Spanning Adjacency Oracles in Sublinear Time

Greg Bodwin
Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor, MI, USA

Henry Fleischmann
Department of Pure Mathematics and Mathematical Statistics,
University of Cambridge, UK

Abstract

Suppose we are given an n-node, m-edge input graph G, and the goal is to compute a spanning subgraph H on $O(n)$ edges. This can be achieved in linear $O(m+n)$ time via breadth-first search. But can we hope for *sublinear* runtime in some range of parameters – for example, perhaps $O(n^{1.9})$ worst-case runtime, even when the input graph has n^2 edges?

If the goal is to return H as an adjacency list, there are simple lower bounds showing that $\Omega(m+n)$ runtime is necessary. If the goal is to return H as an adjacency matrix, then we need $\Omega(n^2)$ time just to write down the entries of the output matrix. However, we show that neither of these lower bounds still apply if instead the goal is to return H as an *implicit* adjacency matrix, which we call an *adjacency oracle*. An adjacency oracle is a data structure that gives a user the illusion that an adjacency matrix has been computed: it accepts edge queries (u, v), and it returns in near-constant time a bit indicating whether or not $(u, v) \in E(H)$.

Our main result is that, for any $0 < \varepsilon < 1$, one can construct an adjacency oracle for a spanning subgraph on at most $(1+\varepsilon)n$ edges, in $\widetilde{O}(n\varepsilon^{-1})$ time (hence sublinear time on input graphs with $m \gg n$ edges), and that this construction time is near-optimal. Additional results include constructions of adjacency oracles for k-connectivity certificates and spanners, which are similarly sublinear on dense-enough input graphs.

Our adjacency oracles are closely related to Local Computation Algorithms (LCAs) for graph sparsifiers; they can be viewed as LCAs with some computation moved to a preprocessing step, in order to speed up queries. Our oracles imply the first LCAs for computing sparse spanning subgraphs of general input graphs in $\widetilde{O}(n)$ query time, which works by constructing our adjacency oracle, querying it once, and then throwing the rest of the oracle away. This addresses an open problem of Rubinfeld [CSR '17].

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Graph algorithms, Sublinear algorithms, Data structures, Graph theory

Digital Object Identifier 10.4230/LIPIcs.ITCS.2024.19

Related Version *Full Version*: `https://arxiv.org/abs/2308.13890`

Funding *Greg Bodwin*: This work was supported by NSF:AF 2153680.

Acknowledgements We are grateful to Merav Parter, Nathan Wallheimer, Amir Abboud, Ron Safier, and Oded Goldreich for references and technical discussions on the paper. We are also grateful to an anonymous reviewer for exceptionally helpful and thorough comments.

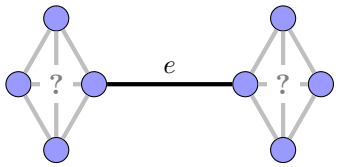
1 Introduction

A *sparsifier* of a graph G is a smaller graph H that approximately preserves some important structural properties of G. Examples include spectral sparsifiers, flow/cut sparsifiers, spanners, preservers, etc. Let us focus for now on the computation of a particularly simple kind of sparsifier, which we call a *sparse spanning subgraph*:

`Sparse Spanning Subgraph` (SSS):
- **Input:** An n-node, m-edge, undirected graph $G = (V, E)$ and $\varepsilon > 0$.
- **Output:** An edge-subgraph H on at most $(1 + \varepsilon)n$ edges that spans G.

An SSS is a slightly relaxed version of a spanning forest,[1] and so this problem can be solved in linear $O(m+n)$ time via breadth-first search (BFS). We might wonder for a moment whether linear runtime for this basic algorithm is *optimal*. Especially for denser input graphs, we might dream of *sublinear-time* algorithms, which try to discover one of the many valid spanning subgraphs without even reading most of the input graph. Can we solve SSS in, say, $O(n^{1.9})$ worst-case time, even when the input graph has $\Theta(n^2)$ edges?

Unfortunately, the canonical answer is *no*: we cannot hope to solve SSS without reading at least a constant fraction of the input graph. The counterexamples generally work by planting a cut edge into an otherwise-random graph. For example, in a lower bound construction from [16, 23], we construct G from two node-disjoint random graphs G_1, G_2 on $n/2$ nodes each, which include each possible edge independently with probability $1/2$, plus a single "cut edge" e connecting a random node from G_1 to a random node from G_2. We absolutely must take e in the spanning subgraph H, but we need to scan essentially the entire input graph just to find e.



The starting point of this paper is that the cut-edge lower bound, while formidable, is actually a bit restricted in scope: it does not quite apply in all graph representation models. We explain this next.

1.1 Adjacency Oracles and Algorithms for SSS

There are two popular ways to represent graphs:

1. First, the **adjacency list representation** is an array of arrays. If we index into an adjacency list L with a node v, then $L[v]$ returns a list of the neighbors of v in some arbitrary order.[2]
2. Second, the **adjacency matrix representation** is an $n \times n$ matrix, which holds a 1 or 0 in each position (u, v) to represent whether (u, v) is or is not an edge in the graph.

The cut-edge lower bound for SSS shows that we cannot hope to return a spanning subgraph H *as an adjacency list* in sublinear time. For adjacency matrices, the situation is even worse: it takes $\Omega(n^2)$ time just to fill out the entries of the matrix, so no sublinear algorithms are possible. However, this paper will consider a tweak on the adjacency matrix model:

[1] An edge-subgraph H *spans* a graph G if it has the same connected components as G. A *spanning forest* of G is any forest that spans G, i.e., it is the union of spanning trees for each connected component of G.

[2] We assume for this discussion that the list is given in a form where we can check its length (corresponding to $\deg(v)$) and where we can query a random neighbor, both in $\widetilde{O}(1)$ time.

▶ **Definition 1** (Adjacency Oracles). *An implicit adjacency matrix for a graph G, which we will call an* ***adjacency oracle****, is a data structure that, on query (u, v), deterministically returns a bit in $\widetilde{O}(1)$ time*[3] *indicating whether or not $(u, v) \in E(G)$.*

The determinism in queries is essential to force the oracle to be *history-independent*; that is, the graph G that the oracle represents must be fixed at the end of the construction phase, and it may not depend on the order in which the oracle receives queries. The point of the adjacency oracle model is that it is not constrained by the $\Omega(n^2)$ output size lower bound – the data structure could, in principle, have $\ll n^2$ bits – and for more subtle reasons, it escapes the cut-edge lower bound as well. The hard part of the cut-edge lower bound is scanning to find the cut edge e, but when we build an adjacency oracle it is the *user* who does the hard work of pointing out the edge e. In other words, an adjacency oracle needs to quickly *recognize* that the cut-edge e is necessary for the spanning subgraph when it is received at query time, but this is potentially an easier task than proactively *discovering* e at preprocessing time.

Given this, we can now reopen the question of whether BFS is the most efficient algorithm to solve SSS, when we allow the output subgraph to be represented by an adjacency oracle. Our first main result is that, in fact, it is not.

▶ **Theorem 2** (Adjacency Oracles for SSS).

- ***(Upper Bound)*** *For any $0 < \varepsilon < 1$ and n-node input graph, there is a randomized algorithm that solves SSS with high probability in $\widetilde{O}(n\varepsilon^{-1})$ time, where the output subgraph H is returned as an adjacency oracle.*
- ***(Lower Bound)*** *However, no algorithm as above can run in $O(n^{1-\delta})$ time, for any constant $\delta > 0$.*

As a point of clarification, we assume that the input graph G is received in all useful forms, i.e., as both an adjacency list and an adjacency matrix/oracle. The adjacency oracle for H is allowed access to the adjacency oracle for G, so that if queried with an edge $(u, v) \notin E(G)$ it can quickly say `NO`. However, we note that this is the *only* way in which our adjacency oracles use access to G, and so they still works in a slightly stronger model where the adjacency oracle for H may not access G in any way but the user promises a priori to only query the oracle with edges from $E(G)$.

This model, in which we obtain sublinear algorithms by providing near-constant-time oracle query access to the output (and where these queries may access the input), is the typical one in sublinear algorithms. It is sometimes called a *solution oracle*, and it has been used previously as a paradigm for sublinear algorithms for vertex cover, dominating set, maximum matching, independent set, and others [30, 9, 22, 31]. It is also analogous to the model used for Local sparsifier algorithms, which we will discuss in detail shortly. Indeed, the lower bound part of Theorem 2 uses a graph construction and analysis developed in the context of Local sparsifier algorithms for most of its heavy lifting [23, 16].

1.2 Adjacency Oracles for k-Connectivity Certificates

A natural generalization of a sparse spanning subgraph is a *k-connectivity certificate*:

▶ **Definition 3** (k-Connectivity Certificates). *Given a graph G, a subgraph H is a k-connectivity certificate if, for any edge set $F \subseteq E(G), |F| \leq k - 1$, the connected components of $G \setminus F$ and $H \setminus F$ are identical.*

[3] Here and throughout the paper, $\widetilde{O}(\cdot)$ notation hides polylog(n) factors.

The following are two equivalent definitions of k-connectivity certificates often used in the literature.

- For any integer r and any cut C in G of size r, the size of C in H is at least $\min\{k, r\}$.
- For any integer r and nodes s, t that are r-connected in G, they are at least $\min\{k, r\}$-connected in H.

A 1-connectivity certificate is the same as a spanning subgraph, and in this sense the problem of constructing sparse k-connectivity certificates generalizes SSS. For this reason, we refer to the problem as k-SSS, with 1-SSS and SSS identical. The size bounds for k-connectivity certificates also extend those for spanning forests:

▶ **Theorem 4** ([21]). *Every n-node graph G has a k-connectivity certificate H of size $|E(H)| \leq k(n-1)$.*

We show:

▶ **Theorem 5** (Adjacency Oracles for k-Connectivity Certificates). *For any $\varepsilon > 0$, n-node input graph G, and $k = \widetilde{O}(1)$, there is a randomized algorithm that with high probability computes an adjacency oracle for a k-connectivity certificate $H \subseteq G$ of size $|E(H)| \leq (1+\varepsilon)kn$, and which runs in $\widetilde{O}(n\varepsilon^{-1})$ time.*

1.3 Adjacency Oracles vs. Local Sparsifier Algorithms

We will temporarily pause discussion of our results to discuss their relationship to Local Computation Algorithms (LCAs) for graph sparsifiers, which are the conceptually closest prior work to ours. LCAs were introduced in a classic paper by Rubinfeld, Tamir, Vardi, and Xie [28]. The high-level goal is to design an algorithm which, on input (X, i), computes the i^{th} part of the output associated to input X, ideally in sublinear time. Interesting LCAs have been discovered for many problems, such as graph coloring, SAT solving, graph sparsifiers, etc. [2, 29, 30, 7, 10, 17, 18, 26, 20, 19]; for more, see the survey of Levi and Medina [13].

An LCA algorithm for SSS would have input (G, ε, e), where $e \in E(G)$, and it would return either `YES` or `NO` in such a way that for any fixed (G, ε) the set of edges e to which the algorithm says `YES` form a valid solution to SSS. As in our setting, the input graph G is typically given to an LCA as both an adjacency list and an adjacency oracle. The main technical difference between adjacency oracles and LCAs for SSS is essentially whether the focus is placed on preprocessing or query time. That is:

- LCAs do not allow any centralized preprocessing before answering queries (G, ε, e) (with the minor exception that they typically allow shared randomness across queries). On the other hand, adjacency oracles for SSS allow centralized preprocessing, and the goal of the problem is to minimize this preprocessing time.
- Adjacency Oracles insist on $\widetilde{O}(1)$ query time in order to provide the illusion that we are indexing into an adjacency matrix. On the other hand, for LCAs the query time may be much larger, and the goal of the problem is to minimize this query time.

Thus, adjacency oracles might be viewed as an investigation of the extent to which a preprocessing phase can improve the query time for LCAs. These differences are enough to formally separate the two models. Levi, Ron, and Rubinfeld [16] proved that any LCA for SSS requires $\Omega(n^{1/2})$ query time. Together with Theorem 2, this implies a strong separation, i.e., LCAs cannot achieve $\widetilde{O}(1)$ query time as in the adjacency oracles of Theorem 2 (which bypass the lower bound of [16] due to their use of centralized preprocessing). An even stronger form of this lower bound was later proved by Parter, Rubinfeld, Vakilian, and

Yodpinyanee [23]. There is also a natural way to interpolate between the adjacency oracle and LCA models – allowing super-constant preprocessing and query time and investigating the tradeoff between them – but we will leave this as a possible direction for future work.

The optimal runtime of LCAs for SSS is a fascinating open problem; we refer to the excellent survey of Rubinfeld [27] for an in-depth presentation. In particular, this survey highlights the following two open questions:

▶ **Open Question** (cf. [27], Problem 1). *It is known that every graph with constant maximum degree or high expansion has an LCA for SSS in $O(n)$ query time [16, 14, 12, 15]. Is there an LCA algorithm for SSS that works on any sparse input graph ($O(n)$ edges), and which runs in $o(n)$ query time?*

▶ **Open Question** (cf. [27], Section 4). *Rubinfeld writes that* "nothing is known [about LCAs for SSS] when there is no bound on the maximum degree of the input graph." *Subsequent work [4, 23, 12] discovered algorithms for general graphs of maximum degree Δ with query complexity $\widetilde{O}(n^{2/3} \cdot poly\Delta)$, but this is still nontrivial only when Δ is a small-enough polynomial in n.*

Our new adjacency oracles partially address these questions:

▶ **Corollary 6.** *There is an LCA for SSS that runs in $\widetilde{O}(n\varepsilon^{-1})$ query time. (This algorithm works for any input graph.)*

Proof. On query (u, v), use Theorem 2 to construct an adjacency oracle for SSS in $\widetilde{O}(n\varepsilon^{-1})$ time (using a shared tape of random bits across all possible queries). Then query the adjacency oracle on (u, v) and return the result. ◀

Corollary 6 positively addresses the latter open question. Although it does not resolve the former open question, it may shed light on a way forward: (1) our algorithm does not use the assumption that the input graph is sparse, and (2) our algorithm performs *the same* $\widetilde{O}(n)$ work for each query, to compute the adjacency oracle, and then only the last $\widetilde{O}(1)$ steps where the adjacency oracle is accessed differ between queries. It seems unlikely that the optimal LCA for SSS would be so query-independent, and so perhaps improved LCAs could be achieved by exploiting knowledge of the query (u, v) at an earlier stage of the construction.

Our adjacency oracle upper bounds for SSS and k-SSS are not technically similar to prior work on LCAs. However, the rest of this work connects to the LCA literature at a technical level as well, and draws on several clever constructions and techniques developed in the context of LCAs. In particular, our lower bound for SSS adjacency oracles uses the lower bound construction from [16] as a black box, and our results on adjacency oracles for spanners (which we discuss next) draw on ideas used in LCAs for graph spanners [4].

1.4 Adjacency Oracles for Graph Spanners

Besides sparse spanning subgraphs, one can more strongly ask for *spanners*, which preserve approximate distances rather than just connectivity.

▶ **Definition 7** (Spanners [25, 24]). *Given a graph G, a k-spanner is an edge-subgraph H satisfying $\texttt{dist}_H(s,t) \leq k \cdot \texttt{dist}_G(s,t)$ for all nodes $s, t \in V$.*

We show the following two results for computing spanners:

▶ **Theorem 8** (Adjacency Oracles for 3-Spanners). *For any n-node input graph, there is a randomized algorithm that with high probability computes an adjacency oracle for a 3-spanner H of size $|E(H)| = \widetilde{O}(n^{3/2})$, and which runs in $\widetilde{O}(n^{3/2})$ time.*

▶ **Theorem 9** (Adjacency Oracles for 5-Spanners). *For any n-node input graph, there is a randomized algorithm that with high probability computes an adjacency oracle for a 5-spanner H of size $|E(H)| = \widetilde{O}(n^{4/3})$, and which runs in $\widetilde{O}(n^{3/2})$ time.*

Both of these results are proved only for unweighted input graphs. The sizes of these 3- and 5-spanners are optimal, up to hidden log factors, and can be viewed as sublinear computation for input graphs on $\gg n^{3/2}$ edges.

The reason that we have results for 3- and 5-spanners, but not spanners of higher stretch, is due to a common technical barrier. These results follow a construction strategy of Baswana and Sen [5], based on hierarchical clustering; 3- and 5-spanners can be achieved using only one level of clustering, while higher-stretch spanners of optimal size require two or more levels of clustering. We are able to optimize the first cluster assignment step, but it is unclear how to achieve sublinear time cluster assignment for depth 2 and beyond.

Nonetheless, by introducing edge-sampling to the algorithm of Baswana and Sen,we show a construction for higher-stretch spanners, with a parameter ρ controlling the tradeoff between size and preprocessing time. Our algorithm runs in a factor of $1/\rho$ less than linear time $\widetilde{O}(m)$ time by incurring a factor of ρ^2 edges over the optimal size (as usual, this optimality is conditional on the girth conjecture [6]).

▶ **Theorem 10** (Adjacency Oracles for $(2k-1)$-spanners). *For any n-node, m-edge input graph and $k = \widetilde{O}(1)$, there exists a randomized algorithm that with high probability computes an adjacency oracle for a $(2k-1)$-spanner H of size $|E(H)| = \widetilde{O}(n^{1+1/k}\rho^2)$, and which runs in $\widetilde{O}(n + m/\rho)$ time.*

Our construction leaves open the question of whether adjacency oracles of optimal size spanners can be computed in sublinear time.

▶ **Open Question.** *For $k > 3$, is there a randomized algorithm that computes the adjacency oracle of an $\widetilde{O}(n^{1+1/k})$ size $(2k-1)$-spanner in polynomially subquadratic time?*

This roughly mirrors a difficulty faced in the corresponding local spanner algorithms, where optimal results were recently achieved for 3- and 5-spanners in a nice paper by Arviv, Chung, Levi, and Pyne [4], but where results for higher-stretch spanners are considerably more restricted in scope. Indeed, [4] is also based on Baswana-Sen clustering, and it introduces a helpful degree-bucketing technique that we adapt and refine for our setting.

2 Spanning Adjacency Oracles

2.1 Adjacency Oracles for Sparse Spanning Subgraphs

We now construct our adjacency oracle for the problem SSS, defined in the introduction.

Preprocessing Algorithm

We first describe the data structures that we create in the construction of the adjacency oracle. We will use three data structures:

- Any kind of set data structure, to represent the edges added to the subgraph H during preprocessing. This data structure only needs to support insertions and queries (checking whether or not an edge (u, v) has been previously inserted to $E(H)$) in $\widetilde{O}(1)$ time each. Many data structures are available that achieve this behavior, e.g., a self-balancing binary search tree suffices.

- We use a union-find data structure to maintain connected components $\{C_i\}$. Initially, each node is in its own connected component, but we will iteratively merge components throughout the algorithm. This requries $\widetilde{O}(n)$ total time.
- Another data structure is used to perform a particular edge-sampling step in preprocessing; it will be easier to describe this in Lemma 11 following our description of the algorithm itself.

We will say that a node v is "in bucket b" at a moment in the algorithm if the connected component C_v containing v currently has size $2^b \leq |C_v| < 2^{b+1}$. So, initially every node is in bucket 0, and nodes may be promoted to higher buckets when their components are merged with other components.

SSS Adjacency Oracle Preprocessing

- Let $b \leftarrow 0$. This is an incremental counter that marks the current bucket we are processing.
- While $b < \log n$:
  - Let $E_b \subseteq E(G)$ be the set of edges for which at least one of the two endpoints is in bucket b. Choose an edge $(u, v) \in E_b$ uniformly at random. (Taking a uniform-random sample from E_b in $\widetilde{O}(1)$ time is nontrivial; we address this in Lemma 11 below.)
  - Let u be the endpoint of the sampled edge (u, v) that is in bucket b. If v is currently in a different connected component than u, and also v is in a bucket $b' \geq b$, then we call this sample a *success*. Otherwise, if u, v are in the same component or if $b' < b$, then we call this sample a *failure*.
    - If the sample is a failure, do nothing.
    - If the sample is a success, add the edge (u, v) to H, and merge the connected components containing u and v. Note that this increases the bucket of v (and possibly also the bucket of u), and so it changes the set of nodes in bucket b, and therefore also the set of edges in E_b.
  - Repeat until we have $c\varepsilon^{-1}2^b \log^2 n$ failure events in a row (where $c > 0$ is a sufficiently large absolute constant that we leave implicit). Then set $b \leftarrow b + 1$, i.e., we move on to analyzing the next bucket.

We will address the step of sampling uniformly from E_b, drawing from similar work on dynamic weighted sampling [8].

▶ **Lemma 11.** *For each bucket index b, we can maintain a data structure that allows us to sample uniformly from the edges currently in E_b in $\widetilde{O}(1)$ time with high probability,*[4] *using $\widetilde{O}(n)$ total update time.*

Proof. First we describe the creation and maintenance of the data structure. When we begin analyzing bucket index b, we first scan all nodes and make a list of the nodes in bucket b as well as their degrees. We sort these nodes into $\log n$ *groups* $\{\Gamma_1, \ldots, \Gamma_{\log n}\}$ by their degrees: group Γ_i contains the nodes in bucket b whose degree falls in the range $[2^i, 2^{i+1})$. We will also maintain the sum of node degrees in each group, which we will write as $\deg(\Gamma_i)$. It takes $\widetilde{O}(n)$ time to create these groups at the beginning of our analysis of bucket b, and as nodes leave bucket b due to connected component merges, we can straightforwardly remove them from the corresponding group and update our size and degree counts in $\widetilde{O}(1)$ time per node.

[4] Here and throughout the paper, "high probability" means probability $\geq 1 - 1/n^c$ for an absolute constant $c > 0$.

Now we describe the sampling algorithm. We execute the following process:

- Choose a group Γ_i with probability proportional to $\deg(\Gamma_i)$, i.e., $\deg(\Gamma_i)/\sum_j \deg(\Gamma_j)$.
- Choose a node $u \in \Gamma_i$ uniformly at random. With probability $\deg(u)/2^{i+1}$ we accept u and move on to the next step; otherwise, repeat this step, selecting a new node $u \in \Gamma_i$ uniformly at random.
- Choose an edge (u, v) incident to the node u selected in the previous step, uniformly at random. Then, check whether the other endpoint v is in bucket b. If not, return (u, v) as the sampled edge. If so, then with probability $1/2$ return (u, v) as the sampled edge, and with probability $1/2$ go back to the beginning of the entire sampling process and repeat from scratch.

Note that we accept our node sample in the second step with probability at least $1/2$, and so with high probability we repeat this step at most $\widetilde{O}(1)$ times. Similarly, in the third step we return an edge with probability at least $1/2$, so with high probability we restart the entire sampling process only $\widetilde{O}(1)$ times. Together, this implies that each sample runs in $\widetilde{O}(1)$ time with high probability.

Finally, we argue that this process selects a uniform-random edge from E_b. Let E_b^* be the set of *oriented* edges $(u, v) \in E(G)$ such that the first node u is in bucket b (if v is also in bucket b, then we have both $(u, v), (v, u) \in E_b^*$). Notice that the first two steps of the sampling procedure, combined with the first part of the third step, select a uniform-random oriented edge from E_b^*. This is because we select a node u with probability proportional to $\deg(u)$ (the number of edges in E_b^* that start with u), and then we select a uniform-random edge incident to u.

This means that an edge (u, v) with both u, v in bucket b is twice as likely to be selected in the third round as an edge with only u in bucket b, since such an edge is represented twice (once with each orientation) in E_b^*. The third step resamples these edges with probability $1/2$, so after this step is applied, all edges in E_b are equally likely to be sampled. ◀

We are now ready to verify the runtime of the preprocessing algorithm:

▶ **Lemma 12.** *The above preprocessing algorithm can be implemented to run in $\widetilde{O}(n\varepsilon^{-1})$ time (with high probability).*

Proof. The union-find data structure has total update time $\widetilde{O}(n)$. We can record the edges added to H in $\widetilde{O}(1)$ time each by tracking these edges using any kind of set data structure; we can have at most $n - 1$ success events (since each success event causes two components to be merged), and so this takes $\widetilde{O}(n)$ time in total.

We next control the number of samples that we take for each of the $\log n$ choices of bucket index b. At the beginning of a round with bucket index b, there are at most $n/2^b$ components in bucket b, since each component has size at least 2^b. Each success event causes at least one component to leave the bucket. There are at most $\widetilde{O}(\varepsilon^{-1}2^b)$ failure events between success events. Thus, we sample at most $\widetilde{O}(n\varepsilon^{-1})$ times per bucket index.

Finally, by the previous lemma we can sample from E_b in $\widetilde{O}(1)$ time with high probability, by paying $\widetilde{O}(n)$ additional runtime per bucket index. ◀

We next look ahead to the step where we bound the number of edges in the subgraph H represented by our adjacency oracle. The following property of the preprocessing algorithm will be helpful:

▶ **Lemma 13.** *With high probability, each time we finish processing a bucket b (i.e. just before we set $b \leftarrow b + 1$), there are at most $\varepsilon n/\log n$ edges $(u, v) \in E(G)$ with the property that u is in bucket b, v is in a different connected component than u, and the bucket b' of v satisfies $b' \geq b$.*

Proof. Let Y_b be the event that the lemma statement holds for bucket index b. Our first goal is to bound

$$\Pr\left[Y_b \mid Y_1 \text{ and } \dots \text{ and } Y_{b-1}\right].$$

To do so, we again let E_b^* be the set of oriented edges (u, v) with u in bucket b. We sort these edges into three types:

- We say that (u, v) is a **success edge** if v is in a bucket $b' \geq b$, and u, v are in different connected components. (That is, sampling a success edge causes a success event, and the goal of this lemma is to bound the number of success edges.)
- We say that (u, v) is a **descending edge** if v is in a bucket $b' < b$.
- We say that (u, v) is an **internal edge** if u, v are in the same connected component.

The number of internal edges can be bounded as

$$\begin{aligned}\sum_{X \text{ component in bucket } b} |X|^2 &\leq \frac{n}{2^b} \cdot \left(2^{b+1}\right)^2 \\ &= O\left(n \cdot 2^b\right).\end{aligned}$$

To bound the number of descending edges (u, v), note that each such edge would be a success edge for a smaller bucket index if we considered it with reversed orientation (v, u). Since we have conditioned on $Y_1, \dots, Y_{b-1}$, we assume that the lemma statement holds for all bucket indices $i < b$, we can bound the number of descending edges as

$$\sum_{i<b} \frac{\varepsilon n}{\log n} \leq \varepsilon n.$$

So, if there are currently at least $\varepsilon n / \log n$ success edges for bucket b, then each time we sample an edge (u, v) the probability that it causes a success event is at least

$$\Theta\left(\frac{\varepsilon n}{2^b \cdot n \log n}\right) = \Theta\left(\frac{\varepsilon}{2^b \cdot \log n}\right).$$

So by standard Chernoff bounds and by choice of large enough constant c, with probability at least $1 - 1/n^{100}$, if we sample $c\varepsilon^{-1} 2^b \log^2 n$ edges we will have at least one success event. Since (conservatively) we will have at most n^2 edges recorded over the course of the algorithm and hence $O(n^2)$ periods of samples between successes, by a union bound, with probability at least $1 - 1/n^{98}$ we will not have $c\varepsilon^{-1} 2^b \log^2 n$ failure events in a row while $\geq \varepsilon n / \log n$ success edges for bucket b still exist. Assuming this event occurs, since we only move on to the next bucket $b + 1$ following $c\varepsilon^{-1} 2^b \log^2 n$ failure events in a row, it must be that $< \varepsilon n / \log n$ success edges remain.

To finish the proof, it now remains only to assemble the previous probability calculations. We bound

$$\begin{aligned}\Pr[Y_1 \text{ and } \dots \text{ and } Y_{\log n}] &= \Pr[Y_1] \cdot \Pr[Y_2 \mid Y_1] \cdot \dots \cdot \Pr[Y_{\log n} \mid Y_1 \text{ and } \dots \text{ and } Y_{\log n - 1}] \\ &\geq \left(1 - 1/n^{98}\right)^{\log n} \\ &\geq 1 - \Theta\left(1/n^{97}\right).\end{aligned}$$

◀

The Query Algorithm

The query algorithm is relatively simple. As discussed in the introduction, this algorithm is phrased in the stronger model where the query algorithm may not access the input graph G, but the user promises to only give edges $(u, v) \in E(G)$ as queries. When we receive an edge query (u, v), we process it as follows.

SSS Adjacency Oracle Query

- If (u, v) was recorded by the set data structure as one of the edges added to $E(H)$ during preprocessing, then answer `YES`.
- Else if u, v are in different connected components according to the union-find data structure, then answer `YES`.
- Else answer `NO`.

▶ **Lemma 14.** *The set of edges to which the adjacency oracle answers `YES` spans the input graph G.*

Proof. There are two kinds of edges to which the oracle answers `YES`. Some are the edges added to $E(H)$ during preprocessing. By construction, these edges span each individual connected component in the data structure at the end of preprocessing. The query algorithm will then answer `YES` to all additional edges in $E(G)$ between these connected components, and the lemma follows. ◀

▶ **Lemma 15.** *With high probability, the query algorithm will only answer `YES` to at most $(1 + \varepsilon)n$ edges in total.*

Proof. First, the edges added to $E(H)$ during preprocessing form a forest, so there are at most $n - 1$ such edges. Second, by Lemma 13, with high probability, for each bucket b there are at most $\varepsilon n / \log n$ edges between the connected components discovered in preprocessing that have one endpoint in bucket b and the other endpoint in a bucket $b' \geq b$. Summing over the $\log n$ buckets, the number of edges between connected components is at most $\log n \cdot \frac{\varepsilon n}{\log n} = \varepsilon n$. Thus, the oracle says `YES` to at most $(n - 1) + \varepsilon n \leq (1 + \varepsilon)n$ edges in total. ◀

Alternate Proof Using Random k-Out Orientations

After this paper was initially released, we realized that our adjacency oracle for SSS can also be proved as a corollary of a nice recent paper by Holm, King, Thorup, Zamir, and Zwick [11]. Their main result is the following structural theorem:

▶ **Theorem 16** ([11])**.** *Let G be any undirected unweighted graph, and let H be a random edge-subgraph obtained by selecting k edges incident to each node uniformly at random and including them in H. Then, if $k \geq c \log n$ for a constant c, then the expected number of edges between connected components of H is $O(n/k)$.*

It follows that we can construct an adjacency oracle that realizes Theorem 2 by sampling $k = \widetilde{O}(1)$ many edges incident to each node and memorizing a spanning forest of the resulting subgraph H. There will be $O(n/k) \ll \varepsilon n$ many edges between components, and the rest of the proof follows by the same analysis as above. In fact, our proof above can be *interpreted* as an alternate proof of a result similar to the theorem of [11]: one can argue that with high probability, our algorithm will query only $\widetilde{O}(1)$ many edges incident to each vertex, and we have proved that $\leq O(\varepsilon n)$ edges then go between the discovered connected components. Our proof is qualitatively different from the one in [11]; subjectively it is a bit simpler, but it loses several log factors that are optimized out in this prior work.

2.2 Adjacency Oracles for Sparse k-Connectivity Certificates

Our method for constructing adjacency oracles for sparse spanning subgraphs also extends to *k-connectivity certificates*. Recall that an edge subgraph $H \subseteq G$ is a k-(edge) connectivity certificate of G if, after deleting any $(k - 1)$ edges in H from both G and H, the connected components of H and G are the same. Then, there is a simple generalization of SSS.

`Sparse` k-`Connectivity Certificate` (k-SSS):
- **Input:** An n-node undirected graph $G = (V, E)$, and $\varepsilon > 0$.
- **Output:** An edge-subgraph H on at most $(1+\varepsilon)kn$ edges that forms a k-connectivity certificate of G.

(We recall from the introduction that 1-SSS and SSS are the same problem.) Due to space constraints, we will defer our construction and analysis to the full version of the paper.

3 Spanning Adjacency Oracle Lower Bounds

In this section we show two lower bounds. First we show that no algorithm with $o(n^{1-\varepsilon})$ preprocessing time can compute a spanning subgraph adjacency oracle on a linear number of edges. This lower bounds hold even when the adjacency oracles have access to the adjacency list and adjacency oracle of the input graph. This is the *weaker* model from the introduction.

Second, we show that no randomized algorithm with $o(n)$ preprocessing time can output a spanning subgraph adjacency oracle on even $o(n^2)$ edges in general. The caveat for this stronger result is that it applies in a slightly weaker model: we assume that the adjacency oracle does not have access to the adjacency oracle or adjacency list of the input graph. Rather, we assume we are instead promised that each edge query received by the adjacency oracle is a query of an edge from G. This is the *stronger* model from the introduction.

3.1 Lower Bounds in the Weaker Model

In this section we show a close to linear lower bound on the preprocessing time of any algorithm computing an $O(n)$-size spanning subgraph adjacency oracle with high probability. We appeal to a construction from [23] in the setting of Local Computation Algorithms (LCAs) for spanners.

▶ **Theorem 17.** *For all absolute constants $\delta > 0$, any algorithm that computes an $O(n)$-size spanning subgraph adjacency oracle with probability at least $2/3$ must take $\Omega(n^{1-\delta})$ preprocessing time.*

Proof. Fix $\delta > 0$. From the proof of Theorem 1.3 of [23], there exists a family of random $n^{\delta/4}$-regular graphs on $n^{\delta/2}$ nodes such that any local computation algorithm outputting an $o(n^{3\delta/4})$ edge spanning subgraph with probability at least $2/3$ requires at least $\Omega(n^{\delta/4})$ queries of the graph. One important property of this family is that the randomly generated graph is connected with high probability.

Now, let our input graph be drawn from the family of disjoint unions of $n^{1-\delta/2}$ copies of graphs independently generated from this family. A spanning subgraph of any graph from this family is the disjoint union of spanning subgraphs on each connected component.

Consider any algorithm running in $o(n^{1-\delta})$ preprocessing time which computes a spanning subgraph adjacency oracle with probability at least $2/3$. At least half of the connected components must be unvisited during preprocessing. Hence, queries of edges from those connected components amount to running a local computation algorithm for those subgraphs with $\widetilde{O}(1)$ queries. Then, from the result of [23], since $\widetilde{O}(1) = o(n^{\delta/4})$ and the algorithm computes a spanning subgraph adjacency oracle with probability at least $2/3$, these connected components must contribute $\Omega(n^{1+\delta/4})$ edges to the spanning subgraph induced by the adjacency oracle. This implies the desired result. ◀

A possible objection to Theorem 17 is that the constructed hard instances are not connected. It turns out that it is relatively simple to get the same lower bound while also assuming connected input graphs. After generating the random connected component subgraphs $\{C_i\}$ as in Theorem 17, introduce a single auxiliary node r and connect it to one node in each C_i chosen at random. This leaves the asymptotic average degree unchanged. Since the C_i's are themselves connected with very high probability, the resultant graph formed after adding these edges is connected with high probability. Observe that any spanning subgraph of an input graph from this family is exactly spanning subgraphs of the C_i's connected by the newly introduced star of edges around r. Although one node in each C_i are now degree $n^{\varepsilon/4}+1$, this information does not distinguish between nodes in the induced subgraph of C_i (e.g., see the proof of Theorem 1.3 in [23]). Hence, the same analysis as in Theorem 17 applies.

3.2 Lower Bounds in the Stronger Model

In this section we show that any algorithm that computes an $o(n^2)$-size spanning subgraph adjacency oracle with high probability must take $\Omega(n)$ preprocessing time. However, the caveat is that this lower bound holds under the algorithmic model where queries to the adjacency oracle only have access to information stored during preprocessing. Namely, they do not have access to the adjacency oracle and adjacency list of the underlying graph accessible previously during preprocessing. We are also promised that every query of the spanning adjacency oracle is a query of an edge that exists in the initial graph. (Without this promise, the family of random spanning trees on n nodes yields the lower bound trivially.) Notably, our algorithms provide upper bounds in this model as well.

▶ **Theorem 18.** *Any algorithm that with constant probability computes an $o(n^2)$-size spanning subgraph adjacency oracle (without the oracle maintaining access to the input graph after preprocesssing) must take $\Omega(n)$ preprocessing time.*

Proof. Suppose that we are promised that our input graph will be two disjoint $n/2$-cliques joined by one random cut edge. The input graph is sampled by uniformly selecting a partition of $[n]$ into two parts of size $n/2$ and then selecting a pair of nodes between the two parts uniformly at random. It suffices to prove the result for this restricted set of input graphs.

Now, note that every degree oracle access will return $n/2-1$ except for queries to the endpoints of the random cut edge. That is, degree queries only reveal whether or not a vertex is an endpoint of the cut edge. Moreover, since the graph is two cliques joined by a cut edge, adjacency queries amount to revealing an additional node in a single clique. Every such query can be viewed as revealing the clique assignment of a node (since in reality it reveals at most that much information). In particular, adjacency queries do not reveal any information about the location of the endpoints of the cut edge unless they actually involve one of those endpoints.

Then, for any $f(n)=o(n)$, we may assume that in any $f(n)$ preprocessing time algorithm, the preprocessing phase amounts to revealing the clique assignment of $f(n)$ random nodes in each clique (since it reveals at most that many) and checking whether each of these nodes are an endpoint of the cut edge. In particular, with probability at least $1-\Theta(\frac{f(n)}{n})$ neither endpoint of the cut edge appears in a query.

Next, consider the subgraph induced by the unqueried nodes. With probability at least $1-\Theta(\frac{f(n)}{n})$, it is composed of two cliques of size $\Theta(n)$ connected by a cut-edge. In particular, queries to the data structure formed by the $f(n)$ preprocessing time reveal nothing beyond this about the unqueried nodes. Assume that the cut edge is in this induced subgraph.

On a query of the cut-edge, the data structure must return YES with at least constant probability (with the randomness here from preprocessing). If it returns NO, then the graph induced by the adjacency oracle is not spanning. However, none of the queried edges can be distinguished by the preprocessing data structure and, by the randomness of the graph input, any algorithm that treats the edge queries differently is equivalent to one that treats all edge queries identically. (The protocols are averaged over the random graph inputs since the cut-edge has an equal probability of being any of the edges between nodes not queried in preprocessing.) However, there are $\Omega(n^2)$ edges with both endpoints unqueried. Hence, the graph underlying the adjacency oracle will have $\Omega(n^2)$ edges, implying the desired result. ◀

4 Constructing Adjacency Oracles for 3-spanners

First, as a warm-up, we outline the construction of the adjacency oracle for approximately regular graphs. Then we extend the construction to general graphs.

4.1 Approximately Regular Graphs

In this section, we will assume that all vertices in the input graph $G(V, E)$ have degree within a constant factor C of D. We will generalize this to graphs of arbitrary degree in the next part. The following algorithm is phrased in the stronger model where the query algorithm may not access G, but we are promised that only edges from $E(G)$ will be queried. (Recall that if the query algorithm may access G, then this promise is not necessary, since we can first check whether the queried edge is in $E(G)$ and answer NO if not.)

Preprocessing Algorithm

If $D \leq n^{0.5}$, we can use the input graph itself as our adjacency oracle: that is, we can simply answer YES to every query. Otherwise, we construct the data structure for our adjacency oracle as follows:

3-Spanner Adjacency Oracle Preprocessing (Approximately Regular Setting)

- Iterate over each $v \in V$ and, independently with probability $\frac{100C \log n}{D}$, initialize a new cluster C_v, for which v is the *cluster center*.
- For all cluster centers c in an arbitrary order, loop over the edges (c, w) incident to c and check whether w is currently assigned to a cluster. If w is not assigned to a cluster, set its cluster membership to be the cluster centered at c and record the edge (c, w).
- For each $v \in V$, sample $100Cr \log n$ incident edges uniformly (where $1 \leq r \leq CD$ is a parameter to be specified later). For each edge (v, w), if w is in a different cluster than v and v has no edges recorded to w's cluster, record that v is adjacent to w's cluster, and record the edge (v, w).

We can implement our data structure with:

- An array, indexed by $v \in V$, with cell corresponding to v storing a set data structure supporting $\widetilde{O}(1)$ time insertions and queries (for checking whether or not v is adjacent to a given cluster).
- Another set data structure supporting $\widetilde{O}(1)$ time insertions and queries (for checking whether a given edge is recorded).

By the Chernoff bound, with high probability $\widetilde{O}(n/D)$ cluster centers are selected in the first step. Then, the second step takes $\widetilde{O}(n)$ time since each vertex is of degree at most CD. Since checking whether a vertex v is adjacent to a given cluster takes $\widetilde{O}(1)$ time using a dictionary, then the third step takes $\widetilde{O}(nr)$ time. In total, this preprocessing takes $\widetilde{O}(nr)$ time. An important feature of the construction is that all vertices will belong to a cluster:

▷ Claim 19. With high probability, every vertex $v \in V$ is assigned to a cluster in our algorithm.

Proof. Fix $v \in V$. Let X_u be the indicator random variable for vertex $u \in N[v]$ (the closed neighborhood of v) being selected as a cluster center. Then, if $\sum X_u \geq 1$, v will be clustered. Since v has degree at least D/C, the expectation of this sum is at least

$$\frac{D}{C} \cdot \frac{100C}{D} \cdot \log n = 100 \log n.$$

Then, by the Chernoff bound, since the X_u's are independent, the probability that the sum of the X_u's is less than one is at most

$$1 - \exp\left(\frac{-100 \log n}{4}\right) = 1 - 1/n^{25}.$$

Then, by the union bound, all vertices are then clustered with high probability. ◁

In the following, we will assume that this high-probability event occurs, and that all nodes are clustered.

Query Algorithm

On query (s, t), the adjacency oracle responds as follows (with `YES` meaning that the underlying 3-spanner has edge (s, t) and `NO` meaning that it does not have the edge):

3-Spanner Adjacency Oracle Query (Approximately Regular Setting)

- If the edge (s, t) was recorded in preprocessing, output `YES`.
- Else if s, t are in different clusters, **and** we did not record an edge from s to the cluster containing t, **and** we did not record an edge from t to the cluster containing s, output `YES`.
- Otherwise, output `NO`.

We can straightforwardly check which case holds in $\widetilde{O}(1)$ time. We next observe correctness of the spanner:

▷ Claim 20. The subgraph $H \subseteq G$ induced by the adjacency oracle is a 3-spanner of the input graph G.

Proof. By standard reductions [3, 1], it suffices to verify that for each edge $(s, t) \in E(G)$ for which the oracle responds `NO` to the query (s, t), there exists an $s \rightsquigarrow t$ path of length ≤ 3 of edges to which the oracle responds `YES`. There are two cases in which the oracles responds `NO`:

- First, suppose that s, t are in the same cluster and that neither s nor t are the center c of this cluster (otherwise we would record the edge (s, t)). Then we record edges (s, c) and (c, t), so $s \to c \to t$ forms a 2-path in H.

- Second, suppose that s is adjacent to the cluster of t via some recorded edge (s, x). Let c be the center of the cluster of t. Then, we record each edge in the 3-path $s \to x \to c \to t$. (This also handles the case in which $c = t$, in which case the above path is a 2-path of recorded edges.) The case in which t is adjacent to the cluster of s via a recorded edge follows analogously. ◁

▷ Claim 21. With high probability, the total number of edges in the subgraph induced by the adjacency oracle is $\widetilde{O}(n^2/r)$.

Proof. With high probability by the Chernoff bound, there are $\widetilde{O}(n/D)$ clusters. We assume this event occurs in the following analysis. We consider the cases in which the the adjacency oracle says `YES`:

- The oracle says `YES` to all recorded edges. The number of recorded edges between cluster centers and vertices is $O(n)$ since each vertex belongs to at most one cluster. The number of sampled edges that we record is at most $\widetilde{O}(n^2/D)$, since there are n vertices, $\widetilde{O}(n/D)$ clusters, and we record at most one edge from each vertex to each cluster. Note that, since we sample $\widetilde{O}(r)$ edges from each vertex, we may assume that r is at most the max degree $O(D)$, so this bound is $\widetilde{O}(n^2/r)$.
- The oracle also says `YES` to some non-recorded edges (u, v), so long as u, v lie in different clusters and we have not recorded an edge from u to the cluster of v, or vice versa. In order to bound these edges, we make two observations:
  - The number of vertex-cluster pairs is at most $\widetilde{O}(n^2/D)$ with high probability.
  - Fix a vertex-cluster pair, (v, X). If $r \geq \frac{D}{100 \log n}$, we sample all edges in G the edge-sampling step, so if there are any edges between v and X then one will be recorded in pre-processing. Otherwise, assume $r < \frac{D}{100 \log n}$ and suppose that v has x edges to X in G. Since $\deg(v) \leq C \cdot D$, each edge incident to v is sampled with probability at least $\frac{100Cr \log n}{CD} = \frac{100r \log n}{D}$. The expected number of sampled edges from v to X is then at least $\frac{100rx \log n}{D}$. So, if $x \geq D/r$, then, by the Chernoff bound, one of these x edges will be sampled with probability at least

$$1 - \exp\left(\frac{-100 \log n}{4}\right) = 1 - 1/n^{25}.$$

    Note that the indicator random variables for whether each edge incident to v is selected are actually not independent. Nonetheless, they are negatively correlated and the Chernoff bound still applies to negatively correlated random variables. Union-bounding over each vertex-cluster pair, at least one of the edges will be sampled for all vertex-cluster pairs (v, X) with $x \geq D/r$ with probability at least $1 - 1/n^{23}$. So, with high probability, the maximum number of edges added in this case is

$$\widetilde{O}\left(\frac{n^2}{D} \cdot \frac{D}{r}\right) = \widetilde{O}(n^2/r).$$

Applying the union bound then yields the result. ◁

▶ **Theorem 22.** *With high probability the above construction yields an adjacency oracle for a 3-spanner of G with $\widetilde{O}(n^2/r)$ edges, in $\widetilde{O}(nr)$ preprocessing time.*

We can set the parameter r to be any value between 1 and CD. In particular, we have the following.

▶ **Corollary 23.** *By setting $r = n^{0.5}$, with high probability the above construction yields an adjacency oracle for a 3-spanner of G with $\widetilde{O}(n^{1.5})$ edges, in $\widetilde{O}(n^{1.5})$ preprocessing time.*

Namely, for $D = \omega(n^{0.5})$, this algorithm computes an adjacency oracle of an optimal size 3-spanner of G in sublinear time.

4.2 Extending to General Graphs

To extend the previous algorithm to general graphs $G(V, E)$, we will create $\log n$ copies of the previous query data structure with each corresponding to a "bucket" of possible node degrees. On the query of a edge, we will pass the query to all of the data structures and return `YES` if any data structure returns `YES`, or `NO` if all data structures return `NO`. The idea is that the data structure that correctly guesses the minimum degree of the endpoints of the edge will ensure the 3-spanner stretch property for this edge. For vertices of especially low degree, e.g., less than $n^{0.5}$, we will just always say `YES` to the edge. Note that these buckets are *not* guessing the degree of *both* endpoints of each edge (u, v) being queried, but rather, the *minimum* degree between u, v. In this sense, we are not directly reducing to the previous analysis. Nonetheless, the high-level idea for each bucket data structure mirrors that of the approximately regular case.

Let us overview the technical reasons why we need to take this bucketing approach. The issue is that there is a tension between sampling enough cluster centers to ensure that each node is clustered, and the number of edges added at query time (which were not recorded in preprocessing). If we attempt the previous algorithm: for a graph with minimum degree δ, we need to sample $\widetilde{O}(n/\delta)$ many cluster centers, and so there would be $\widetilde{O}(n^2/\delta)$ vertex-cluster pairs. On the other hand, we can afford to sample $\widetilde{O}(n^{0.5})$ edges per node; for a graph with average degree D, this means that each edge is included with probability roughly $D/n^{0.5}$. Following the analysis from before, we would add $\Omega(n^{3/2}D/\delta)$ total edges, which is suboptimal for graphs with $D \gg \delta$.

But, if we instead handle edges with the minimum degree endpoint of degree ℓ in some data structure where we sample only $\widetilde{O}(n/\ell)$ cluster centers, both endpoints will be clustered with high probability. Moreover, for a given vertex-cluster pair (v, X) where v is degree ℓ and X is a cluster from the data structure with $\widetilde{O}(n/\ell)$ cluster centers, there can only be $\widetilde{O}(\ell/n^{0.5})$ unrecorded edges added per pair involving v. Otherwise, by an analysis similar to Claim 21 we would have sampled one with high probability. Handling the edges in this way then circumvents the previous conflict to yield the desired $\widetilde{O}(n^{3/2})$ edges.

Formally, we partition the interval $[n^{0.5}, n]$ into $O(\log n)$ buckets of the form $[2^k n^{0.5}, 2^{k+1} n^{0.5})$ for $k \in \{0, 1, \ldots, (\log n)/2 - 1\}$. For each non-empty bucket with lower bound ℓ there is a corresponding data structure. We describe the preprocessing and query algorithms for each individual data structure and describe how to combine these parametrized data structures to create the 3-spanner adjacency oracle.

Preprocessing Algorithm

For each non-empty bucket with lower bound ℓ, we construct its corresponding data structure as follows. The preprocessing is almost exactly as in the case of approximately D-regular graphs except using ℓ instead of D in the number of sampled cluster centers and $C = 2$. We also include a run-time optimization in the cluster assignment step that will be especially valuable when we extend these ideas to 5-spanners. We assume that the average degree D of the graph is at least $n^{0.5}$ or else we just return the graph as the adjacency oracle. We can check whether this is the case in $O(n)$ time.

3-Spanner Adjacency Oracle Preprocessing

- Iterate over each $v \in V$ and, independently with probability $\frac{c \log n}{\ell}$, initialize a distinct cluster for v and assign it as its cluster's *cluster center* (where $c > 0$ is a sufficiently large absolute constant we will leave implicit).
- Now we assign nodes to clusters.
  - If $\ell \geq \sqrt{D}$, loop over each neighbor of each cluster center and assign nodes to the first cluster center they are found to be adjacent to, recording the respective edge.
  - Otherwise, if $\ell < \sqrt{D}$, for all $v \in V$, iterate over its neighbors in a random order and assign it to the cluster of the first neighboring cluster center found, recording the corresponding edge.
- For all $v \in V$, sample $cr \log n$ of its incident edges. For each (v, w) sampled, if w is in a different cluster than v, record the edge (v, w) and that v is adjacent to w's cluster. Additional edges from v to w's cluster are then ignored.

We can implement each data structure using the same data structures as in the approximately regular case (array and set data structures).

Note that each degree is between 0 and n. The sum of the degrees of the sampled cluster centers is $\frac{cnD \log n}{\ell}$ in expectation. Then, since the cluster centers are sampled independently, the average degree of the cluster centers is at most ρD with probability at least

$$1 - \exp\left(\frac{-\rho c n D \log n}{2n\ell}\right) = 1 - n^{-c\rho D/(2\ell)}$$

by the Chernoff bound (for independent, bounded, and non-negative random variables), where $\rho = \max(5, \ell/D)$. The Chernoff bound used here arises from the standard Chernoff bound for independent random variables in $[0, 1]$ via dividing each random variable by their uniform bound (in this case n, since each cluster center has degree at most n). Then, for c a sufficiently large constant, with high probability the first case of the cluster assignment step takes $\widetilde{O}(\frac{n}{\ell} \cdot \rho D)$ time. When $\max(5, \ell/D) = \ell/D$, this is $\widetilde{O}(n)$. Otherwise, this is $\widetilde{O}(nD/\ell)$.

The second case of the second step takes $\widetilde{O}(n\ell)$ time with high probability. Namely, for nodes of degree at least ℓ, by the Chernoff bound, with high probability it takes checking at most ℓ random neighbors to find a cluster center and terminate. We can then union bound over all vertices to get the result for all vertices with high probability.

Hence, with high probability, the total preprocessing time over all data structures will be

$$\widetilde{O}(n \min(D/\delta, \sqrt{D}) + nr),$$

where $\delta := \max(\delta(G), n^{1/2})$. The variable δ corresponds to the order of the lower bound of the first bucket we instantiate. Note that the second step is somewhat different than in the approximately regular case in that we do not check all of the neighbors of v. This is a runtime optimization for graphs without a worst-case gap between average degree and minimum degree, but does not change the fact that all sufficiently high degree vertices will be clustered.

Query Algorithm

On the query of an edge (s, t), we query all $O(\log n)$ data structures and return `YES` if any data structure returns `YES`. Otherwise we return `NO`. On query (s, t), the data structure corresponding to lower bound ℓ responds as follows.

3-Spanner Adjacency Oracle Query

- If $\min(\deg(s), \deg(t)) < n^{0.5}$, output `YES`.
- Else if the edge (s, t) was recorded, output `YES`.
- Else if s or t is unclustered or $\min(\deg(s), \deg(t)) > 2\ell$ output `NO`.
- Else if s and t are in the same cluster or s or t have a recorded edge to the cluster of the other, output `NO`.
- Otherwise, output `YES`.

Each query can be made in $\widetilde{O}(1)$ time. (If we do not assume that the oracle has access to the input graph, we can store the degrees of each node in linear time during preprocessing.) Observe that the oracle behavior is identical to that in the case of approximately D-regular graphs except for the degree-checking conditions.

Intuitively, for an edge with minimum degree endpoint in the bucket $[\ell, 2\ell)$, the data structure corresponding to ℓ will handle the 3-spanner property for that edge. We observe that the endpoints of the edge will be clustered in that data structure:

▷ Claim 24. With high probability, for all bucket lower bounds ℓ, each vertex of degree at least ℓ is clustered in the data structure corresponding to ℓ.

Proof. For $\ell \geq \sqrt{D}$, this follows from the Chernoff bound and the union bound via a proof nearly identical to the proof of Claim 19.

For $\ell < \sqrt{D}$, let $v \in V$ be a vertex with $\deg(v) \geq \ell$. For each $u \in N[v]$, where $N[v]$ is the closed neighborhood of v, create an indicator random variable X_u for u being selected as a cluster center. The variables are independent, and their sum has expectation at least $c \log n$. Hence, by the Chernoff bound, the probability that the sum is at least 1 is at least $1 - n^{-c/4}$, yielding the desired result after union bounding over all $v \in V$ ◁

Now we can show that the subgraph $H \subseteq G$ induced by the adjacency oracle is indeed a 3-spanner of the input graph G.

▶ **Lemma 25.** *With high probability, the subgraph $H \subseteq G$ induced by the adjacency oracle is a 3-spanner of the input graph G.*

Proof. Let $(s, t) \in E(G)$ with $\min(\deg(s), \deg(t)) \in [\ell, 2\ell)$. If $\ell < n^{0.5}$, $(s, t) \in E(H)$ since every data structure outputs `YES`. Otherwise, consider the data structure corresponding to lower bound ℓ. Both s and t are clustered with high probability by Claim 24. We consider the two applicable `NO` cases and verify that the distance between s and t in the induced graph is at most 3.

Again, it suffices to verify that for each edge $(s, t) \in E(G)$ for which the oracle responds `NO` to the query (s, t), there exists an $s \rightsquigarrow t$ path of length ≤ 3 of edges to which the oracle responds `YES`.

- Suppose that s and t are in the same cluster in the data structure corresponding to ℓ (say centered at x). We cannot have $s = x$ or $t = x$ because then (s, t) was recorded and the data structure outputs `YES`. The oracle responds `YES` to each edge in the path $s \to x \to t$ since they are the edges recorded from s and t to their cluster center.
- Now, suppose that s and t belong to distinct clusters and an edge was recorded from s to the cluster of t (centered at x), with the edge being (s, u) for u in the cluster centered at x. Then, the data structure will respond `YES` to each edge in the path $s \to u \to x \to t$. In particular, the data structure responds `YES` to edges (u, x) and (x, t) since they are the edges recorded from u and t to their cluster centers. The data structure also responds

YES to (s, u) since that edge was recorded in the edge sampling step. (Note that it is possible that $u = x$ or $x = t$. In either of those cases, the path is of length 2 and is still composed of edges contained in H.) ◀

Finally, we bound the number of edges in the graph induced by the oracle.

▶ **Lemma 26.** *With high probability, the number of edges in the graph induced by the oracle is* $\widetilde{O}(n^2/\delta + n^2/r)$.

Proof. We first bound the number of edges added in each YES case of the data structure corresponding to bucket with lower bound ℓ. We will use that, with high probability, the number of clusters is $\widetilde{O}(n/\ell)$ and assume that this event holds in the below.

- If δ is the minimum degree of G, then the first YES case does not add any edges. Otherwise, it adds $O(n^{1.5})$ edges. Hence, this case adds at most $O(n^2/\delta)$ edges.
- Each vertex is adjacent to at most one cluster center and has at most one edge to each cluster. Hence, the second YES case adds at most $\widetilde{O}(n^2/\ell) = \widetilde{O}(n^2/\delta)$ edges, using the assumption of $\widetilde{O}(n/\ell)$ clusters.
- In the final YES case, one of the endpoints is of degree at most 2ℓ. We can bound the number of edges added in this case by bounding the number of unsampled edges added from vertices of degree at most 2ℓ. The number of pairs of vertices of degree at most 2ℓ and clusters is $\widetilde{O}(n^2/\ell)$. If $r \geq \frac{2\ell}{c \log n}$, we will sample all edges for each vertex of degree at most 2ℓ and no edges will be added from this case. Otherwise, assume $r < \frac{2\ell}{c \log n}$. For such a given vertex-cluster pair (s, X), if s has more than ℓ/r edges to vertices in X, then the expected number of sampled edges from s to vertices in X is at least $\frac{\ell}{r} \cdot \frac{r}{2\ell} \cdot c \log n = (c \log n)/2$. Hence, as in Claim 21, by the Chernoff bound (for negatively correlated random variables) and the union bound, with high probability, for each vertex cluster pair (s, X) with s without an edge recorded to X and s of degree at most 2ℓ, s has at most ℓ/r edges added to X. Hence, the total number of edges added in this case is $\widetilde{O}(n^2/r)$ with high probability.

Union bounding over all $O(\log n)$ data structures, for large enough c, these bounds hold for all data structures with high probability. Then, the number of total edges in the underlying graph of the oracle is at most a $\log n$ factor greater than $\widetilde{O}(n^2/\delta + n^2/r)$ which is still $\widetilde{O}(n^2/\delta + n^2/r)$. ◀

Union bounding all high probability bounds and choosing an appropriate c (say 200), this yields the following.

▶ **Theorem 27.** *With high probability the subgraph $H \subseteq G$ induced by the adjacency oracle is a 3-spanner of G with $\widetilde{O}(n^2/\delta + n^2/r)$ edges. The oracle can be constructed in $\widetilde{O}(nr + n \min(D/\delta, \sqrt{D}))$ time, for parameter r and $\delta = \max(n^{1/2}, \delta(G))$, where $\delta(G)$ is the minimum degree of a vertex in G.*

We have the following corollary.

▶ **Corollary 28.** *By setting $r = n^{0.5}$, with high probability the above construction yields an adjacency oracle for a 3-spanner of G with $\widetilde{O}(n^{1.5})$ edges, in $\widetilde{O}(n^{1.5})$ preprocessing time.*

5 Adjacency Oracles for 5-spanners and Beyond

In this work, we extend our adjacency oracles for 3-spanners to stretch 5 and beyond. Although the basic clustering technique is the same, some additional technical ideas are needed to execute the stretch analysis for larger stretch values. Due to space constraints we have deferred this to the full version of the paper.

References

1 Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Keaton Hamm, Mohammad Javad Latifi Jebelli, Stephen Kobourov, and Richard Spence. Graph spanners: A tutorial review. *Computer Science Review*, 37:100253, 2020.

2 Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1132–1139. SIAM, 2012.

3 Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9(1):81–100, 1993.

4 Rubi Arviv, Lily Chung, Reut Levi, and Edward Pyne. Improved lcas for constructing spanners. *arXiv preprint*, 2023. `arXiv:2105.04847`.

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

6 Paul Erdős. Extremal problems in graph theory. In *Proceedings of the Symposium on Theory of Graphs and its Applications*, page 2936, 1963.

7 Guy Even, Moti Medina, and Dana Ron. Best of two local models: Centralized local and distributed local algorithms. *Information and Computation*, 262:69–89, 2018.

8 Torben Hagerup, Kurt Mehlhorn, and James Ian Munro. Optimal algorithms for generating discrete random variables with changing distributions. *Lecture Notes in Computer Science*, 700:253–264, 1993.

9 Avinatan Hassidim, Jonathan A Kelner, Huy N Nguyen, and Krzysztof Onak. Local graph partitions for approximation and testing. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31. IEEE, 2009.

10 Avinatan Hassidim, Yishay Mansour, and Shai Vardi. Local computation mechanism design. *ACM Transactions on Economics and Computation (TEAC)*, 4(4):1–24, 2016.

11 Jacob Holm, Valerie King, Mikkel Thorup, Or Zamir, and Uri Zwick. Random k-out subgraph leaves only o (n/k) inter-component edges. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 896–909. IEEE, 2019.

12 Christoph Lenzen and Reut Levi. A centralized local algorithm for the sparse spanning graph problem. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

13 Reut Levi and Moti Medina. A (centralized) local guide. *Bulletin of the EATCS*, 122:60–92, 2017.

14 Reut Levi, Guy Moshkovitz, Dana Ron, Ronitt Rubinfeld, and Asaf Shapira. Constructing near spanning trees with few local inspections. *Random Structures & Algorithms*, 50(2):183–200, 2017.

15 Reut Levi, Dana Ron, and Ronitt Rubinfeld. A local algorithm for constructing spanners in minor-free graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2016)*, volume 60, pages 38:1–38:15, 2016.

16 Reut Levi, Dana Ron, and Ronitt Rubinfeld. Local algorithms for sparse spanning graphs. *Algorithmica*, 82(4):747–786, 2020.

17 Reut Levi, Ronitt Rubinfeld, and Anak Yodpinyanee. Local computation algorithms for graphs of non-constant degrees. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 59–61, 2015.

18 Yishay Mansour, Boaz Patt-Shamir, and Shai Vardi. Constant-time local computation algorithms. *Theory of Computing Systems*, 62:249–267, 2018.

19 Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. Converting online algorithms to local computation algorithms. In *Automata, Languages, and Programming: 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I 39*, pages 653–664. Springer, 2012.

20 Yishay Mansour and Shai Vardi. A local computation approximation scheme to maximum matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques: 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, pages 260–273. Springer, 2013.

21 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.

22 Huy N Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 327–336. IEEE, 2008.

23 Merav Parter, Ronitt Rubinfeld, Ali Vakilian, and Anak Yodpinyanee. Local Computation Algorithms for Spanners. In *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*, volume 124, pages 58:1–58:21, 2018.

24 David Peleg and Jeffrey Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing (SICOMP)*, 18(4):740–747, 1989.

25 David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM (JACM)*, 36(3):510–530, 1989.

26 Omer Reingold and Shai Vardi. New techniques and tighter bounds for local computation algorithms. *Journal of Computer and System Sciences*, 82(7):1180–1200, 2016.

27 Ronitt Rubinfeld. Can we locally compute sparse connected subgraphs? In *Computer Science–Theory and Applications: 12th International Computer Science Symposium in Russia, CSR 2017, Kazan, Russia, June 8-12, 2017, Proceedings 12*, pages 38–47. Springer, 2017.

28 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. *arXiv preprint*, 2011. `arXiv:1104.1377`.

29 Shay Solomon. Local algorithms for bounded degree sparsifiers in sparse graphs. *arXiv preprint*, 2021. `arXiv:2105.02084`.

30 Nithin Varma and Yuichi Yoshida. Average sensitivity of graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 684–703. SIAM, 2021.

31 Yuichi Yoshida, Masaki Yamamoto, and Hiro Ito. Improved constant-time approximation algorithms for maximum matchings and other optimization problems. *SIAM Journal on Computing*, 41(4):1074–1093, 2012.