Similarly, for any $F \subseteq E(H)$, $H \setminus F$ denote the graph after removing edges in F.

We also use the notion of *multigraphs*. For a multigraph H, its edge set $E(H)$ is a *multiset*. We use $+$ and $\sum$ to denote the union operation and use $-$ to denote the subtraction operation on multiset. We let ω denote the exponent of matrix multiplication and ω_{bool} denote the exponent of Boolean matrix multiplication. To our best knowledge, currently ω and ω_{bool} have the same upper bound.

4 The Low Degree Hierarchy

The *low degree hierarchy* was first introduced in [5] to design efficient vertex-failure connectivity oracles. The construction of this hierarchy in [5] is based on the approximate minimum degree Steiner forest algorithm of [6], which gives $\tilde{O}(mn)$ construction time. Later, an alternative construction algorithm was shown in [14] by exploiting vertex expander decomposition, which improves the construction time to $m^{1+o(1)}$, at a cost of a small quality loss.

In this section, we will show a new construction algorithm, which still runs in almost-linear time and gives a hierarchy with quality better than the one in [14] (but still worse than the one in [5]). To obtain the quality improvement, we basically simplify the construction in [14] and bypass the vertex expander decomposition.

We define the low degree hierarchy in Definition 6, and the main result of this section is Theorem 7. It was known that constructing a low degree hierarchy reduces to several rounds of *low-degree Steiner forest decomposition*. In Section 4.1, we introduce our key subroutine Lemma 10, which given an input graph, either computes a balanced sparse vertex cut or a low-degree Steiner tree covering a large fraction of terminals. In Section 4.2, we show the low-degree Steiner forest decomposition algorithm Lemma 12 using Lemma 10 in a standard divide and conquer framework, and then complete the proof of Theorem 7.

▶ **Definition 6** (Low Degree Hierarchy [5], Definition 5.1 in [14]). *Let G be a connected undirected graph. A (p,Δ)-low degree hierarchy with height p and degree parameter Δ on G is a pair $(\mathcal{C},\mathcal{T})$ of sets, where $\mathcal{C}$ is a set of vertex-induced connected subgraphs called components, and $\mathcal{T}$ is a set of Steiner trees with maximum vertex degree at most Δ.*

The set $\mathcal{C}$ of components is a laminar set. Concretely, it satisfies the following properties.

(1) *Components in $\mathcal{C}$ belong to p levels and we denote by $\mathcal{C}_i$ the set of components at level i. In particular, at the top level p, $\mathcal{C}_p = \{G\}$ is a singleton set with the whole G as the unique component. Furthermore, for each level $i \in [1,p]$, components in $\mathcal{C}_i$ are vertex-disjoint and there is no edge in $E(G)$ connecting two components in $\mathcal{C}_i$.*

(2) *For each level $i \in [1,p-1]$ and each component $\gamma \in \mathcal{C}_i$, there is a unique component $\gamma' \in \mathcal{C}_{i+1}$ such that $V(\gamma) \subseteq V(\gamma')$, where we say that γ' is the parent-component of γ and that γ is a child-component of γ'.*

(3) *For each component $\gamma \in \mathcal{C}$, the terminals of γ, denoted by $U(\gamma)$, are vertices in γ but not in any of γ's child-components. Note that $U(\gamma)$ can be empty. In particular, for each $\gamma \in \mathcal{C}_1$, $U(\gamma) = V(\gamma)$.*

Generally, for each level $i \in [1,p]$, we define the terminals at level i be terminals in all components in $\mathcal{C}_i$, denoted by $U_i = \bigcup_{\gamma \in \mathcal{C}_i} U(\gamma)$.

The set $\mathcal{T}$ of low-degree Steiner trees has the following properties.

(4) *$\mathcal{T}$ can also be partitioned into subsets $\mathcal{T}_1, ..., \mathcal{T}_p$, where $\mathcal{T}_i$ denote trees at level i and trees in $\mathcal{T}_i$ are vertex-disjoint.*

(5) *For each level $i \in [1,p]$ and tree $\tau \in \mathcal{T}_i$, the terminals of τ is defined by $U(\tau) = U_i \cap V(\tau)$.*

(6) *For each level $i \in [1,p]$ and each component $\gamma \in \mathcal{C}_i$ with $U(\gamma) \neq \emptyset$, there is a tree $\tau \in \mathcal{T}_i$ such that $U(\gamma) \subseteq U(\tau)$, denoted by $\tau(\gamma)$. We emphasize that two different components γ and $\gamma' \in \mathcal{C}_i$ may correspond to the same tree $\tau \in \mathcal{T}_i$.*

For better understanding, we note that the terminal sets of component $\{U(\gamma) \mid \gamma \in \mathcal{C}\}$, levels $\{U_i \mid 1 \leq i \leq p\}$, and Steiner trees $\{U(\tau) \mid \tau \in \mathcal{T}\}$ are all partitions of $V(G)$. One may also get the picture of the hierarchy from the perspective of construction. See the construction described in Algorithm 2, which invokes Lemma 12 in a black-box way.

▶ **Theorem 7.** *Let G be an undirected graph. There is a deterministic algorithm that computes a (p,Δ)-low degree hierarchy with $p = O(\log n)$ and $\Delta = O(\log^2 n)$. The running time is $m^{1+o(1)}$.*

Corollary 8 is obtained by substituting the construction of low degree hierarchy in [14] with ours. Formally speaking, it is a corollary of Theorem 7, and Lemma 6.14, Theorem 7.2 and Section 7.3 in [14].

▶ **Corollary 8.** *There is a deterministic vertex-failure connectivity oracle with $\widehat{O}(m) + O(d_\star m \log^3 n)$ preprocessing time, $O(m \log^3 n)$ space, $O(d^2(\log^7 n + \log^5 n \cdot \log^4 d))$ update time and $O(d)$ query time.*

4.1 A Balanced Sparse Vertex Cut or a Low-Degree Steiner Tree

The goal of this subsection is to show Lemma 10, a subroutine which given a graph with terminals, outputs either a balanced sparse vertex cut or a low-degree Steiner tree covering a large fraction of terminals. In fact, some expander decomposition algorithms (e.g. [3]) exploit a similar subroutine which either computes a balanced sparse cut or certifies that a large part of the graph is an expander. Our subroutine can be viewed as a weaker and simplified version, because similar to the notion of expanders, a low-degree Steiner tree is also an object that certifies some kind of (weaker) well-linkedness.

At a high level, our algorithm uses a simplified *cut-matching-game* framework. A cut-matching game is an interactive process between a cut player and a matching player with several rounds. Start from a graph with no edge. In each round, the cut player will select a cut and then the matching player is required to add a perfect matching on this cut. It is known that there exists cut-player strategies against an arbitrary matching player that guarantees the final graph is an expander after $\widetilde{O}(1)$ rounds [12, 11]. In the proof of Lemma 10, we show a cut-player strategy that only guarantees the final graph is a *connected graph.* Combining a classic matching player as shown in Lemma 9, we can either find a balanced sparse vertex cut or embed a connected graph covering most of the terminals into the original graph with low vertex congestion. In the latter case, the embedding leads to a low-degree subgraph covering most of the terminals. Finally, picking an arbitrary spanning tree in this subgraph suffices.

Given a cut w.r.t. terminals, Lemma 9 will either output a balanced sparse vertex cut or a large matching between terminals that is embeddable into the original graph with low vertex congestion. In fact, it is a simplified version of the matching player in [14], and the proof can be found in Appendix A.1 of the full version.

▶ **Lemma 9.** *Let G be an undirected graph with a terminal set U. Given a parameter ϕ and a partition (A,B) of U, there is a deterministic algorithm that computes either*

- *a vertex cut (L,S,R) with $|R \cap U| \geq |L \cap U| \geq \min\{|A|,|B|\}/3$ and $|S| \leq \phi \cdot |L \cap U|$, or*
- *a matching M between A and B with size $|M| \geq \min\{|A|,|B|\}/3$ s.t. there is an embedding $\Pi_{M \to G}$ of M into G with vertex congestion at most $\lceil 1/\phi \rceil$.*

The running time is $m^{1+o(1)}$. If the output is a matching M, the algorithm can further output the edge set $E(\Pi_{M \to G})$ of the embedding $\Pi_{M \to G}$.

▶ **Lemma 10.** *Let G be an undirected graph with a terminal set U. Given parameters $0 < \epsilon, \phi \leq 1/4$, there is a deterministic algorithm that computes either*

- *a vertex cut (L, S, R) with $|R \cap U| \geq |L \cap U| \geq \epsilon|U|/3$ and $|S| \leq \phi \cdot |L \cap U|$, or*
- *a subset $U_{\text{drop}} \subseteq U$ of terminals with $|U_{\text{drop}}| \leq \epsilon|U|$ and a Steiner tree τ on $G \setminus U_{\text{drop}}$ of terminal set $U \setminus U_{\text{drop}}$ with maximum degree $O(\log|U|/\phi)$.*

Proof. The algorithm is made up of an iteration phase and a postprocessing phase. The iteration phase will maintain an incremental graph W with $V(W) = U$, called the *witness graph*, and its embedding $\Pi_{W \to G}$ into G. Precisely, instead of storing the embedding $\Pi_{W \to G}$ explicitly, the algorithm will only store its edge set $E(\Pi_{W \to G})$. Initially, the witness graph $W^{(0)}$ has no edge and $E(\Pi_{W^{(0)} \to G})$ is empty. We use $W^{(i)}$ and $E(\Pi_{W^{(i)} \to G})$ to denote the witness graph and the edge set of the embedding right after the i-th round.

In the iteration phase, we do the following steps in the i-th round.

1. We compute all the connected components of $W^{(i-1)}$, which forms a partition $\mathcal{Q}^{(i-1)}$ of U s.t. each $Q \in \mathcal{Q}^{(i-1)}$ is a subset of $|U|$, called a *cluster*. If there is a cluster $Q^\star \in \mathcal{Q}^{(i-1)}$ has $|Q^\star| \geq (1-\epsilon)|U|$, then we terminate the iteration phase and go to the postprocessing phase, otherwise we proceed to the next step.
2. Because step 1 guarantees that all clusters in $\mathcal{Q}^{(i-1)}$ have size at most $(1-\epsilon)|U|$, we will partition $\mathcal{Q}^{(i-1)}$ into two groups $\mathcal{Q}_A$ and $\mathcal{Q}_B$ depending on the following two cases.

   (a) If all clusters in $\mathcal{Q}^{(i-1)}$ have size at most $|U|/2$, then we partition $\mathcal{Q}^{(i-1)}$ into $\mathcal{Q}_A$ and $\mathcal{Q}_B$ s.t. $\sum_{Q \in \mathcal{Q}_A} |Q| \geq |U|/4$ and $\sum_{Q \in \mathcal{Q}_B} |Q| \geq |U|/4$.
   (b) Otherwise, there is a unique cluster $Q^\star$ s.t. $|U|/2 < |Q^\star| \leq (1-\epsilon)|U|$, and we let $\mathcal{Q}_A = \mathcal{Q} \setminus \{Q^\star\}$ and $\mathcal{Q}_B = \{Q^\star\}$.

   Let $A_i = \bigcup_{Q \in \mathcal{Q}_A} Q$ and $B_i = \bigcup_{Q \in \mathcal{Q}_B} Q$. Note that by definition, (A_i, B_i) forms a partition of U. We have $|A_i|, |B_i| \geq |U|/4$ in case (a) and $|A_i|, |B_i| \geq \epsilon|U|$ in case (b).
3. We apply Lemma 9 on graph G and terminal U with parameter ϕ and the partition (A_i, B_i) of U. If we get a vertex cut (L, S, R), it will satisfy $|R \cap U| \geq |L \cap U| \geq \min\{|A_i|, |B_i|\}/3 \geq \epsilon|U|/3$ and $|S| \leq \phi \cdot |L \cap U|$ as desired, so we can terminate the whole algorithm with (L, S, R) as the output. Otherwise, we get a matching M_i between A_i and B_i with size $|M_i| \geq |A_i|/3$, and the edge set $E(\Pi_{M_i \to G})$ of some embedding $\Pi_{M_i \to G}$ that has vertex congestion $O(1/\phi)$. Then we let $W^{(i)} = W^{(i-1)} \cup M_i$ and $E(\Pi_{W^{(i)} \to G}) = E(\Pi_{W^{(i-1)} \to G}) \cup E(\Pi_{M_i \to G})$, and proceed to the next round.

If the algorithm does not end at step 3, it exits the iteration phase at step 1, and then we perform the following postprocessing phase. Let W denote the final witness graph with connected components $\mathcal{Q}$ and a cluster $Q^\star \in \mathcal{Q}$ s.t. $|Q^\star| \geq (1-\epsilon)|U|$. Note that $Q^\star \subseteq U$. Let G' be the subgraph of G induced by $E(\Pi_{W \to G})$. By the definition of embedding, vertices in $Q^\star$ are also connected in G'. In other words, $Q^\star$ is contained by a connected component of G'. We can take an arbitrary spanning tree τ of this component as a Steiner tree of $V(\tau) \cap U$, and define $U_{\text{drop}} = U \setminus V(\tau)$ be the uncovered terminals.

We now show that U_{drop} and τ have the desire property. The number of uncovered terminals is bounded by $|U_{\text{drop}}| \leq |U| - |V(\tau) \cap U| \leq |U| - |Q^\star| \leq \epsilon|U|$, and τ is a Steiner tree of $U \setminus U_{\text{drop}}$ with maximum degree $O(\log|U|/\phi)$ because G' has maximum degree at most the vertex congestion of $\Pi_{W \to G}$, which is at most $O(\log|U|/\phi)$ by Claim 11.

▷ Claim 11. The number of rounds in the iteration phase is at most $O(\log|U|)$, and the vertex congestion of the final embedding $\Pi_{W \to G}$ is at most $O(\log|U|/\phi)$.

Proof. Note that the early rounds will go into case (a) in step 2, while the late rounds will go into case (b). We bound the number of case-(a) rounds and case-(b) rounds separately.

The number of case-(a) rounds is at most $O(\log|U|)$ by the following reason. We define a potential function $\Phi(W)$ of the witness graph by

$$\Phi(W) = \sum_{Q\in\mathcal{Q}}\sum_{v\in Q}\log|Q| = \sum_{Q\in\mathcal{Q}}|Q|\cdot\log|Q|.$$

In particular, for each cluster $Q\in\mathcal{Q}$ and each vertex $v\in Q$, we say the potential at v is $\log|Q|$.

Because initially $\Phi(W^{(0)})=0$ and we always have $\Phi(W)\le|U|\log|U|$, it is sufficient to show that each case-(a) round increases the potential by at least $\Omega(|U|)$. To see this, consider the i-th case-(a) round. For each matching edge $\{u,v\}\in M_i$, let $Q_v^{(i-1)}$ (resp. $Q_u^{(i-1)}$) be the connected component of $W^{(i-1)}$ that contains v (resp. u), and assume without loss of generality that $|Q_v^{(i-1)}|\le|Q_u^{(i-1)}|$. Then the connected component $Q_v^{(i)}$ of $W^{(i)}$ that contains v will have $|Q_v^{(i)}|\ge 2|Q_v^{(i-1)}|$, because $Q_v^{(i)}\supseteq Q_u^{(i-1)}\cup Q_v^{(i-1)}$. In other words, this round will increase the potential at v (from $\log|Q_v^{(i-1)}|$ to $\log|Q_v^{(i)}|$) by at least 1. Summing over $|M_i|$ matching edges, the total potential $\Phi(W)$ will be increased by at least $|M_i|\ge|U|/12$ as desired, because the potential at any $v\in V(W)$ will never drop.

It remains to show that the number of case-(b) rounds is at most $O(\log|U|)$. This is simple because in each round, the matching M_i will merge at least $|A_i|/3$ terminals in $|A_i|$ into the giant cluster $Q^\star$, which means $|Q^\star|$ will reach the threshold $(1-\epsilon)|U|$ in $O(\log|U|)$ many case-(b) rounds and then the iteration phase ends.

The final embedding $\Pi_{W\to G}$ has vertex congestion $O(\log|U|/\phi)$ because there are $O(\log|U|)$ rounds and the embedding $\Pi_{M_i\to G}$ has vertex congestion $O(1/\phi)$ each round. ◁

◀

4.2 The Low-Degree Steiner Forest Decomposition

Lemma 12 describes the low-degree Steiner forest decomposition algorithm, which invokes Lemma 10 in a divide-and-conquer fashion. For simplicity, the readers can always assume $\epsilon=1/2$, which is the value we will choose when constructing the low degree hierarchy. We introduce this tradeoff parameter ϵ just to show that our algorithm has the same flexibility as those in [5, 14].

▶ **Lemma 12.** *Let G be an undirected graph with a terminal set U. Given a parameter $0<\epsilon\le1/2$, there is a deterministic algorithm that computes*

- *a vertex set $X\subseteq V(G)$, called the separator, s.t. $|X|\le\epsilon|U|$, and*
- *for each connected component Y of $G\setminus X$ s.t. U intersects $V(Y)$, a Steiner tree τ_Y spanning $U\cap V(Y)$ on Y with maximum degree $O(\log^2|U|/\epsilon)$.*

The running time is $m^{1+o(1)}/\epsilon$.

We include the algorithm of Lemma 12 in Algorithm 1, and the complete proof can be founded in the full version.

Algorithm 1 The low-degree Steiner forest decomposition $\textsc{SFDecomp}(G, U)$.

```
Input: An undirected graph $G$ with terminals $U$.
Output: A separator $X$ and a collection $\mathcal{T}$ of Steiner trees $\{\tau_Y\}$.
1: Let $\epsilon' = \epsilon/2$ and $\phi = \epsilon'/\log|U|$
2: Apply Lemma 10 on $G$ and $U$ with parameters $\epsilon'$ and $\phi$.
3: if Lemma 10 outputs a vertex cut $(L, S, R)$ then
4:     $(X_L, \mathcal{T}_L) \leftarrow \textsc{SFDecomp}(G[L], L \cap U)$
5:     $(X_R, \mathcal{T}_R) \leftarrow \textsc{SFDecomp}(G[R], R \cap U)$
6:     Return $X = X_L \cup X_R \cup S$ and $\mathcal{T} = \mathcal{T}_L \cup \mathcal{T}_R$.
7: else
8:     Otherwise Lemma 10 outputs $U_{\text{drop}} \subseteq U$ and a Steiner tree $\tau$ of $U \setminus U_{\text{drop}}$ on $G \setminus U_{\text{drop}}$.
9:     Return $X = U_{\text{drop}}$ and $\mathcal{T} = \{\tau\}$.
10: end if
```

As shown in [5], to construct a low-degree hierarchy $(\mathcal{C}, \mathcal{T})$, it suffices to invoke the low-degree Steiner forest decomposition (with $\epsilon = 1/2$) $O(\log n)$ times. The algorithm is shown in Algorithm 2, and the proof of correctness is included in Appendix A.2 in the full version.

Algorithm 2 The construction of the low-degree hierarchy.

```
Input: An undirected graph $G$.
Output: A low-degree hierarchy $(\mathcal{C}, \mathcal{T})$.
1: Initialize $i = 1$, $X_1 = V(G)$.
2: while $X_i$ is not empty do
3:     $(X_{i+1}, \mathcal{T}_i) \leftarrow \textsc{SFDecomp}(G, X_i)$ with $\epsilon = 1/2$.
4:     $i \leftarrow i + 1$.
5: end while
6: $p \leftarrow i - 1$, which denotes the number of levels.
7: for each level $i$ do
8:     $U'_i \leftarrow X_i \cup \ldots \cup X_p$.
9:     $\mathcal{C}_i \leftarrow$ the connected component of $G \setminus U'_{i+1}$ (particularly, $U'_{p+1} = \emptyset$).
10:    $U_i \leftarrow U'_i \setminus U'_{i+1}$, which denotes the terminals of level $i$.
11: end for
```

5 The Preprocessing Algorithm

In this section, we will describe the preprocessing algorithm, which basically first computes the low degree hierarchy on $G_{\text{on}} := G[V_{\text{on}}]$, and then constructs some affiliated data structures on top of the hierarchy.

The low degree hierarchy $(\mathcal{C}, \mathcal{T})$ is computed by applying Theorem 7 on G_{on}, if G_{on} is a connected graph. In the case that G_{on} is not connected, we simply apply Theorem 7 on each of the connected components of G_{on}. To simplify the notations, we use $(\mathcal{C}, \mathcal{T})$ to denote the union of hierarchies of connected components of G_{on}, and still say $(\mathcal{C}, \mathcal{T})$ is the low degree hierarchy of G_{on}. Note that $(\mathcal{C}, \mathcal{T})$ has all properties in Definition 6, except that the top level $\mathcal{C}_1$ are now made up of connected components of G_{on}.

In Section 5.1, we introduce the notions of artificial edges and the artificial graph $\hat{G}$. In Section 5.2, we define a global order π based on Euler tour orders of Steiner trees in $\mathcal{T}$, and then construct a 2D range counting structure which can answer the number of edges in $E(\hat{G})$ between two intervals on π. Finally, in Section 5.3, we summarize what we will store, and analyse the preprocessing time and the space complexity.

5.1 Artificial Edges and the Artificial Graph $\hat{G}$

The artificial graph $\hat{G}$ is a multi-graph constructed by adding some *artificial edges* into the original graph G in the following way. For each component $\gamma \in \mathcal{C}$, let A_γ collect the neighbors of $V(\gamma)$ in G, formally defined by $A_\gamma = \{v \mid v \in V(G) \setminus V(\gamma) \text{ s.t. } \exists\{u, v\} \in E(G) \text{ with } u \in V(\gamma)\}$. We call A_γ the *adjacency list* of γ. Let $A_{\gamma,\mathrm{on}} = A_\gamma \cap V_{\mathrm{on}}$ and $A_{\gamma,\mathrm{off}} = A_\gamma \cap V_{\mathrm{off}}$. Next, we let $B_{\gamma,\mathrm{off}} = A_{\gamma,\mathrm{off}}$ and let $B_{\gamma,\mathrm{on}}$ be an arbitrary subset of $A_{\gamma,\mathrm{on}}$ with size $\min\{d_\star + 1, |A_{\gamma,\mathrm{on}}|\}$. Then define $B_\gamma = B_{\gamma,\mathrm{on}} \cup B_{\gamma,\mathrm{off}}$.

The artificial edges added by the component γ is then $\hat{E}_\gamma = \{\{u, v\} \mid u \in A_\gamma, v \in B_\gamma, u \neq v\}$. Namely, $\hat{E}_\gamma$ consists of a clique on B_γ and a biclique between B_γ and $A_\gamma \setminus B_\gamma$. Finally, the artificial graph $\hat{G}$ is defined by $\hat{G} = G + \sum_{\gamma\in\mathcal{C}} \hat{E}_\gamma$. We emphasize that $\hat{G}$ is a multi-graph, and those edges connecting the same endpoints will have different identifiers.

We show some useful properties in Proposition 13. Item 3 of Proposition 13 basically says that, if A_γ has an on-vertex after update, then B_γ also has one.

▶ **Proposition 13.** *We have the following.*

1. $\sum_{\gamma\in\mathcal{C}} |A_\gamma| \le O(pm)$.
2. $|E(\hat{G})| \le O(pm(n_{\mathrm{off}} + d_\star))$.
3. *Given any update $D \subseteq V$ with $|D| \le d_\star$, if $(A_{\gamma,\mathrm{on}} \setminus D) \cup (A_{\gamma,\mathrm{off}} \cap D) \neq \emptyset$, then we have $(B_{\gamma,\mathrm{on}} \setminus D) \cup (B_{\gamma,\mathrm{off}} \cap D) \neq \emptyset$.*

Proof.

Part 1. For each $\gamma \in \mathcal{C}$, observe that $|A_\gamma| \le \sum_{v\in V(\gamma)} \deg_G(v)$. Hence $\sum_{\gamma\in\mathcal{C}} |A_\gamma| \le O(pm)$ because each vertex can appear in at most p components (at most one at each level).

Part 2. By definition, $|E(\hat{G})| \le m + \sum_{\gamma\in C} |\hat{E}_\gamma| \le m + \sum_{\gamma\in\mathcal{C}} |A_\gamma| \cdot |B_\gamma|$. Note that $|B_\gamma| \le n_{\mathrm{off}} + d_\star + 1$ for all $\gamma \in \mathcal{C}$ by construction. Combining part 1, we have $|E(\hat{G})| \le O(pm(n_{\mathrm{off}} + d_\star))$.

Part 3. If $|A_{\gamma,\mathrm{on}}| \le d_\star + 1$, we have $A_\gamma = B_\gamma$ by construction and the proposition trivially holds. Otherwise, B_γ will include $d_\star + 1$ vertices in $A_{\gamma,\mathrm{on}}$. Because $|D| \le d_\star$, at least one of them will survive in $B_{\gamma,\mathrm{on}} \setminus D$, which implies $(B_{\gamma,\mathrm{on}} \setminus D) \cup (B_{\gamma,\mathrm{off}} \cap D) \neq \emptyset$. ◀

5.2 The Global Order and Range Counting Structures

Next, we define an order, called the *global order* and denoted by π, over the whole vertex set $V(G)$, based on the Euler Tour orders of Steiner trees in $\mathcal{T}$.

For each $\tau \in \mathcal{T}$, we define its Euler tour order $\mathrm{ET}(\tau)$ as an ordered list of vertices in $V(\tau)$ ordered by the time stamps of their first appearances in an Euler tour of τ (starting from an arbitrary root). Intuitively, the Euler tour order $\mathrm{ET}(\tau)$ can be interpreted as a linearization of τ, i.e. after the removal of failed vertices in τ, the remaining subtrees will corresponding to intervals on $\mathrm{ET}(\tau)$, as shown in Lemma 14.

▶ **Lemma 14** (Lemma 6.3 in [14], Rephrased)**.** *Let τ be an undirected tree with maximum vertex degree Δ. A removal of d failed vertices from τ will split τ into at most $O(\Delta d)$ subtrees $\hat{\tau}_1, \hat{\tau}_2, ..., \hat{\tau}_{O(\Delta d)}$, and there exists a set $\mathcal{I}_\tau$ of at most $O(\Delta d)$ disjoint intervals on $ET(\tau)$, such that each interval is owned by a unique subtree and for each subtree τ_i, $V(\tau_i)$ is equal to the union of intervals it owns.*

Furthermore, by preprocessing τ in $O(|V(\tau)|)$ time, we can store $\mathrm{ET}(\tau)$ and some additional information in $O(|V(\tau)|)$ space, which supports the following operations.

- *Given a set D_τ of d failed vertices, the intervals $\mathcal{I}_t$ can be computed in $O(\Delta d \log(\Delta d))$ update time.*
- *Given a vertex $v \in V(\tau) \setminus D_\tau$, it takes $O(\log d)$ query time to find an interval $I \in \mathcal{I}_t$ s.t. vertices in I are connected to v in $\tau \setminus D_\tau$.*

Given the Euler tour orders of all $\tau \in \mathcal{T}$, we define the global order π as follows. We first concatenate $\mathrm{ET}(\tau) \cap U(\tau)$ (i.e. the restriction of $\mathrm{ET}(\tau)$ on the terminals of τ) of all $\tau \in \mathcal{T}$ in an arbitrary order, and then append all vertices in V_{off} to the end. Recall that $\{U(\tau) \mid \tau \in \mathcal{T}\}$ partitions V_{on}, so π is well-defined.

With the global order π, we will construct a 2D-range counting structure Table, which can answer the number of edges in $E(\hat{G})$ that connect two disjoint intervals on π. We first initialize $\mathsf{Table}_{\text{init}}$ to be an ordinary 2D array on range $\pi \times \pi$. For each $u, v \in \pi$, we store a non-negative integer in the entry $\mathsf{Table}_{\text{init}}(u, v)$ representing the number of edges in $E(\hat{G})$ connecting vertices u and v.

▶ **Lemma 15.** *Suppose that we can access the lists A_γ and B_γ for all $\gamma \in \mathcal{C}$. There is a combinatorial algorithm that computes $\mathsf{Table}_{\text{init}}$ in $O(|E(\hat{G})|)$ time, or $\mathsf{Table}_{\text{init}}$ can be computed in $O(p \cdot n^\omega)$ time using fast matrix multiplication.*

Proof. A trivial construction of $\mathsf{Table}_{\text{init}}$ is to construct the edge sets $E(\hat{G})$ explicitly, and then scan the edges one by one. Obviously, this takes $O(|E(\hat{G})|)$ time.

When $|E(\hat{G})|$ is large, we can use fast matrix multiplication (FMM) to speed up the construction of $\mathsf{Table}_{\text{init}}$. Recall that $E(\hat{G}) = E(G) + \sum_{\gamma \in \mathcal{C}} \hat{E}_\gamma$. We first add the contribution of $E(G)$ into $\mathsf{Table}_{\text{init}}$ using the trivial algorithm, which takes $O(m)$ time. Next, we compute the contribution of artificial edges, i.e. $\sum_{\gamma \in \mathcal{C}} \hat{E}_\gamma$, using FMM. We construct a matrix X with n rows and $|\mathcal{C}|$ columns, where rows are indexed by the global order π and columns are indexed by components (in an arbitrary order). For each vertex $u \in \pi$ and component $\gamma \in \mathcal{C}$, the entry $X(u, \gamma) = 1$ if and only if $u \in A_\gamma$. Similarly, we define an n-row $|\mathcal{C}|$-column matrix Y, in which each entry $Y(u, \gamma) = 1$ if and only if $u \in A_\gamma \setminus B_\gamma$. Let $Z = X \cdot X^\intercal - Y \cdot Y^\intercal$. Observe that, for each pair of distinct vertices $u, v \in \pi$,

$$\begin{aligned} Z(u,v) &= \sum_{\gamma \in \mathcal{C}} (X(u,\gamma) \cdot X(v,\gamma) - Y(u,\gamma) \cdot Y(v,\gamma)) \\ &= \sum_{\gamma \in \mathcal{C}} \mathbb{1}[u, v \in A_\gamma] - \mathbb{1}[u, v \in A_\gamma \setminus B_\gamma] \\ &= \sum_{\gamma \in \mathcal{C}} \mathbb{1}[\{u, v\} \in \hat{E}_\gamma]. \end{aligned}$$

Therefore, the matrix Z count the contribution of $\sum_\gamma \hat{E}_\gamma$ correctly and the last step is to add Z to $\mathsf{Table}_{\text{init}}$. The construction time is dominated by the computation of Z, which takes $O(p \cdot n^\omega)$ time because it involves multiplying an $n \times |\mathcal{C}|$ matrix and a $|\mathcal{C}| \times n$ matrix, and $|\mathcal{C}| = O(pn)$. ◀

▶ **Lemma 16.** *With access to the positive entries of* $\mathsf{Table}_{\text{init}}$*, we can construct a data structure* Table *that given any disjoint intervals* I_1 *and* I_2 *on* π*, answers in* $O(\log n)$ *time the number of edges in* $E(\hat{G})$ *with one endpoint in* I_1 *and the other one in* I_2*. The structure* Table *can be constructed in* $O(N \log n)$ *time and takes space* $O(N \log n)$*, where* N *denotes the number of positive entries in* $\mathsf{Table}_{\text{init}}$*.*

Proof. We simply construct Table as a standard weighted 2D range counting structure of $\mathsf{Table}_{\text{init}}$, By using textbook algorithms such as range trees and persistent segment trees, we can construct Table in $O(N \log n)$ time and it takes space $O(N \log n)$. The correctness of Table follows the definition of $\mathsf{Table}_{\text{init}}$. ◀

5.3 Preprocessing Time and Space Analysis

In conclusion, we will compute and store the following in the preprocessing phase.

- First, we store the low degree hierarchy $(\mathcal{C}, \mathcal{T})$. Constructing the low degree hierarchy takes $\hat{O}(m)$ time by Theorem 7. Storing the low degree hierarchy explicitly takes $O(pn)$ space, because for each level i, the components in $\mathcal{C}_i$ are vertex disjoint, also Steiner trees in $\mathcal{T}_i$.
- Next, for each $\gamma \in \mathcal{C}$, we store the lists A_γ and B_γ after ordering them by π. Computing the lists A_γ and B_γ takes $O(pm)$ time by checking the incident edges of each vertex in each component. Storing the lists A_γ and B_γ takes $O(pm)$ space by Item 1 in Proposition 13. Additionally, for each $\gamma \in \mathcal{C}$, store the list $A_{\gamma,\text{on}}$.
  For each $v \in V_{\text{off}}$ and $\gamma \in \mathcal{C}$, store a binary indicator to indicate whether $v \in A_{\gamma,\text{off}}$ or not. Computing the indicators takes $O(pm)$ time by scanning all the lists A_γ. Storing the indicators explicitly takes $O(|V_{\text{off}}| \cdot |\mathcal{C}|) = O(pn \cdot |V_{\text{off}}|)$ space.
- We also store the global order π, which takes $O(n)$ space. For each $\tau \in \mathcal{T}$, we store $\text{ET}(\tau)$ and the additional information stated in Lemma 14 in $O(|V(\tau)|)$ space. Computing the things above takes totally $\sum_{\tau \in \mathcal{T}} |V(\tau)| = O(pn)$ time by Lemma 14.
- Finally, we store the data structure Table. Combining Item 2 in Proposition 13 and Lemmas 15 and 16, we can compute Table in $O(pm(n_{\text{off}} + d) \log n)$ time using an combinatorial algorithm, or in $O(p \cdot n^\omega \log n)$ time using fast matrix multiplication. The space to store Table is $\min\{pm(n_{\text{off}} + d) \log n, n^2\}$.

In conclusion, the total preprocessing time can be upper bounded by $\widehat{O}(m) + O(pm(n_{\text{off}} + d) \log n)$ using an combinatorial algorithm, then $t_p = \hat{\Omega}(md)$, or $\hat{O}(m) + O(p \cdot n^\omega \log n)$ using fast matrix multiplication. The space complexity is $O(\min\{pm(n_{\text{off}} + d) \log n, n^2\})$. Because the low degree hierarchy has $p = O(\log n)$ levels, the preprocessing time is $\widehat{O}(m) + O(\min\{m(n_{\text{off}} + d) \log^2 n, n^\omega \log^2 n\})$, and the space is $O(\min\{m(n_{\text{off}} + d) \log^2 n, n^2\})$.

6 The Update and Query Algorithms

Let $D \subseteq V(G)$ be a given update. We use $D_{\text{on}} = D \cap V_{\text{on}}$ to denote the vertices that will be turned off in this update and $D_{\text{off}} = D \cap V_{\text{off}}$ to denote the vertices that will be turned on. Let $V_{\text{new}} = (V_{\text{on}} \setminus D_{\text{on}}) \cup D_{\text{off}}$ be the on-vertices after updates.

Our update strategy is to recompute the connectivity of a subset of *affected vertices* $Q^\star \subseteq V_{\text{new}}$ on some affected graph $G^\star$. In Section 6.1, we will define $Q^\star$ and $G^\star$, and prove that $Q^\star$ has the same connectivity on the affected graph $G^\star$ and the updated original graph $G[V_{\text{new}}]$. In Section 6.2, we will partition $G^\star$ into a small number of sets s.t. each set forms an *interval* on the global order π and it is certified to be connected by some Steiner tree in $\mathcal{T}$. Thus, it suffices to solve the connectivity of intervals on $Q^\star$, which is formalized in Lemma 19.

▶ **Theorem 17.** *There exists a deterministic fully dynamic sensitivity oracle for subgraph connectivity with* $O(\min\{m(n_{\text{off}} + d_\star)\log^2 n, n^2\})$ *space,* $O(d^2 \log^7 n)$ *update time and* $O(d)$ *query time. The preprocessing time is* $\widehat{O}(m) + O(m(n_{\text{off}} + d_\star)\log^2 n)$ *by a combinatorial algorithm, and* $\widehat{O}(m) + O(n^\omega \log^2 n)$ *using fast matrix multiplication.*

We first conclude our fully dynamic sensitivity oracle for subgraph connectivity in Theorem 17. The bounds on preprocessing time and space are shown in Section 5.3. The update time is given by Lemma 19. The query algorithm and the query time analysis are omitted here and they can be founded in the full version.

6.1 Affected Vertices $Q^\star$ and the Affected Graph $G^\star$

For each component $\gamma \in \mathcal{C}$, we call γ an *affected component* if $V(\gamma)$ intersects D_{on}, otherwise it is *unaffected*. Let $\mathcal{C}_{\text{aff}}$ denote the set of affected components. Let $\mathcal{T}_{\text{aff}} = \{\tau(\gamma) \mid \gamma \in \mathcal{C}_{\text{aff}}\}$ denote the Steiner trees corresponding to affected components.

We then define the *affected vertices* to be $Q^\star = D_{\text{off}} \cup \bigcup_{\tau\in\mathcal{T}_{\text{aff}}} U(\tau) \setminus D_{\text{on}}$. Namely, $Q^\star$ collect the newly opened vertices and the open terminals of affected components. Note that $Q^\star \subseteq V_{\text{new}}$. The *affected graph* $G^\star$ is $G^\star = \hat{G}[Q^\star] - \sum_{\gamma\in\mathcal{C}_{\text{aff}}} \hat{E}_\gamma$. In other words, $G^\star$ is the subgraph of the artificial graph $\hat{G}$ induced by the affected vertices $Q^\star$, with the artificial edges from affected components removed.

▶ **Lemma 18.** *For any two vertices* $u, v \in Q^\star$*,* u *and* v *are connected in* $G[V_{\text{new}}]$ *if and only if* u *and* v *are connected in* $G^\star$*.*

The proof of Lemma 18 can be founded in the full version. Intuitively, this lemma holds because those maximal unaffected components in $\mathcal{C}$ partition $V_{\text{new}} \setminus Q^\star$, and the artificial edges will capture the connectivity contributed by these maximal unaffected components.

6.2 Solving Connectivity of Intervals

Although the primary goal of our update algorithm is to compute the connectivity of $Q^\star$ on $G[V_{\text{new}}]$, Lemma 18 tells that it is equivalent to compute the connectivity of $Q^\star$ on $G^\star$.

▶ **Lemma 19.** *There is a deterministic algorithm that computes a partition* $\mathcal{I}$ *of* $Q^\star$ *s.t. each set* $I \in \mathcal{I}$ *forms an interval on* π *and all vertices in* $\mathcal{I}$ *are connected in* $G^\star$*, and then computes a partition* $\mathcal{R}$ *of* $\mathcal{I}$ *s.t. for each group* $R \in \mathcal{R}$*, the union of intervals in* R *forms a (maximal) connected component of* $G^\star$*. The running time is* $O(p^2 d^2 \Delta^2 \log n)$*.*

Intervals

We first describe how to compute the partition $\mathcal{I}$ of $Q^\star$. Because we require each set $I \in \mathcal{I}$ forms an *interval* on the global order π, we can represent I by the positions of its endpoints on π. Recall that $Q^\star = (\bigcup_{\gamma\in\mathcal{T}_{\text{aff}}} U(\gamma) \setminus D_{\text{on}}) \cup D_{\text{off}}$.

- We first construct the intervals of $\bigcup_{\tau\in\mathcal{T}_{\text{aff}}} U(\tau) \setminus D_{\text{on}}$ by exploiting the Steiner trees. For each $\tau \in \mathcal{T}_{\text{aff}}$, by invoking Lemma 14 on τ with failed vertices D_{on}, we will obtain a partition $\mathcal{I}'_\tau$ of $V(\tau) \setminus D_{\text{on}}$ s.t. each $I' \in \mathcal{I}'_\tau$ is an interval on $\text{ET}(\tau)$ and it is contained by a subtree of $\tau \setminus D_{\text{on}}$. We construct a set $\mathcal{I}_t$ of intervals on $\text{ET}(\tau) \cap U(\tau)$ by taking the restriction of intervals $\mathcal{I}'_t$ on $U(\tau)$. Therefore, intervals in $\mathcal{I}_t$ are indeed intervals on π because $\text{ET}(\tau) \cap U(\tau)$ is a consecutive sublist of π. Also, for each interval $I \in \mathcal{I}_\tau$, vertices in I are connected in $G[V_{\text{new}}]$ (because $\tau \setminus D_{\text{on}}$ is a subgraph of $G[V_{\text{new}}]$), which implies vertices in I are connected in $G^\star$ by Lemma 18.
- For each vertex $v \in D_{\text{off}} \subseteq Q^\star$, we construct a singleton interval $I_v = \{v\}$.

Finally, the whole set of intervals is $\mathcal{I} = \bigcup_{\tau \in \mathcal{T}_{\text{aff}}} \mathcal{I}_\tau \cup \{I_v \mid v \in D_{\text{off}}\}$.

▶ **Proposition 20.** *The total number of intervals is $|\mathcal{I}| = O(pd\Delta)$, and computing all intervals takes $O(pd\Delta \log(d\Delta))$ time.*

Proof. By Lemma 14, the number of intervals generated by a tree $\tau \in \mathcal{T}_{\text{aff}}$ is at most $O(|V(\tau) \cap D_{\text{on}}| \cdot \Delta)$, and it takes $O(|V(\tau) \cap D_{\text{on}}| \cdot \Delta \cdot \log(|V(\tau) \cap D_{\text{on}}| \cdot \Delta))$ time to generate them. Observe that $\sum_{\tau \in \mathcal{T}_{\text{aff}}} |V(\tau) \cap D_{\text{on}}| = O(p \cdot |D_{\text{on}}|)$ because each vertex in D_{on} can appear in at most p trees in $\mathcal{T}$ (at most one at each level). Furthermore, the trivial intervals generated by vertices in D_{off} is obviously $|D_{\text{off}}|$. Therefore, the total number of intervals in $O(p|D_{\text{on}}|\Delta) + |D_{\text{off}}| = O(pd\Delta)$, and computing all intervals takes $O(pd\Delta(d\Delta))$ time. ◀

Borůvka's Algorithm

We now discuss how to compute the partition $\mathcal{R}$ of $\mathcal{I}$. We will merge the intervals by a Borůvka's styled algorithm. The algorithm has several *phases*, and each phase j receives a partition $\mathcal{R}^{(j)}$ of $\mathcal{I}$ as input. each group $R \in \mathcal{R}^{(j)}$ is either *active* or *inactive*. Initially, $\mathcal{R}^{(1)} = \{\{I\} \mid I \in \mathcal{I}\}$ is the trivial partition of $\mathcal{I}$ and all groups in $\mathcal{R}^{(1)}$ are active. For each phase j, we do the following to update $\mathcal{R}^{(j)}$ to $\mathcal{R}^{(j+1)}$.

1. For each active group R in $\mathcal{R}^{(j)}$, we will ask the following *adjacency query*.
   (Q1) Given an active group $R \in \mathcal{R}^{(j)}$, find another active group $R' \in \mathcal{R}^{(j)}$ s.t. there exists an edge $e = \{u, v\} \in E(G^\star)$ with $u \in I_u \in R$ and $v \in I_v \in R'$, or claim that there is no such R'.

   After asking (Q1) for all active groups, for each active group R, if (Q1) tells that no such R' exists, we mark R as an inactive group, otherwise we find an *adjacent group-pair* $\{R, R'\}$.
2. Given the adjacent group-pairs in step 1, we construct a graph K with vertices corresponding to active groups and edges corresponding to adjacent group-pairs. Note that for each adjacent group-pair $\{R, R'\}$, R and R' must still be active. Then, for each connected component of K, we merge the groups inside it into a new active group.

The algorithm terminates once it reaches a phase $\bar{j}$ s.t. all groups in $\mathcal{R}^{(\bar{j})}$ are inactive, and we let $\mathcal{R} = \mathcal{R}^{(\bar{j})}$ be the final output. Obviously, $\mathcal{R}$ satisfies the output requirement of Lemma 19. Furthermore, the number of phases is bounded in Proposition 21. Let $\mathcal{R}^{(j)}_{\text{act}} \subseteq \mathcal{R}^{(j)}$ denote the active groups in $\mathcal{R}^{(j)}$ at the moment when phase j starts and let $\bar{k}^{(j)} = |\mathcal{R}^{(j)}_{\text{act}}|$.

▶ **Proposition 21.** *For each $j \geq 2$, $\bar{k}^{(j)} \leq \bar{k}^{(j-1)}/2$. The number of phases is $O(\log |\mathcal{I}|)$.*

Proof. At each phase, the number of active groups is halved because we mark all old active groups without adjacent group inactive in step 1, and each connected component of the graph K in step 2 contains at least two old active groups. Because initially $\bar{k}^{(0)} = |\mathcal{R}^{(0)}| = |\mathcal{I}|$, the number of phases is $O(\log |\mathcal{I}|)$. ◀

Next, we will discuss the implementation of step 1. Basically, for each phase j, we need an algorithm that answers the *adjacency query* (Q1) efficiently. Instead of answering (Q1) directly, we will reduce (Q1) to the following *batched adjacency query* (Q2). We give an arbitrary order to the groups in $\mathcal{R}^{(j)}_{\text{act}}$, denoted by $\mathcal{R}^{(j)}_{\text{act}} = \{R^{(j)}_1, R^{(j)}_2, ..., R^{(j)}_{\bar{k}^{(j)}}\}$.

(Q2) Given a group $R^{(j)}_k \in \mathcal{R}^{(j)}_{\text{act}}$ and a batch of consecutive groups $R^{(j)}_\ell, R^{(j)}_{\ell+1}, ..., R^{(j)}_r \in \mathcal{R}^{(j)}_{\text{act}}$ s.t. $k \notin [\ell, r]$, decide if there exists $R^{(j)}_{k'}$ s.t. $k' \in [\ell, r]$ and $R^{(j)}_{k'}$ is adjacent to $R^{(j)}_k$.

▶ **Lemma 22.** *At phase j, one adjacency query can be reduced to $O(\log \bar{k}^{(j)})$ batched adjacency queries.*

Proof. Consider an adjacency query for some $R_k^{(j)} \in \mathcal{R}_{\text{act}}^{(j)}$. We can either find some $R_{k'}^{(j)} \in \mathcal{R}_{\text{act}}^{(j)}$ s.t. $k+1 \le k' \le \bar{k}^{(j)}$ and $R_{k'}^{(j)}$ is adjacent to $R_k^{(j)}$ or claim there is no such $R_{k'}^{(j)}$ in the following way: first fix $\ell = k+1$, and then perform a binary search on r in range $[k+1, \bar{k}^{(j)}]$, in which each binary search step is guided by a batched adjacency query with parameters k, ℓ, r. Similarly, we can try to find an adjacent group $R_{k'}^{(j)}$ to the left of $R_k^{(j)}$ by fixing $r = k-1$ and performing a binary search on ℓ in range $[1, k-1]$. The total number of calls to (Q2) is obviously $O(\log \bar{k}^{(j)})$ in these two binary searches. ◀

To answer batched adjacency queries in each phase, we will first introduce some *additional structures*, and then use them to design the algorithm answering (Q2), which is formalized in Lemma 23.

▶ **Lemma 23.** *There is a deterministic algorithm that computes some additional structures in $O(p^2d^2\Delta^2 \log n)$ time to support any batched adjacency query in $O(pd)$ time.*

We are now ready to analyse the running time of the Borůvka's algorithm, which completes the proof of Lemma 19. At each phase j, the number of adjacency queries is at most $\bar{k}^{(j)}$ (one for each active group in $\mathcal{R}^{(j)}$), so the number of batched adjacency queries is $O(\bar{k}^{(j)} \log \bar{k}^{(j)})$ by Lemma 22. Thus the total number of batched adjacency queries is $\sum_{j\ge 1} O(\bar{k}^{(j)} \log \bar{k}^{(j)}) = O(|\mathcal{I}| \log |\mathcal{I}|)$ by Proposition 21. By Lemma 23, the total running time of step 1 is $O(p^2d^2\Delta^2 \log n) + O(pd|\mathcal{I}| \log |\mathcal{I}|) = O(p^2d^2\Delta^2 \log n)$. The total running time of the Borůvka's algorithm is asymptotically the same because step 2 takes little time.

In what follows, we prove Lemma 23.

The Additional Structures

We start with introducing some notations. For a group $R \subseteq \mathcal{I}$, we use $V(R) = \bigcup_{I\in R} I$ to denote its vertex set. For two disjoint groups $R_1, R_2 \subseteq \mathcal{I}$ and a (multi) set E of undirected edges, let $\delta_E(R_1, R_2)$ denote the number of edges in E with one endpoint in $V(R_1)$ and the other one in $V(R_2)$. Also, recall that we gave an order to groups in $\mathcal{R}_{\text{act}}^{(j)}$, denoted by $\mathcal{R}_{\text{act}}^{(j)} = \{R_1^{(j)}, \ldots, R_{\bar{k}^{(j)}}^{(j)}\}$.

For each phase j, we will construct the following data structures.

- First, we construct a two-dimensional $(\bar{k}^{(j)} \times \bar{k}^{(j)})$-array $\mathsf{CountAll}^{(j)}$, where for each $1 \le x, y \le \bar{k}^{(j)}$, the entry $\mathsf{CountAll}^{(j)}(x,y) = \delta_{E(\hat{G})}(R_x^{(j)}, R_y^{(j)})$. Furthermore, we store the 2D-prefix sum of $\mathsf{CountAll}^{(j)}$.
- For each affected component γ, we prepare a one-dimensional array $\mathsf{CountA}_\gamma^{(j)}$ with length $\bar{k}^{(j)}$, where for each $1 \le x \le \bar{k}^{(j)}$, the entry $\mathsf{CountA}_\gamma^{(j)}(x) = |A_\gamma \cap V(R_x^{(j)})|$. Similarly, we construct an one-dimensional array $\mathsf{CountB}_\gamma^{(j)}$ with length $\bar{k}^{(j)}$ in which the entry $\mathsf{CountB}_\gamma^{(j)}(x) = |B_\gamma \cap V(R_x^{(j)})|$. Furthermore, we store the prefix sum of $\mathsf{CountA}_\gamma^{(j)}$ and $\mathsf{CountB}_\gamma^{(j)}$.

▶ **Lemma 24.** *The total construction time of arrays $\mathsf{CountAll}^{(j)}$, $\mathsf{CountA}_\gamma^{(j)}$ and $\mathsf{CountB}_\gamma^{(j)}$ summing over all phases j and all affected components γ is $O(p^2d^2\Delta^2 \log d)$.*

Proof. We first initialize $\mathsf{CountAll}^{(1)}, \mathsf{CountA}_\gamma^{(1)}, \mathsf{CountB}_\gamma^{(1)}$ for phase 1. For each entry $\mathsf{CountAll}^{(1)}(x,y)$ of $\mathsf{CountAll}^{(1)}$, note that $R_x^{(1)}$ and $R_y^{(1)}$ are both singleton groups. Let I_x and I_y be the intervals in $R_x^{(1)}$ and $R_y^{(1)}$. Then $\mathsf{CountAll}^{(1)}(x,y)$ is exactly the number of $E(\hat{G})$-edges that connect I_x and I_y, which can be answered by querying Table in $O(\log n)$ time by Lemma 16 because I_x and I_y are intervals on the global order π. For an entry $\mathsf{CountA}_\gamma^{(1)}(x)$ of $\mathsf{CountA}_\gamma^{(1)}$, let I_x be the single interval in $R_x^{(1)}$, and we can easily compute

$|A_\gamma \cap I_x|$ by binary search in $O(\log n)$ time because I_x is an interval on π and A_γ is ordered consistently with π. Similarly, we can compute the array $\mathsf{CountB}_\gamma^{(1)}$. The construction time of additional structures at phase 1 is $O(((\bar{k}^{(1)})^2 + |\mathcal{C}_{\text{aff}}| \cdot \bar{k}^{(1)}) \log n)$.

For each phase $j \geq 2$, we will compute $\mathsf{CountAll}^{(j)}, \mathsf{CountA}_\gamma^{(j)}, \mathsf{CountB}_\gamma^{(j)}$ based on the arrays of phase $j-1$. For an entry $\mathsf{CountAll}^{(j)}(x, y)$ of $\mathsf{CountAll}^{(j)}$, recall that $R_x^{(j)}$ is the union of several groups $R_{x_1}^{(j-1)}, R_{x_2}^{(j-1)}, \dots$ inside $\mathcal{R}_{\text{act}}^{(j-1)}$, and $R_y^{(j)} = R_{y_1}^{(j-1)} \cup R_{y_2}^{(j-1)} \cup \dots$. Furthermore, $x_1, x_2, \dots, y_1, y_2, \dots$ are distinct indexes in $[1, \bar{k}^{(j-1)}]$. Therefore,

$$\mathsf{CountAll}^{(j)}(x, y) = \sum_{x'=x_1,x_2,\dots} \sum_{y'=y_1,y_2,\dots} \mathsf{CountAll}^{(j-1)}(x', y').$$

We can compute $\mathsf{CountA}_\gamma^{(j)}$ and $\mathsf{CountB}_\gamma^{(j)}$ in a similar way. The construction time of additional structures at phase j is proportional to the total size of additional structures at phase $j-1$, i.e. $O((\bar{k}^{(j-1)})^2 + |\mathcal{C}_{\text{aff}}| \cdot \bar{k}^{(j-1)})$.

The overall construction time is

$$O(((\bar{k}^{(1)})^2 + |\mathcal{C}_{\text{aff}}| \cdot \bar{k}^{(1)}) \log n) + \sum_{j \geq 2} O((\bar{k}^{(j-1)})^2 + |\mathcal{C}_{\text{aff}}| \cdot \bar{k}^{(j-1)}) = O(p^2 d^2 \Delta^2 \log n),$$

because $\bar{k}^{(1)} = |\mathcal{I}| = O(pd\Delta)$, $|\mathcal{C}_{\text{aff}}| = O(pd)$ and $\bar{k}^{(j)} \leq \bar{k}^{(j-1)}/2$ for each phase j. ◀

Answering Batched Adjacency Queries

Consider a batched adjacency query at phase j with parameters k, ℓ, r. It is equivalent to decide whether the number of $G^\star$-edges connecting $R_k^{(j)}$ and some $R_{k'}^{(j)}$ where $k' \in [\ell, r]$ is greater than zero or not. Namely, it suffices to decide whether

$$\sum_{\ell \leq k' \leq r} \delta_{E(G^\star)}(R_k^{(j)}, R_{k'}^{(j)}) > 0. \tag{1}$$

▶ **Lemma 25.** *For any two disjoint groups $R_1, R_2 \subseteq \mathcal{I}$,*

$$\delta_{E(G^\star)}(R_1, R_2) = \delta_{E(\hat{G})}(R_1, R_2) - \sum_{\gamma \in \mathcal{C}_{\text{aff}}} \delta_{\hat{E}_\gamma}(R_1, R_2).$$

Proof. First the RHS is equal to $\delta_{E(\hat{G}) - \sum_{\gamma \in \mathcal{C}_{\text{aff}}} \hat{E}_\gamma}(R_1, R_2)$ because $\hat{E}_\gamma$ of all $\gamma \in \mathcal{C}_{\text{aff}}$ are disjoint subsets of $E(\hat{G})$ (note that $E(\hat{G})$ is defined to be a multiset).

Recall that $G^\star = \hat{G}[Q^\star] - \sum_{\gamma \in \mathcal{C}_{\text{aff}}} \hat{E}_\gamma$. The LHS is at most the RHS because $E(G^\star) \subseteq E(\hat{G}) - \sum_{\gamma \in \mathcal{C}_{\text{aff}}} \hat{E}_\gamma$. On the other direction, each edge in $E(\hat{G})$ connecting $V(R_1)$ and $V(R_2)$ is inside $\hat{G}[Q^\star]$ since $V(R_1), V(R_2) \subseteq Q^\star$, so the RHS is at most the LHS. ◀

▶ **Lemma 26.** *For each $\gamma \in \mathcal{C}_{\text{aff}}$ and two disjoint groups $R_1, R_2 \subseteq \mathcal{I}$,*

$$\delta_{\hat{E}_\gamma}(R_1, R_2) = |A_\gamma \cap V(R_1)| \cdot |A_\gamma \cap V(R_2)| - |(A_\gamma \setminus B_\gamma) \cap V(R_1)| \cdot |(A_\gamma \setminus B_\gamma) \cap V(R_2)|.$$

Proof. Recall that $\hat{E}_\gamma$ is the union of a clique on B_γ and a biclique between $A_\gamma \setminus B_\gamma$ and B_γ. In other words, $\hat{E}_\gamma$ is a clique on A_γ with the clique on $A_\gamma \setminus B_\gamma$ removed. Because $V(R_1)$ and $V(R_2)$ are disjoint, the equation follows. ◀

Using Lemma 25 and Lemma 26, we can rewrite the LHS of inequality 1 as follows.

$$\begin{aligned}\sum_{\ell\le k'\le r}\delta_{E(G^\star)}(R_k^{(j)},R_{k'}^{(j)}) &= \sum_{\ell\le k'\le r}\delta_{E(\hat{G})}(R_k^{(j)},R_{k'}^{(j)}) - \sum_{\ell\le k'\le r}\sum_{\gamma\in\mathcal{C}_{\text{aff}}}\delta_{\hat{E}_\gamma}(R_k^{(j)},R_{k'}^{(j)})\\ &= \sum_{\ell\le k'\le r}\delta_{E(\hat{G})}(R_k^{(j)},R_{k'}^{(j)})\\ &\quad - \sum_{\gamma\in\mathcal{C}_{\text{aff}}}\sum_{\ell\le k'\le r}|A_\gamma\cap V(R_k^{(j)})|\cdot|A_\gamma\cap V(R_{k'}^{(j)})|\\ &\quad + \sum_{\gamma\in\mathcal{C}_{\text{aff}}}\sum_{\ell\le k'\le r}|(A_\gamma\setminus B_\gamma)\cap V(R_k^{(j)})|\cdot|(A_\gamma\setminus B_\gamma)\cap V(R_{k'}^{(j)})|.\end{aligned}$$

For convenience, we denote $\alpha_\gamma^{(j)}(k)=\mathsf{CountA}_\gamma^{(j)}(k)$, $\beta_\gamma^{(j)}(k)=\mathsf{CountB}_\gamma^{(j)}(k)$
Combining the definition of the additional structures, we further have

$$\begin{aligned}\sum_{\ell\le k'\le r}\delta_{E(G^\star)}(R_k^{(j)},R_{k'}^{(j)}) &= \sum_{\ell\le k'\le r}\mathsf{CountAll}^{(j)}(k,k') - \sum_{\gamma\in\mathcal{C}_{\text{aff}}}\left(\alpha_\gamma^{(j)}(k)\cdot\sum_{\ell\le k'\le r}\alpha_\gamma^{(j)}(k')\right)\\ &\quad + \sum_{\gamma\in\mathcal{C}_{\text{aff}}}\left((\alpha_\gamma^{(j)}(k)-\beta_\gamma^{(j)}(k))\cdot\sum_{\ell\le k'\le r}(\alpha_\gamma^{(j)}(k')-\beta_\gamma^{(j)}(k'))\right)\end{aligned}$$

Because we have stored the prefix sum of the arrays $\mathsf{CountAll}^{(j)}, \mathsf{CountA}_\gamma^{(j)}, \mathsf{CountB}_\gamma^{(j)}$, computing the value of the above expression takes $O(|\mathcal{C}_{\text{aff}}|)=O(pd)$ time.

References

1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014. `doi:10.1109/FOCS.2014.53`.

2 Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental SSSP and approximate min-cost flow in almost-linear time. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1000–1008. IEEE, 2021. `doi:10.1109/FOCS52979.2021.00100`.

3 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 Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1158–1167. IEEE, 2020. `doi:10.1109/FOCS46700.2020.00111`.

4 Ran Duan and Seth Pettie. Connectivity oracles for failure prone graphs. In *Proceedings of the forty-second ACM Symposium on Theory of Computing*, pages 465–474, 2010.

5 Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. *SIAM Journal on Computing*, 49(6):1363–1396, 2020.

6 Martin Furer and Balaji Raghavachari. Approximating the minimum-degree steiner tree to within one of optimal. *Journal of Algorithms*, 17(3):409–423, 1994.

7 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015. `doi:10.1145/2746539.2746609`.

8 Monika Henzinger and Stefan Neumann. Incremental and fully dynamic subgraph connectivity for emergency planning. In *24th Annual European Symposium on Algorithms (ESA 2016)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

9 Bingbing Hu, Evangelos Kosinas, and Adam Polak. Connectivity oracles for predictable vertex failures. *CoRR*, abs/2312.08489, 2023. doi:10.48550/arXiv.2312.08489.

10 Arkady Kanevsky, Roberto Tamassia, Giuseppe Di Battista, and Jianer Chen. On-line maintenance of the four-connected components of a graph. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 793–801. IEEE Computer Society, 1991.

11 Rohit Khandekar, Subhash Khot, Lorenzo Orecchia, and Nisheeth K Vishnoi. On a cut-matching game for the sparsest cut problem. *Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2007-177*, 6(7):12, 2007.

12 Rohit Khandekar, Satish Rao, and Umesh V. Vazirani. Graph partitioning using single commodity flows. *J. ACM*, 56(4):19:1–19:15, 2009. doi:10.1145/1538902.1538903.

13 Evangelos Kosinas. Connectivity queries under vertex failures: Not optimal, but practical. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPIcs*, pages 75:1–75:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ESA.2023.75.

14 Yaowei Long and Thatchaphol Saranurak. Near-optimal deterministic vertex-failure connectivity oracles. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1010. IEEE, 2022.

15 Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992. doi:10.1007/BF01758778.

16 Merav Parter, Asaf Petruschka, and Seth Pettie. Connectivity labeling and routing with multiple vertex failures. In *Proceedings 56th ACM Symposium on Theory of Computing (STOC)*, 2024.

17 Michal Pilipczuk, Nicole Schirrmacher, Sebastian Siebertz, Szymon Torunczyk, and Alexandre Vigny. Algorithms and data structures for first-order logic with connectivity under vertex failures. In 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 102:1–102:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICALP.2022.102.

18 Jan van den Brand and Thatchaphol Saranurak. Sensitive distance and reachability oracles for large batch updates. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 424–435. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00034.