An Exploration into Local Sparse Spanning Graph Algorithms

UROP+ Final Paper, Summer 2021

Jeffery Li*
Mentor/Coauthor: Aaron Berger†
Project Suggested by: Aaron Berger

September 2, 2021

Abstract

In this paper, we examine the problem of "computing" a sparse spanning subgraph in a connected graph with constant bounded degree in sublinear time with respect to the number of vertices n. Finding a spanning tree in a graph with constant bounded degree in linear time is straightforward and can be done with simple algorithms like BFS or DFS; determining whether an edge belongs to a specific sparse subgraph in sublinear time in a way that is consistent across queries is more challenging, even if we are allowed to add up to ϵn more edges, as we're not allowed to construct the entire subgraph. We summarize some known results, such as an algorithm that achieves complexity $n^{1/2+O(1/\log d)}$ when the graph has expansion close to its max degree d, and a more general $\tilde{O}(n^{2/3})$ algorithm, examining some of the bottlenecks that restrict the algorithm to specific cases or prevent the algorithm from achieving a lower runtime. We then discuss a different approach involving another search method involving examining truncated likelihoods of a random walk process, in hopes of using a more symmetrical search method to solve the general sparse spanning subgraph problem in faster time.

1 Intro

One well-studied problem in graph theory is finding a sparse subgraph, with some properties such as connectedness and low number of edges, in a connected graph. The most basic version of this problem is to compute a spanning tree, which can be solved in time linear in the number of edges, using algorithms like BFS and DFS. However, when the graph that we're dealing with is especially large even a runtime that is linear in the number of edges may be too high. To achieve a sublinear runtime, we use the Local Computation model, introduced by Rubinfeld, Tamir, Vardi, and Xie in 2011 [5].

In this model, we must give an algorithm that provides oracle access to a sparse spanning subgraph in the form of edge membership queries. A local computation algorithm is constrained to be globally consistent and memoryless. That is, if one were to ask the algorithm to provide membership information on every edge, the combined output should with high probability satisfy the problem constraints. Moreover, if the algorithm is called multiple times in sequence, it cannot use previous outputs when performing subsequent computations, and instead must recompute any such information it needs. As each edge is surely contained in some sparse subgraph, global consistency prevents us from simply answering positively for every edge. The memoryless condition prevents us from running, for example, Kruskal's algorithm one edge at a time, and ensures that we produce the same output even when edges are queried in a different order.

One can show that a local computation algorithm is not feasible if we require our sparse subgraph to have the minimum number of edges possible. For example, it is impossible to distinguish between a path on

*jeli@mit.edu
†bergera@mit.edu

n vertices and a cycle on n vertices without looking at every edge, and in order for the global output to be a spanning tree, we must answer positively for every edge on the path but negatively for exactly one edge on the cycle. Therefore, any LCA which produces a spanning tree must first distinguish between these two cases, which takes linear time. As such, we allow for some leniency - instead of requiring our subgraph to have exactly $n-1$ edges, we allow for it to have $n(1+O(\epsilon))$ edges, for some small parameter ϵ. This lets us answer positively on more queries without having to worry as much about the overall global structure of the graph (such as distinguishing between a path versus a cycle).

With this motivation, in 2014 Levi, Ron, and Rubinfeld [3] gave the following definition.

Definition 1.1. *An algorithm is a **Local Sparse Spanning Graph (LSSG) algorithm** if, given $n \geq 1$, $\epsilon > 0$, a sequence of random bits $r \in \{0,1\}^*$, and query access to a connected graph $G = (V, E)$, in incidence-lists representation, with n vertices, will provide oracle access to a subgraph $G' = (V, E')$ of G such that G' is connected, $|E'| \leq (1+\epsilon)n$ with high probability, and E' is determined by G and r. Note that G' has to be consistent across queries.*

We assume that the degrees of the vertices in our graph are bounded by some constant Δ. We are interested in LSSG algorithms that achieve a time complexity of $o(n)$, either in specific cases (such as the case where our graph has high expansion throughout, or is itself sparse), or in the general case.

Following this definition, Levi, Ron, and Rubinfeld [3] showed that the complexity of any LSSG algorithm must be $\Omega(\sqrt{n})$ in the general case, and gave two algorithms, one which achieves complexity $n^{1/2+O(1/\log\Delta)}$ in the case where there is high expansion, and another which achieves complexity $O(\Delta^{\rho(\epsilon)})$ in the case where our graph is ρ-hyperfinite (in other words, the graph can be partitioned into subsets of size $\rho(\epsilon)$ such that at most ϵn edges pass between partitions), which is independent of n. Work on this problem has since expanded, particularly in the case of minor-free graphs (which is related to hyperfiniteness), and further relaxation of the number of edges in the spanning subgraph to $\tilde{O}(n^{1+1/k})$ for some positive integer k, where the key improvement is maintaining a bound on the stretch factor of the subgraph [4]. Lenzin and Levi [2] achieved a runtime of $\tilde{O}(n^{2/3})$ in the general case, expanding on the ideas in the $n^{1/2+O(1/\log\Delta)}$ algorithm presented earlier, and including an algorithm of Elkin and Neiman [1] to deal with the cases where the neighborhood around a vertex is sparse.

The structure of this paper is as follows. In Section 2, we will examine the algorithm introduced by Levi, Ron, and Rubinfeld in 2014 that achieves complexity $n^{1/2+O(1/\log\Delta)}$ in the case where there is high expansion, and discuss some of the bottlenecks that prevent this algorithm from being applicable in the general case. In Section 3, we will examine the algorithm introduced by Lenzin and Levi in 2018 that achieves $\tilde{O}(n^{2/3})$ in the general case, along with the main bottlenecks that prevent the algorithm from achieving a better time complexity in its current form. Lastly, in Section 4, we discuss a new approach involving examining truncated likelihoods of a random walk process, with the hope of using a more symmetric search method to achieve a better time complexity in the general case.

2 Algorithm for High-Expansion Graphs

We first consider the case where the graph has fairly high expansion, following the notation and the algorithm presented by Levi, Ron, and Rubinfeld [3].

2.1 High-Expansion Case

We use a slightly modified version of an expander graph: we say that a graph G is an (s,h)-*expander* if $|N(S)| \geq h|S|$ for all subsets $S \subseteq V$ of size at most s, where $N(S)$ denotes the set of vertices either in S or adjacent to a vertex in S. From this, we can also define $h_s(G)$ to be the largest h such that G is a (s,h)-expander, given s and G.

We will describe a global algorithm, and then describe how to implement the global algorithm locally. We first define a few terms which we will be working with for these algorithms:

- Let $s = 2\sqrt{2n/\epsilon} \ln n = \tilde{O}(\sqrt{n/\epsilon})$,
- Let $\ell = \sqrt{\epsilon n/2}$, and
- Let t be a number such that between $(1-\epsilon/(2\Delta))n$ and $(1-\epsilon/(4\Delta))n$ vertices v have at least s vertices that are within distance t of v. One can show that random sampling can find such a value with high probability, and that $t \leq \frac{\log s}{\log h_s(G)} + 1$, so that if $h_s(G)$ is large, i.e. $\Delta^{\Omega(1)}$, then $t = O(\log_\Delta n)$.

We now introduce the global algorithm:

Algorithm 1: High-Expansion Global Algorithm

Input : Graph $G = (V, E)$, parameter t and $\epsilon > 0$, sequence of random bits $r \in \{0,1\}^*$
Output: Outputs a subgraph $G' = (V, E')$ of G such that G' is connected and $|E'| \leq (1+\epsilon)n$ with high probability.

Select ℓ centers uniformly and independently at random from V (based on r); call these $\sigma_1, \ldots, \sigma_\ell$.
Make all vertices unassigned initially. Then, for $i \in \{0, 1, \ldots, t\}$ and $j \in \{1, 2, \ldots, \ell\}$:
    Letting L_j^i denote the vertices at the ith level of the BFS tree of σ_j, assign all currently unassigned vertices in L_j^i to σ_j.
Now we have sets $S(\sigma_j)$ of vertices assigned to each center σ_j, along with a set of singletons S'. Now, we construct $G' = (V, E')$:
    For $j \in \{1, 2, \ldots, \ell\}$, find the BFS tree with root σ_j spanning $S(\sigma_j)$; add all of the edges $E'(\sigma_j)$ of the BFS tree into E'.
    For each pair of centers σ_{j_1} and σ_{j_2}, find the shortest path $P(\sigma_{j_1}, \sigma_{j_2})$ between the centers, with minimum lexicographic order (determined by ids). If all vertices in $P(\sigma_{j_1}, \sigma_{j_2})$ belong in either $S(\sigma_{j_1})$ or $S(\sigma_{j_2})$, then add to E' the single edge (v_1, v_2) such that $v_1 \in S(\sigma_{j_1})$ and $v_2 \in S(\sigma_{j_2})$ (otherwise, do nothing).
    For each $w \in S'$, add all edges incident to w to E'.
Return $G' = (V, E')$.

The output satisfies the following properties.

Theorem 2.1 ([3, Lemma 2, Lemma 3])**.** *The output graph $G' = (V, E')$ is connected, and $|E'| \leq (1+\epsilon)n$ with high probability.*

Proof. To prove the first part, we first note that all vertices within each $S(\sigma_j)$ are connected via the BFS trees. It thus suffices to show that all the centers are connected to each other, and all the singletons are connected to some center.

To show that all the centers are connected, we proceed by induction based on the distance between the two centers, $d = d(\sigma_i, \sigma_j)$. The base case, $d = 1$, is clear, as we would simply add the edge connecting the two centers.

For the inductive step, consider the path $P(\sigma_i, \sigma_j)$. If all the vertices are contained in $S(\sigma_i) \cup S(\sigma_j)$, then we would've added the edge bridging the two, and thus connected the centers.

Otherwise, if there is some vertex v' that belonged to some other non-singleton set $S(\sigma_k)$, then note that v' is closer to σ_k than to σ_i and σ_j (or has lower id). This means that, since $P(\sigma_i, \sigma_j)$ is the shortest path between σ_i and σ_j, the distances $d(\sigma_i, \sigma_k)$ and $d(\sigma_k, \sigma_j)$ are both upper-bounded by d, and if they are equal to d, then k has a lower id. This allows us to invoke the inductive hypothesis and note that, since σ_i and σ_k are connected and σ_k and σ_j are connected, σ_i and σ_j are connected.

Otherwise, there are only singletons along this path, besides vertices in $S(\sigma_i) \cup S(\sigma_j)$. In this case, since we took all edges incident to singletons, these edges would bridge between the two sets, and thus connect the centers.

This finishes the inductive step and shows that all the centers are connected to each other.

To show that each singleton is connected to some center, we induct on the distance d to the nearest center, where the base case $d = t + 1$ is clear (we connect to a vertex of distance t from the nearest center, which in turn is connected to the center), and the inductive step follows similarly (connect to a vertex of distance $d - 1$ from the nearest center, which in turn is connected to the center). Thus, G' is connected, as desired.

To show that $|E'| \leq (1 + \epsilon)n$ with high probability, note that the number of edges added by the BFS trees is bounded above by n and the number of edges added by the shortest paths is bounded above by $\ell^2 = \epsilon n/2$. We can then show that each vertex $v \in V_{t,s} = \{v \mid v \text{ has at least } s \text{ vertices within } t \text{ of it}\}$ has a $(1 - s/n)^\ell < 1/n^2$ chance of being a singleton, so all $v \in V_{t,s}$ are assigned to some center with probability at least $1 - 1/n$ (i.e. with high probability). This means that there are at most $\epsilon n/2\Delta$ singletons, and thus at most $\epsilon n/2$ additional edges, with high probability. Thus, adding these all up, we get that $|E'| \leq (1 + \epsilon)n$ with high probability.

Thus, our algorithm has the intended behavior. □

It now suffices to implement the algorithm locally, in what Levi, Ron, and Rubinfeld call the Centers' Algorithm.

Algorithm 2: Centers' Algorithm [3, Algorithm 1]

Input : Edge $(u, v) \in E$ of graph $G = (V, E)$, with parameter t and $\epsilon > 0$ and choice of centers $\sigma_1, \ldots, \sigma_\ell$ that is fixed over all queries

Output: Outputs whether or not $(u, v) \in E'$.

1 Perform a BFS from both u and v to depth t, in G, to find out if u and v belong to some center (which can be verified using, say, a hash table). If at least one of u and v is a singleton, then return `YES`. Otherwise, let $\sigma(u)$ and $\sigma(v)$ denote the centers closest to u and v.

2 If $\sigma(u) = \sigma(v) = \sigma$, then note that $|d(u, \sigma) - d(v, \sigma)| \leq 1$ (as (u, v) is an edge in G), and do the following:

3 If $d(u, \sigma) = d(v, \sigma)$, then return `NO`.

4 If $d(u, \sigma) = d(v, \sigma) + 1$, then look at all neighbors of u. If there exists a neighbor $w \neq v$ such that $d(w, \sigma) = d(v, \sigma)$ (can be verified via BFS) and $\mathrm{id}(w) < \mathrm{id}(v)$, then return `NO`. Otherwise, return `YES`.

5 Otherwise, if $\sigma(u) \neq \sigma(v)$, then perform a BFS to depth t from $\sigma(u)$ and $\sigma(v)$ to try to find the shortest path $P(\sigma(u), \sigma(v))$ (by incrementing the depth from both sides and checking if and when the BFS's meet). If both u and v belong to $P(\sigma(u), \sigma(v))$, then return `YES`. If either one doesn't belong, or the BFS from both sides fails to find $P(\sigma(u), \sigma(v))$ after t steps from both sides, return `NO`.

It's clear how this is a local implementation of the global algorithm, as all of the computations are done within the neighborhood of u and v. Running through the analysis, the query complexity is mostly determined by the BFS's, which cause the runtime to be

$$O(\Delta^{t+1}) = O(\Delta^2 s^{\log_{h_s(G)} \Delta})$$

(using the fact $a^{\log b} = b^{\log a}$). In the case where $h_s(G)$ is very close to Δ (i.e. there is high expansion at every vertex, up to a neighborhood of size s), then the query complexity is

$$O(\Delta^2 s) = n^{1/2 + O(1/\log \Delta)}.$$

The running time has an extra $\log n$ factor due to the length of the id's of the vertices. The amount of randomness needed here is $O(\sqrt{\epsilon n} \log n)$, for the centers $\sigma_1, \ldots, \sigma_\ell$.

2.2 Guess for General Case

With the Centers' Algorithm in mind, we now consider what happens in the general case, where there may be areas of low expansion throughout the graph. We first analyze what happens when we try to use the ideas from the Centers' Algorithm in the general case.

Once again, we start with presenting the global algorithm, with some slight modifications:

Algorithm 3: General Global Algorithm (Centers)

Input : Graph $G = (V, E)$, $\epsilon > 0$, sequence of random bits $r \in \{0,1\}^*$
Output: Outputs a subgraph $G' = (V, E')$ of G such that G' is connected and $|E'| \leq (1+\epsilon)n$ with high probability.

Select $\ell = \sqrt{\epsilon n/2}$ centers uniformly and independently at random from V (based on r); call these $\sigma_1, \ldots, \sigma_\ell$.
Make all vertices unassigned initially. Then, for each vertex v, run BFS until s vertices have been explored, and assign v to the first center that is hit, or let it be a singleton otherwise.
Now we have sets $S(\sigma_j)$ of vertices assigned to each center σ_j, along with a set of singletons S'. Now, we construct $G' = (V, E')$:
    For $j \in \{1, 2, \ldots, \ell\}$, find the BFS tree with root σ_j spanning $S(\sigma_j)$; add all of the edges $E'(\sigma_j)$ of the BFS tree into E'.
    For each pair of centers σ_{j_1} and σ_{j_2}, find the shortest path $P(\sigma_{j_1}, \sigma_{j_2})$ between the centers, with minimum lexicographic order (determined by ids). If all vertices in $P(\sigma_{j_1}, \sigma_{j_2})$ belong in either $S(\sigma_{j_1})$ or $S(\sigma_{j_2})$, then add to E' the single edge (v_1, v_2) such that $v_1 \in S(\sigma_{j_1})$ and $v_2 \in S(\sigma_{j_2})$ (otherwise, do nothing).
    For each $w \in S'$, add all edges incident to w to E'.
Return $G = (V, E')$.

This global algorithm still works, with the proof being similar to the proof of why the global algorithm works in the high-expansion case. But is it possible to implement locally? We need to be able to answer three questions locally:

1. Given a vertex v, what is its center?
2. Given u, v with the same center, does (u, v) lie on the BFS tree from $\sigma(u)$?
3. Given u, v with different centers, does (u, v) lie on the shortest path between $\sigma(u)$ and $\sigma(v)$?

The first question is fairly straightforward to answer - we can run BFS until s vertices have been explored like in the global algorithm.

For the second question, we can always find the answer; there are a few possible ways to proceed (here we assume that $d = d(u, \sigma(u)) = d(v, \sigma(u)) - 1$):

1. One way, which isn't too different from the method used in the high-expansion algorithm, is still to run a BFS from $\sigma(u)$ outwards until we find out whether or not (u, v) lie on the BFS tree, but we prune any part of the search that goes outside of the set of vertices searched through during the BFS from vertex v. This still allows us to look at the relevant parts of the BFS tree with root $\sigma(u)$, but we reduce the set of vertices that we look at by ignoring vertices that lie outside the original search.
2. Another way, which flips which vertex we start our BFS from, is to use the fact that, for any arbitrary vertex v with center $\sigma(v)$, all vertices along the shortest path from v to $\sigma(v)$ also have center $\sigma(v)$ (which can be seen via a proof by contradiction). This means that we could BFS from each of the neighbors of v to depth d to see which neighbors of v are on a shortest path from v to $\sigma(v) = \sigma(u)$ (as a neighbor of v is on a shortest path iff it is of distance d away from $\sigma(u)$), and then see if u has the smallest id out of these neighbors that are of distance d from $\sigma(u)$.

3. Yet another way changes which edges we keep in E' - instead of keeping the edges of the BFS trees, we do the following: for each vertex u with center $\sigma(u)$, find the shortest path, with minimum lexicographic order, from u to $\sigma(u)$, and keep the first edge in that shortest path. It's clear that this keeps a spanning tree, as each non-center vertex contributes exactly one edge and has a unique parent with depth that is one smaller than its depth. This also allows us to determine whether or not (u, v) is in E' relatively easily - find the the shortest path, with minimum lexicographic order, from u to $\sigma(u)$ via BFS, and see if (u, v) is in this path.

The third question seems much harder to answer, even if we are allowed to err on the side of caution for up to ϵn vertices. The main issue is that the sizes of the clusters may vary a lot due to the expansion of the graph at various regions - we may consider, for example, the case where our graph contains a substructure consisting of a binary tree on $n^{1/4}$ vertices, where each leaf vertex then has a distinct path of length $\theta(n^{1/2})$ connected to it. Note that if the root vertex of this substructure were to be a center, all of the vertices would be able to find the root vertex as the center, but this would mean that the set of vertices assigned to this center is $\theta(n^{3/4})$, which makes searching via BFS very costly. Of course, we may not end up in the scenario where all the vertices are assigned to the root vertex, because we would expect $\theta(n^{1/4}\sqrt{\epsilon})$ vertices in this substructure to be centers, but the imbalance in expansion at various regions of our graph may still cause some issues.

3 Refined algorithm for general case

Since the third question seems to be extremely difficult to remedy, at least using our guess for the global algorithm, we consider a modified version of the global algorithm. Here, we follow the notation and algorithms presented by Lenzin and Levi [2].

3.1 Algorithm with a Promise

For this "algorithm with a promise," still consider the cases where expansion is relatively high (i.e. we can reach $n^{1/3}$ vertices within $\log(n)/\log(1+\epsilon)$ steps of every vertex - more precisely, $k = cn^{1/3}\log n \cdot \ell\Delta/\epsilon$ vertices within a neighborhood of depth $\ell \in [b\log n/\log(1+\epsilon), b\log n/\log(1+\epsilon)+\Delta/\epsilon])$; the general algorithm involves combining this algorithm with an algorithm for ultra-sparse graphs by Elkin and Neiman [1], along with a little bit more work.

The initial set-up is similar in concept, but with slightly different settings: pick $r = \Theta(\epsilon n^{2/3}/\log n)$ centers, and assign each vertex to the closest center. However, we now call these groupings *Voronoi cells*, to distinguish these from *clusters*, which we can define in the following way: if a Voronoi cell has less than k vertices, then it is its own cluster. Otherwise, consider the BFS tree from the center of the Voronoi cell, and consider some vertex v. If the subtree with root v has at least k vertices, then v is in its own cluster. Otherwise, it belongs to the cluster whose "center" is the highest ancestor (potentially itself) with at most k vertices in its subtree. Another way to think about this construction is, we consider the clustering starting from the clustering given by Voronoi cells, along with their BFS trees, and then repeatedly do the following until all clusters have at most k vertices: pick a cluster, make the center of the cluster its own center, and split up the subtrees into their own clusters.

This way of defining clusters ensures that we have an upper bound on the sizes of the clusters. It's possible that some Voronoi cell will have a lot of vertices even though the expected number of vertices is $\Theta(n^{1/3}\log n)$, an issue that came up in our guess for the general case, so that's why we perform some more splitting of these cells like the above. This size bound is useful as it allows us to compute the entire cluster that we are in fairly quickly.

It turns out that the number of clusters is also bounded fairly well.

Lemma 3.1 ([2, Lemma 1]). *We have the following bound:* $s \leq r + n\ell(\Delta+1)/k = O(\epsilon n^{2/3}/\log n)$.

Proof. We bound the number of singletons that are created due to a vertex having at least k vertices in its subtree. Define a vertex to be *special* if its subtree has at least k vertices, but none of its children have at least k vertices in their subtrees. Note that every singleton created must be the ancestor of a special vertex - either all of its children have less than k vertices in their subtrees, in which case the vertex itself is special, or some child has at least k vertices in its subtree, in which case we can use an inductive argument downwards through that child. Furthermore, each special vertex has at most ℓ ancestors, and the subtrees corresponding to special vertices are vertex-disjoint, as they cannot be children of each other (as then some child of some special vertex has at least k children). This means that there are at most n/k special vertices, and thus at most $n\ell/k$ ancestors of special vertices, or at most $n\ell/k$ singletons created.

From this, we can also bound the number of clusters who has a vertex whose parent is a singleton (i.e. has at least k vertices in its subtree) - since the degrees are bounded by Δ, and there are at most $n\ell/k$ singletons, the number of such clusters is bounded by $n\ell\Delta/k$.

Finally, the last set of clusters to consider are those that are entire Voronoi cells. This is bounded by the total number of Voronoi cells, which is r.

Combining all of these together, we get the bound $s \leq r + n\ell(\Delta+1)/k = O(\epsilon n^{2/3}/\log n)$, as desired. □

Now we consider the edge set of our sparse subgraph. As before, we include the BFS trees of all of the Voronoi cells. What we do to connect these Voronoi cells is slightly different, however. We first mark each center independently and at random, with probability $p = n^{-1/3}$. If a center is marked, then we also say that the Voronoi cell is marked, along with all clusters inside the Voronoi cell. Then, roughly speaking, we do the following:

1. We connect each cluster to all adjacent marked clusters,
2. We connect an unmarked cluster with no neighboring marked clusters to all adjacent cells, and
3. We connect all unmarked clusters A with any neighboring unmarked cluster B that is adjacent to a marked Voronoi cell $\mathrm{Vor}(C)$, under certain conditions (such as $\mathrm{Vor}(B)$ having the minimum rank among all Voronoi cells adjacent to both A and C, and the edge of minimum rank between A and $\mathrm{Vor}(B)$ being between A and B) to trim down the number of edges added.

Here, by "connecting" two clusters, we mean add the edge with smallest rank between the two clusters (and similar for "connecting" a cluster to a cell).

We now have the following two lemmas, regarding the sparse subgraph that is formed.

Lemma 3.2 ([2, Lemma 2]). *The expected number of edges in* E' *is* $(1+O(\epsilon))n$. *In fact,* $|E'| = (1+O(\epsilon))n$ *with large constant probability.*

Proof. For the first step, notice that the expected number of marked cells is sp, as there are s clusters and each has probability p of being marked. Thus, the expected number of edges added in the first step is $s^2p = O(\epsilon n/\log n)$.

For the second step, note that if a cluster A has at least $\frac{3\ln n}{p}$ neighboring cells, then the probability that all neighbors of A are unmarked is at most $(1-p)^{3\ln(n)/p} \leq \frac{1}{n^3}$; since there are at most n such cells, the probability that some cluster with at least $\frac{3\ln n}{p}$ neighboring cells has no marked neighboring cluster is at most $\frac{1}{n^2}$ by the union bound, and so the expected number of edges added due to these clusters is at most $|E|/n^2 < 1$. Now, for the clusters that have at most $\frac{3\ln n}{p}$ neighboring cells, these will add at most

$\frac{3s \ln n}{p} = O(\epsilon n)$ edges regardless, meaning that expected number of edges added in the second step is at most $\frac{3s \ln n}{p} + 1 = O(\epsilon n)$.

For the third step, notice that for any arbitrary cluster A and arbitrary marked cluster C, there can only be at most 1 edge connecting A to an unmarked neighboring cluster B of C, as otherwise we get a contradiction, either from the minimality of the rank of one of the edges or from the minimality of the rank of one of the Voronoi cells. Thus, since there are s clusters in total and expected sp marked clusters, the expected number of edges added in the third step is $s^2 p = O(\epsilon n / \log n)$.

Therefore, the expected number of edges added between clusters is at most $O(\epsilon n)$; using Markov's inequality, we can say that with constant probability, the number of edges added between clusters is at most $O(\epsilon n)$. Adding in the edges added through the BFS trees gives us the $(1 + O(\epsilon))n$ edges, as desired. □

As a remark, notice that when looking at the number of extra edges added, the first and third step produce $O(s^2 p)$ edges in expectation, and the second step produces $O(s \ln n / p)$ edges in expectation. These two combined give us our first "bottleneck" of the algorithm - if we want both of these to be $O(\epsilon n)$, then we need $s^3 = (s^2 p)(s/p) = O(\epsilon^2 n^2)$, or $s = O((\epsilon n)^{2/3})$. Thus, we can't have more than $n^{2/3}$ clusters, at least in this set-up, or our clusters can't have size less than $n^{1/3}$ (otherwise we are very likely to create more edges than we want).

Lemma 3.3 ([2, Lemma 3])**.** *The resulting graph $G' = (V, E')$ is connected.*

Proof. Since the vertices in each Voronoi cell are connected via the BFS trees in each cell, and the original graph itself is connected, it suffices to show that every pair of adjacent Voronoi cells $(\text{Vor}_1, \text{Vor}_2)$ is connected. Now, there are multiple cases to consider:

1. At least one of $(\text{Vor}_1, \text{Vor}_2)$ is marked. Then, by the first step above (connecting clusters to all adjacent marked clusters), the two cells are connected.
2. Neither of $(\text{Vor}_1, \text{Vor}_2)$ are marked, and there is some pair of adjacent clusters $(A, B) \in (\text{Vor}_1, \text{Vor}_2)$ such that both clusters are not adjacent to any marked clusters. Then, by the second step above (connecting all pairs of adjacent unmarked clusters where both have no neighboring marked clusters), the two clusters are connected, and so the two cells are connected.
3. Neither of $(\text{Vor}_1, \text{Vor}_2)$ are marked, and there is no pair of adjacent clusters $(A, B) \in (\text{Vor}_1, \text{Vor}_2)$ such that both clusters are not adjacent to any marked clusters. In this case, consider the edge of minimum rank between $(\text{Vor}_1, \text{Vor}_2)$; WLOG let it be between (A, B) with B adjacent to some marked cluster C in a marked Voronoi cell Vor_3 (the argument works the same if A is adjacent to some marked cluster). Then, there are two sub-cases here:
   - Vor_2 has the minimum rank among all Voronoi cells adjacent to both A and C. In this case, we would add this edge in our third step, meaning that (A, B) are connected, and so the two cells are connected.
   - Vor_2 does not have the minimum rank among all Voronoi cells adjacent to both A and C. Suppose Vor_4 is the cell of minimum rank among all Voronoi cells adjacent to both A and C; in particular, note that it has lower rank than Vor_2. Now, because Vor_3 is marked, and Vor_3 is adjacent to both Vor_2 and Vor_4, it suffices to show that Vor_1 and Vor_4 are connected. This is because there would be a path from Vor_1 to Vor_4 to Vor_3 to Vor_2, and so Vor_1 and Vor_2 would be connected. However, because Vor_4 has lower rank than Vor_2, we can induct downwards to show that Vor_1 and Vor_4 are connected, and thus Vor_1 and Vor_2 are connected.

Thus, in all cases, the pair of adjacent Voronoi cells are connected, and so the graph is connected. □

3.2 General Algorithm

Now that we have the algorithm that works under a promise regarding the expansion around vertices, we describe the general algorithm. We first give an overview of the other ingredient needed for the general algorithm, an algorithm for ultra-sparse spanners given by Elkin and Neiman.

Algorithm 4: Elkin-Neiman Algorithm [1, Section 2]

Input : Connected graph $G = (V, E)$, integer h, parameter δ controlling success probability, sequence of random bits $r \in \{0,1\}^*$

Output: Outputs a subgraph $G' = (V, E')$ of G such that G' is connected and the expected number of edges is at most $n \cdot (n/\delta)^{1/h}$.

1 Each vertex u samples a value r_u from an exponential distribution $\mathcal{EXP}(\ln(n/\delta)/h)$.

2 Then, each vertex v receives all r_u from every vertex u such that $d(u,v) \leq h$, and stores all $m_u(v) := r_u - d(u,v)$ along with a neighbor $n_u(v)$ on the shortest path from u to v.

3 Lastly, for all vertices v, add all edges in $C(v) := \{(v, n_u(v)) \mid m_u(v) \geq \max_{w \in V}(m_w(v) - 1)\}$ to E'.

4 Return $G' = (V, E')$.

One way we can think of this is that each vertex will generate a random signal from an exponential distribution, whose support (i.e. probability density function is greater than 0) is the set of positive real numbers and whose cumulative density function is $1 - e^{-\beta x}$, for $x \geq 0$ and a parameter β. This distribution has the *memoryless property* - that is, if $X \sim \mathcal{EXP}(\beta)$, then $\mathbb{P}(X \geq a + b \mid X \geq a) = \mathbb{P}(X \geq b)$ (one can think of this as a continuous version of flipping coins - the probability that we get our first heads within $a + b$ flips, given that our first a flips are all tails, is the probability that we get our first heads within b flips). Then, each of the vertices will broadcast their signals outwards h steps, each signal weakening by 1 every edge it travels across. All of the vertices will receive and store these weakened versions of the signals of vertices within distance h of it. Then, the vertices will find the maximum out of these signals, and for each signal that is within 1 of the maximum (i.e. a threshold), the vertex will add an edge along the shortest path from itself to the vertex corresponding to that signal.

The paper by Elkin and Neiman that introduces this algorithm proves that the stretch factor is at most $2h - 1$; in particular, since this stretch factor is finite, this means that the graph produced by running this algorithm does not break up any connected components. Thus, since our algorithm with a promise mainly works on graphs with vertices that have a large neighborhood of vertices within $O(\log n)$ distance (more precisely, $k = \tilde{O}(n^{1/3})$ vertices within ℓ distance, where k and ℓ are defined earlier), and this algorithm works by having each vertex receive signals from all vertices within a certain distance h of it, it makes sense to have this algorithm run on only the vertices that don't have a center within ℓ of them, which we can call *remote* vertices.

This leads to the general global algorithm:

Algorithm 5: General Global Algorithm (Clusters) [2]

Input : Connected graph $G = (V, E)$, $\epsilon > 0$, sequence of random bits $r \in \{0,1\}^*$
Output: Outputs a subgraph $G' = (V, E')$ of G such that G' is connected and $|E'| \leq (1 + O(\epsilon))n$ in expectation.

1 Randomly choose a set S of centers, with $|S| = \Theta(\epsilon n^{2/3}/\log n)$.
2 Randomly choose $\ell \in [b \log n/\log(1+\epsilon), b \log n/\log(1+\epsilon) + \Delta/\epsilon]$, and define $k = cn^{1/3} \log n \cdot \ell\Delta/\epsilon$ (where b and c are large enough constants).
3 Let R denote the set of *remote* vertices, or vertices that don't have a center within ℓ of them, and let $\overline{R} = V \setminus R$. Define $G_R = (R, E_R)$, with E_R denoting the set of edges whose endpoints are both remote, and $G_{\overline{R}} = (\overline{R}, E_{\overline{R}})$, with $E_{\overline{R}}$ denoting the set of edges whose endpoints are both not remote.
4 Run the Elkin-Neiman Algorithm on the connected components of G_R, with $h = \ell$ and $\delta = 1/n^{b-1}$. Add all added edges into E'.
5 Run our "algorithm with a promise" on the connected components of $G_{\overline{R}}$, with each Voronoi cell marked randomly with probability $p = 1/n^{1/3}$. Add all added edges into E'.
6 Add any edges $(u, v) \in E$ with $u \in R$ and $v \in \overline{R}$ (or vice versa) to E'.
7 Return $G' = (V, E')$.

We now show that this algorithm satisfies the required conditions.

Theorem 3.4 ([2])**.** *The output graph $G' = (V, E')$ is connected, and the expected number of edges in E' is at most $(1 + O(\epsilon))n$.*

Proof. It's clear that this subgraph is connected - the connected components of G_R and $G_{\overline{R}}$ remain connected and we preserve all connections between the connected components of G_R and the connected components of $G_{\overline{R}}$.

As for the number of edges added, for the fourth and fifth steps (running the Elkin-Neiman Algorithm and the "algorithm with a promise"), the subroutines produce a subgraph of G_R with at most $O(|R| \cdot (n^b)^{\log(1+\epsilon)/(b \log n)}) = O(|R|(1+\epsilon))$ edges in expectation, and a subgraph of $G_{\overline{R}}$ with at most $O(|\overline{R}|(1+\epsilon))$ edges in expectation. For the sixth step (adding the edges that bridge between G_R and $G_{\overline{R}}$, note that for each edge (u, v), there is at most one value of $r_{u,v}$, within the range that ℓ is pulled from, such that one vertex is remote with respect to S but the other one isn't, and so the probability that an edge (u, v) gets added in this case is $\Pr[\ell = r_{u,v}] \leq \epsilon/\Delta$, so we add at most $(\Delta n)(\epsilon/\Delta) = \epsilon n$ edges in expectation at this step. Thus, in total, we have $O((|R| + |R'|)(1+\epsilon) + n\epsilon) = n(1 + O(\epsilon))$ edges in expectation.

Thus, our algorithm has the desired behavior. □

It's also clear how to make this algorithm local.

Algorithm 6: General Local Algorithm [2, Algorithm 1]

Input : Connected graph $G = (V, E)$, $\epsilon > 0$, set S of chosen centers (with $|S| = \Theta(\epsilon n^{2/3}/\log n)$) along with whether or not they are marked, constants $\ell \in [b \log n/\log(1+\epsilon), b \log n/\log(1+\epsilon) + \Delta/\epsilon]$ and $k = cn^{1/3} \log n \cdot \ell\Delta/\epsilon$, sequence of random bits $r \in \{0,1\}^*$, and a query edge (u, v)

Output: Outputs whether or not $(u, v) \in E'$, where E' is as specified in the global algorithm.

1 Determine whether or not u or v or both are remote. If exactly one of them is remote, output `YES`.

2 If both are remote, run the Elkin-Neiman Algorithm within the set of remote vertices in the union of the ℓ-neighborhoods of u and v (which should contain $\tilde{O}(n^{1/3})$ vertices with high probability). Return the answer that the Elkin-Neiman Algorithm returns.

3 Else, if both are not remote, then proceed as in our "algorithm with a promise":

4 If both are in the same Voronoi cell, determine whether or not this edge is in the BFS tree of this Voronoi cell (see last part of Section 3), and return that answer.

5 Else, determine the clusters $A \ni u$ and $B \ni v$. If A and B don't satisfy any of the conditions: (i) at least one is marked, (ii) one of them, WLOG A, is unmarked and has no marked neighboring clusters, or (iii) neither are marked and one of them, WLOG B, is adjacent to a marked Voronoi cell $\mathrm{Vor}(C)$, such that $\mathrm{Vor}(B)$ has minimum rank among all cells adjacent to both A and C, and the edge of minimum rank between A and $\mathrm{Vor}(B)$ is between A and B, then output `NO`. Otherwise, if (u, v) is the minimum-rank edge between A and B (or in the case of the second condition, between A and $\mathrm{Vor}(B)$, or A and $\mathrm{Vor}(B)$, depending on which one has no marked neighboring clusters), output `YES`. Otherwise, output `NO`.

We consider the general runtime of this algorithm. Determining whether or not a vertex is remote takes $\tilde{O}(n^{1/3})$ time, based on the number of centers that we chose. Thus, the first step takes $\tilde{O}(n^{1/3})$ time. If either vertex happens to be not remote, we can also obtain information about its center and the distance from the center with no extra cost.

Now, in the "both remote" case, even before the Elkin-Neiman algorithm subroutine, this involves looking through all vertices within the union of the ℓ-neighborhoods of u and v, and determining whether or not they are remote. Since both vertices are remote, there should be $\tilde{O}(n^{1/3})$ vertices within ℓ of either u or v with high probability. Combining this with the cost of determining whether or not a vertex is remote (which takes $\tilde{O}(n^{1/3})$ time), we get a $\tilde{O}(n^{2/3})$ cost before running the Elkin-Neiman algorithm. However, running the Elkin-Neiman algorithm would take time $\tilde{O}(n^{1/3})$ as we can BFS outwards to receive all of the signals and information about the neighbors on the shortest paths, thus giving us a $\tilde{O}(n^{2/3})$ total runtime.

Finally, in the "both not remote" case, the subcase where the vertices are in the same Voronoi cell takes $\tilde{O}(n^{1/3})$ time, through similar reasoning as in the last part of Section 3. However, in the other subcase, determining the cluster a vertex v is in takes $\tilde{O}(n^{2/3})$ time, as we explore via BFS to find which vertices are in the same Voronoi cell and which ones should correspond to the cluster v is in, where we may explore up to $\tilde{O}(n^{1/3})$ vertices and each requires $\tilde{O}(n^{1/3})$ time to determine which Voronoi cell it is in, along with necessary information (such as distance to the center). Checking the first condition in step 5 takes $O(1)$ time, as we have information on the centers; the other two conditions require $\tilde{O}(n^{2/3})$ time, as there may be up to $\tilde{O}(n^{1/3})$ neighboring clusters and finding the centers corresponding to those clusters takes $\tilde{O}(n^{1/3})$ time. Lastly, determining whether or not a specific edge is the minimum-rank edge between two clusters, or a cluster and a cell, takes $\tilde{O}(n^{1/3})$ time, as there are $\tilde{O}(n^{2/3})$ edges leaving a specific cluster.

Thus, this local algorithm takes $\tilde{O}(n^{2/3})$ time overall. This is another bottleneck in this algorithm, as this means that we can't make the clusters any larger, or the number of clusters any smaller (asymptotically), otherwise we bump up the runtime of the algorithm.

3.3 Attempting to remove some of the bottlenecks

Here, we discuss some of our approaches to overcoming some of these bottlenecks and achieve a faster runtime.

Our first observation is that for Algorithm 6, instead of running the Elkin-Neiman Algorithm within only the set of remote vertices, we can run it on all vertices, and keep only the edges whose endpoints are both remote vertices. Note that this cuts out the cost of figuring out which vertices are remote, thus lowering the cost of the "both remote" case down to $\tilde{O}(n^{1/3})$ (as the Elkin-Neiman Algorithm only requires a BFS to find the relevant data, and then a linear-time scan through the data to recover which edges are kept).

We first claim that the subgraph remains connected, even if we run the Elkin-Neiman Algorithm on our original graph and keep only the edges whose endpoints are both remote vertices. Suppose the subgraph is disconnected; then there are several connected components that are disconnected from each other. Consider any two such connected components that are connected by at least one edge in the original graph. If one of the edges is between a remote vertex and a non-remote vertex, then we would've added this edge in Step 6 of our global algorithm, meaning that these two connected components are actually connected, contradicting the fact that these connected components are disconnected from each other. Similarly, if one of the edges is between two non-remote vertices, then the "algorithm with a promise" should preserve a connection between these vertices, though not necessarily this edge in particular. Since the algorithm with a promise keeps only edges between non-remote vertices, any such path must be present in the output, contradicting the disconnectedness assumption.

Therefore, all edges between our connected components must have been between remote vertices. However, notice that the set of edges between remote vertices that we take is determined by the output of running the Elkin-Neiman Algorithm on our original graph. In particular, since none of the edges between our connected components showed up in the output, and all of these edges are between remote vertices, none of the edges between our connected components showed up in the output of the Elkin-Neiman Algorithm. Therefore, the Elkin-Neiman Algorithm produced an output that is disconnected, which is impossible. Thus we have reached a contradiction in all cases, and so the subgraph is connected.

In addition, we claim that we add at most $|R|+O(\epsilon n)$ edges between remote vertices via this modification, compared to the original $|R|(1+O(\epsilon))$. To see this, notice that because the output of the Elkin-Neiman Algorithm is connected and contains $n(1+O(\epsilon))$ edges, we can pick a spanning tree from this subgraph and ignore the $O(\epsilon n)$ edges not in the spanning tree for now. Then, because a tree contains no cycles, if we restrict our attention to only edges between remote vertices, we see that there are at most $|R|$ edges between remote vertices in this spanning tree, since there are $|R|$ remote vertices, and having more than $|R|$ edges between remote vertices would cause us to form a cycle. Therefore, if we add back the $O(\epsilon n)$ edges that we ignored, this gives us the upper bound of $|R|+O(\epsilon n)$ edges. Note that the difference between ϵn and $\epsilon|R|$ is at most ϵn, so we still have $n(1+O(\epsilon))$ edges in our subgraph.

This takes care of one of the bottlenecks, at least in the runtime of the local algorithm; however, the other bottleneck is harder to take care of, because computing an entire cluster and checking neighboring clusters takes quadratic time in our set-up since we have to verify the centers of all of the vertices we look through.

One proposed idea is to make the "center-finding" more symmetric - for each center a vertex finds within some search limit (like $\tilde{O}(n^{1/3})$), it also checks whether or not the center can find that vertex within the same search limit. However, this would produce a third category of vertices ("hidden" vertices, or the set of vertices which do contain a center within their search limit but cannot be found by any center within the same search limit), which might complicate things even more. We might even end up with a large number of hidden vertices; consider an example where we have a graph on n vertices, which contains a $n/2$-vertex subgraph with high expansion, and $n/(2\log n)$ paths of length $\log n$ sticking out from $n/(2\log n)$ random vertices in that $n/2$-vertex high-expansion subgraph. Note that if we choose a sublinear number of centers, then most of the vertices along these paths will be hidden, as most paths will not contain a center, in which case it's easy to find a center through BFS from the vertices along the path, but it's hard for the centers to

find them through BFS due to the centers being in the high-expansion part of the graph.

4 Random walk approach

One of the main drawbacks of using BFS, as noted in the end of the previous section, is its "asymmetry" - in particular, a vertex v_1 finding another vertex v_2 through BFS may end up searching through more vertices than if v_2 were to find v_1 through BFS, as the expansion around v_1 may be significantly higher than the expansion around v_2. This can be seen in a jellyfish-like graph, as described in the end of the previous section - the vertices in the tendrils (long paths, or areas of low expansion) can more easily find the vertices in the main body (the area of high expansion) than the other way around.

Thus, we would ideally want a search algorithm that is roughly "symmetric" - we would want to make it so that if a vertex u can find another vertex v, then v can also find u. This suggests an approach using likelihoods and random walks; the probabilities that two vertices find each other within some number of steps is roughly the same because walks can be "reversed" and the probability associated with a specific walk is inversely proportional to the products of the degrees of the vertices visited.

In particular, one of the approaches using likelihoods is the following.

Algorithm 7: Pure Likelihood Search

Input : Connected graph $G = (V, E)$, vertex $v \in V$, threshold $t(n) = O(\sqrt{n})$

Output: Outputs a set of vertices S that can be found from v using this approach, along with weights on the vertices (denoting likelihoods).

1 Initialize the likelihood $L_{v,v}$ to be 1, and the rest of the likelihoods on all other vertices to be undefined (if a likelihood is undefined, then we don't keep track of it).

2 Run a BFS-like search: Initialize a set $R_0 = S_0 = \{v\}$, and for $0 \le i \le 100 \log n$, do the following:

3 Consider the set of vertices R_{i+1} adjacent to S_i but not in any of S_0, S_1, ..., S_i. For each $u \in R_{i+1}$, consider the maximum value of $\frac{L_{v,w}}{\deg(w)-1}$, if $i > 0$, or $\frac{L_{v,w}}{\deg(w)}$, if $i = 0$, over all neighbors w of u that are also in S_i; let this maximum value be equal to m_u. If $m_u \ge t(n)^{-1} = \Omega\left(\frac{1}{\sqrt{n}}\right)$, then set $L_{v,u} = m_u$ and put u into S_{i+1}. Otherwise, leave $L_{v,u}$ as undefined and don't put u into S_{i+1}.

4 Return the set $S = S_0 \cup S_1 \cup S_2 \cup \cdots \cup S_{100 \log n}$, along with the set of values $L_{v,u}$ associated with each vertex $u \in S$.

We can see the symmetry in effect for this approach - if a vertex u can find v, then there exists a path $u \to v_1 \to \cdots \to v_i \to v$ for some i such that the likelihood computed for v, which is

$$L_{u,v} = \frac{1}{\deg(u) \prod_{j=1}^{i} (\deg(v_j) - 1)},$$

is at least $\Omega\left(\frac{1}{\sqrt{n}}\right)$. This means that if we consider the reverse of this path, $v \to v_i \to \cdots \to v_1 \to u$, the likelihood computed for u if we were to start at v would be

$$L_{v,u} = \frac{1}{\deg(v) \prod_{j=1}^{i} (\deg(v_j) - 1)} = L_{u,v} \cdot \frac{\deg(u)}{\deg(v)},$$

which is also at least $\Omega\left(\frac{1}{\sqrt{n}}\right)$, and so v will also find u.

We note that having a threshold of $\epsilon = \Omega\left(\frac{1}{\sqrt{n}}\right)$ also ensures that we don't look at too many vertices.

Lemma 4.1. *This search has runtime $\tilde{O}(t(n))$.*

Proof. We prove by induction that sizes of each of the R_i's is bounded by $\Delta t(n)$ and the sums of the likelihoods $L_{v,u}$ over all $u \in S_i$ is bounded by 1. The base case, $i = 0$, is clear.

For the inductive step, notice that since the sums of the likelihoods $L_{v,u}$ over all $u \in S_i$ is bounded by 1 and each of the vertices $u \in S_i$ must have likelihood at least $t(n)^{-1}$ in order for it to be in S_i, there are at most $t(n)$ vertices in S_i. Thus, since R_{i+1} is a subset of vertices adjacent to vertices in S_i, there are at most $\Delta t(n)$ vertices in R_{i+1}. Further, we can bound the sum of the likelihoods $L_{v,u}$ over all $u \in S_{i+1}$ as follows:

$$\sum_{u \in S_{i+1}} L_{v,u} \leq \sum_{u' \in S_i} \frac{L_{v,u'}}{\deg(u')} \cdot \deg(u') = \sum_{u' \in S_i} L_{v,u'} \leq 1,$$

where the second inequality is because each vertex in S_{i+1} has likelihood of the form $\frac{L_{v,u'}}{\deg(u')}$ for some $u' \in S_i$ adjacent to u, and each $u' \in S_i$ is adjacent to at most $\deg(u')$ vertices in S_{i+1}. This means that $|R_{i+1}| \leq \Delta t(n)$ and $\sum_{u \in S_{i+1}} L_{v,u} \leq 1$, completing the inductive step and thus the proof.

Now, since the sizes of each of the R_i's is bounded by $\Delta t(n) = O(t(n))$, this means that we only look through $O(t(n))$ vertices at each iteration, hence giving us a runtime of $\tilde{O}(t(n))$ due to the logarithmic number of iterations. □

However, one of the issues is that we don't have a guarantee on the number of vertices that we are guaranteed to search through. Indeed, it's possible that this vertex will only find $O(\log n)$ vertices: consider a structure which contains a root node r and $k = O(\log n)$ sets of vertices $L_1, L_2, \ldots, L_k$, each containing $\Delta/2$ vertices, such that r is connected to all vertices in L_1, and all vertices in L_i are connected to all vertices in L_{i+1} for all $1 \leq i \leq k-1$. Notice that if we were to run this search from any vertex, the likelihood would decay exponentially as the depth of our search increases, but we only find $O(\Delta)$ vertices each iteration. Thus, when we hit our threshold, we would only have found $O(\Delta \log n)$ vertices, which is not even polynomial in n. It's possible to try to fix this approach by instead looking at the sum of the $\frac{L_{v,w}}{\deg(w)-1}$ instead of the maximum of the $\frac{L_{v,w}}{\deg(w)-1}$ at step 3 of Algorithm 7, with roughly the same runtime (and proof of runtime). However, the main issue is that this approach might end up losing a lot of our "likelihood mass" as the depth of our search increases, particularly when dealing with cycles of odd length, and thus we cannot guarantee that we'll be able to search through sufficiently many vertices, even if the neighborhood of depth $O(\log n)$ contains a large number of vertices.

As such, we want to consider a search algorithm that is still symmetric, but one that can still search through sufficiently many vertices if the neighborhood contains a large number of vertices. The key problem is being able to balance between the amount of symmetry we have and the amount of distinct vertices that we are able to search through, in the high-expansion case.

We thus turn our attention to an algorithm called `Nibble`, designed by Spielman and Tang.

Algorithm 8: Nibble Algorithm [6]

Input : Connected graph $G = (V, E)$, vertex v, $0 < \phi < 1$, positive integer b

Output: Outputs either the empty set, or a set S of vertices with low conductance (i.e. the number of edges leaving S is low compared to the sum of degrees of vertices in S), sum of degrees at least 2^b (high enough) but at most $5(\sum_{v \in V} \deg(v))/3$ (low enough), and the likelihood of reaching each $u \in S$ from v after a certain number of steps t is high enough (at least on the order of $1/(2^b \log n)$).

1 Set a threshold $\epsilon = \Omega(1/(2^b \text{polylog}(n)))$. Initialize likelihoods $L_{u,0}$ for all vertices, so that vertex v has likelihood 1 and all other vertices have likelihood 0 (if a vertex has likelihood 0, then we don't keep track of it).

2 Repeat for $O(\log(n \log n)/\phi^2)$ iterations, keeping a time counter t (initially at 0) that increments every iteration:

3 Update likelihoods: initialize each of the likelihoods $L_{u,t+1}$ for every vertex u to 0 for the next time-step. for every vertex u that currently has positive likelihood $L_{u,t}$, add $L_{u,t}/d(v)$ to each of the likelihoods of u's neighbors. After doing this for all vertices with positive likelihood at time t, check the likelihoods of the vertices at time $t + 1$; if some likelihood is lower than ϵ, round the likelihood to 0 (and don't keep track of it this timestep).

4 If there exists an integer j such that the set of j vertices S_j with highest ratio $L_{u,t+1}/d(u)$ satisfies the conditions as specified in the output (the conductance $\Phi(S_j)$ is at most ϕ, the sum of degrees is between 2^b and $5(\sum_{v \in V} d(v))/3$, and the likelihoods are $\Omega(1/(2^b \log n))$), then return S_j. Otherwise, continue.

5 If after $O(\log(n \log n)/\phi^2)$ iterations we haven't output anything, return the empty set.

The general idea is to compute the likelihoods involved for a lazy random walk starting from v (where at each step, we have a probability $p = \frac{1}{2}$ of staying at our current vertex), truncating the likelihoods if they become too small, and then taking a set of vertices with high enough volume (in our case, because of bounded degrees, we would get at least $2^b/\Delta$ vertices) and each vertex in the set has high enough likelihood, which necessarily bounds the number of vertices in our set (as if each vertex needs likelihood $\Omega(1/(2^b \log n))$, and the sum of likelihoods is bounded by 1, then we get at most $2^b \log n$ vertices in our set). Spielman and Tang [6, Theorem 2.1] show that this algorithm runs in $O(2^b(\log^6 n)/\phi^4)$ time, or roughly linear in the number of vertices output. Furthermore, Spielman and Tang [6, Lemma 2.13] show that the truncated likeihoods $L_{u,t}$ computed for each vertex u at a timestep t differs from the normal likelihood by at most $t\epsilon \deg(u)$, which in our case is at most $t\epsilon\Delta = O(1/(\phi^2 2^b \text{polylog}(n)))$ and is thus negligible compared to the threshold of $\Omega(1/(2^b \log n))$ that we set when finding S_j.

Of course, this algorithm doesn't directly apply to our situation - the main questions we need to answer are:

1. Is this this search is roughly symmetric (if u finds v, then v can also find u)? The low conductance requirement, the threshold for cutting off likelihoods, and the fact that we can return an empty set can potentially break the symmetry that we desire.

2. Is our output a connected component? We would ideally like our output to be connected, particularly if we want to design a partitioning algorithm around this, and this is not necessarily guaranteed if we look at the set of j vertices with highest ratio $L_{u,1}/d(u)$.

One modification that we can make is instead of trying to search for the set S_j at step 4 of `Nibble`, we instead keep a separate set R_t of vertices whose likelihoods clear the ϵ bound at timestep t, and then return the union of all of the R_t's after all of our iterations, just like in Algorithm 7. Since the truncated likelihoods don't differ by much from the normal likelihoods, the search should still roughly be symmetric, up to adjusting constant factors in the algorithm (since, again, we can reverse paths and the likelihoods only change by a constant factor), and the output should be connected because any vertex in R_t must have had a

neighbor in R_{t-1} in order for it to have had nonzero truncated likelihood at time t. As such, we can envision our algorithm to do the following.

1. If we run BFS from v and don't find $\tilde{O}(n^{1/2})$ vertices within a neighborhood of depth $O(\log n/\log(1+\epsilon))$, then run the Elkin-Neiman algorithm.
2. If we do find $\tilde{\Omega}(n^{1/2})$ vertices within a neighborhood of depth $O(\log n/\log(1+\epsilon))$ (where we stop once we find $\tilde{O}(n^{1/2})$ vertices), and then we run and find $\tilde{\theta}(n^{1/2})$ vertices through modified `Nibble` (where we keep a separate set R_t of vertices whose likelihoods clear the ϵ bound at timestep t, and then return the union of all of the R_t's after all of our iterations), then we can find a center from this set of $\tilde{\theta}(n^{1/2})$ vertices that can find it in return.

However, it's not clear what we should do if this modified `Nibble` only finds $o(n^{1/2})$ vertices, even if the neighborhood around v is large. This might suggest that this vertex v is in a set of low conductance. This means that if we can find a set of vertices around v that has conductance lower than ϵ, and all but an ϵ fraction of vertices within this set can agree on being within this set of vertices, then we can separate out this set of low conductance from the rest of the graph and add all edges between this set and the rest of the graph, due to the low conductance of this set. However, we are still not fully sure how to approach this, as ensuring that all but an ϵ fraction of vertices within a set of low conductance agree on being within the set is tricky.

5 Final Remarks

The key ingredient to make an LSSG algorithm like the one discussed in Section 4 work will be a partition oracle. This oracle assigns every vertex to a connected component, and every vertex in a component is able to find the entire component containing the vertex (or at least, is able to find some superset of this component). Every component is one of three types: it has size $\Omega(\sqrt{n})$, it has low conductance, or it is a leftover vertex, but there should be few leftover vertices in total with high probability. To make a sparse spanning subgraph, we keep a spanning tree within each component, all edges leaving the low expansion components and leftover vertices, and at most one edge between each pair of large components (the lexicographically smallest such edge).

How do we create this partition oracle? We don't know, but our idea is as follows. Run some random-walk based algorithm similar to Nibble. This should assign to each vertex a set of other vertices it is likely to find in a random walk. If this set has size $\Omega(\sqrt{n})$, we're happy by symmetry of random walk probabilities: we can choose around $\sqrt{n}$ centers and use these. (But there is still a problem of connectedness: the set of vertices assigned to a center may not be connected.)

If the set of likely vertices has size $o(\sqrt{n})$, what then? Ideally, there must be small expansion/conductance happening, and we can turn that into some small conductance components that everyone can agree on, with maybe a few leftover vertices.

References

[1] Elkin, M., and Neiman, O. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Trans. Algorithms 15*, 1 (2019), Art. 4, 29.

[2] Lenzen, C., and Levi, R. A centralized local algorithm for the sparse spanning graph problem. In *45th International Colloquium on Automata, Languages, and Programming*, vol. 107 of *LIPIcs. Leibniz Int. Proc. Inform.* Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2018, pp. Art. No. 87, 14.

[3] Levi, R., Ron, D., and Rubinfeld, R. Local algorithms for sparse spanning graphs. In *Approximation, randomization, and combinatorial optimization*, vol. 28 of *LIPIcs. Leibniz Int. Proc. Inform.* Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2014, pp. 826–842.

[4] Parter, M., Rubinfeld, R., Vakilian, A., and Yodpinyanee, A. Local computation algorithms for spanners. In *10th Innovations in Theoretical Computer Science*, vol. 124 of *LIPIcs. Leibniz Int. Proc. Inform.* Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2019, pp. Art. No. 58, 21.

[5] Rubinfeld, R., Tamir, G., Vardi, S., and Xie, N. Fast local computation algorithms. arXiv:1104.1377.

[6] Spielman, D. A., and Teng, S.-H. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM J. Comput. 42*, 1 (2013), 1–26.