Near-Linear Cost Sequential and Distributed Constructions of Sparse Neighborhood Covers

Baruch Awerbuch * Bonnie Berger † Lenore Cowen ‡ David Peleg §

Abstract

This paper introduces the first near-linear (specifically, $O(E \log n + n \log^2 n)$) time algorithm for constructing a sparse neighborhood cover in sequential and distributed environments.

This automatically implies analogous improvements (from quadratic to near-linear) to all the results in the literature that rely on network decompositions, both in sequential and distributed domains, including adaptive routing schemes with $\tilde{O}(1)$ [1] stretch and memory, small edge cuts in planar graphs, sequential algorithms for dynamic approximate shortest paths with $\tilde{O}(E)$ cost for edge insertion/deletion and $\tilde{O}(1)$ time to answer shortest-path queries, weight and distance-preserving graph spanners with $\tilde{O}(E)$ running time and space, and distributed asynchronous "from-scratch" Breadth-First-Search and network synchronizer constructions with $\tilde{O}(1)$ message and space overhead (down from $O(n)$).

*Lab. for Computer Science, MIT, Cambridge, MA 02139. Supported by Air Force Contract AFOSR F49620-92-J-0125, NSF contract 9114440-CCR, DARPA contracts N00014-91-J-1698 and N00014-J-92-1799, and a special grant from IBM.

†Dept. of Mathematics and Lab. for Computer Science, MIT. Supported in part by an NSF Postdoctoral Research Fellowship and an ONR grant provided to the Radcliffe Bunting Institute.

‡Dept. of Mathematics and Lab. for Computer Science, MIT. Supported in part by Air Force Contract OSR-89-02171, Army Contract DAAL-03-86-K-0171 and DARPA Contract N00014-91-J-1698

§Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. Supported in part by an Allon Fellowship, a Bantrell Fellowship, a Minerva Fellowship and a Walter and Elise Haas Career Development Award.

[1] In the following notation, $\tilde{O}(R)$ time is $O(R \log^{O(1)} R)$ time. (We use the standard $\tilde{O}$ notation for cleaner statement of bounds in the abstract and beginning of the introduction.)

1 Introduction

An *r-neighborhood cover* is a set of overlapping "clusters" of nodes in an undirected weighted graph (network) with the property that any node has all of its nodes within distance r away belonging to one of these clusters. An r-neighborhood cover efficiently represents the local neighborhoods in a graph when all clusters in the cover are low-diameter and the overlap (or number of clusters which can intersect at a node) is minimized.

There is a tradeoff between achieving low diameter and low overlap. If the clusters are balls of radius r around ever node, all clusters have diameter r, but each node can be in as many as n clusters, where n is the number of nodes in the network. On the other hand, the entire network can be considered one giant cluster, where each node is in just the one cluster, but the diameter of this cluster can be as large as n.

Awerbuch and Peleg [11] presented a general r-neighborhood cover construction with $O(r \log n)$ cluster diameter and $O(\log n)$ cluster overlap; it was also shown in [11] how to achieve an essentially optimal trade-off between these parameters. The Awerbuch-Peleg construction, however, takes $O(nE)$ sequential and quadratic distributed time.

In this paper, we present a new construction that achieves the optimal tradeoff in near-linear time. To achieve this, we abandon the "set coarsening" method of [11], and instead show how to construct the cover only producing BFS trees from carefully selected vertices. The novel aspect of our approach is that the area of the BFS trees is carved out in such a way that one can bound the overlap with other BFS trees. We stress that as soon as $r \gg 1$, bounding the overlap between

BFS trees poses significant difficulties. In order to derive good complexity bounds, we need to charge the work performed in overlapping regions against the work performed in disjoint regions. Also, we need to lower-bound the fraction of r-neighborhoods that have "escaped carving" in a given iteration.

In the sequential domain, this algorithm makes the r-neighborhood cover, for the first time, a viable data structure for speeding up sequential algorithms. In the distributed domain, the algorithm removes the bottleneck that lay at the heart of solving many of the fundamental network control problems.

1.1 Summary of results and applications

The main results are two efficient deterministic neighborhood cover algorithms:

1. *A sequential algorithm* with running time and space $O(E \log n + n \log^2 n)$. (Down from $O(nE)$ time and space in [11].)

2. *A distributed asynchronous algorithm* with $O(\log^4 n)$ messages and space per edge. (Down from $O(n)$ messages and space in [11].)

We express all results in this paper in terms of the sequential running time, which is equivalent to the synchronous distributed implementation, where sequential running time becomes distributed message complexity. The distributed space requirement will simply be the maximum overlap of the clusters. This construction can be adapted to an *asynchronous* distributed network, where additional techniques are needed to maintain $\tilde{O}(1)$ per edge message complexity in an asynchronous environment. A synchronous cover construction is bootstrapped with a synchronizer protocol that runs on covers in a mutually-recursive fashion. For lack of space, we postpone a detailed description to the journal version of this paper.

The results in this paper *automatically* imply analogous improvements (e.g., from quadratic to near-linear) to all the results in the literature that rely on neighborhood covers as a "black box". These include adaptive routing schemes with $\tilde{O}(1)$ stretch and memory (down from $O(n^{\frac{1}{\sqrt{\log n}}})$ [5, 12]); small edge cuts in planar graphs in $\tilde{O}(E)$ running time and space (as a consequence of this paper and the algorithm in [23], down from $O(n^2)$ time and space using [11, 23]); sequential algorithms for approximating k-pairs shortest paths (see below and Section 6 for details); weight and distance-preserving graph spanners with $\tilde{O}(E)$ running time and space (see [6, 15], down from $O(nE)$ time using [14] or a combination of [6] and [11]); and a distributed *from-scratch* network synchronizer construction with $\tilde{O}(1)$ messages and space per edge (down from $\Omega(n)$ time and space using [10, 11]). We also comment that the *worst case* running times of our new algorithms are superior to the *expected* running times of the best existing randomized algorithms for these tasks, using constructions of [20, 7].

We present two example applications in Section 6: online construction of adaptive routing schemes with compact tables, and approximating k-pairs shortest paths.

Finally, it is worth mentioning that there exists a somewhat weaker notion of neighborhood covers: a small *average degree cover* [11, 9], where the *average* cluster overlap over all nodes is small. In contrast, our neighborhood cover algorithm finds a solution with small *maximum* cluster overlap over all nodes. The average degree cover problem is known to be significantly easier [11, 4]; in fact, the maximum degree cover algorithm in this paper works through a logarithmic number of iterations, each producing an average degree cover in the "remaining graph".

We remark that average degree covers do not work for implementing our algorithms in a distributed setting, since average degree covers may require a prohibitively large ($\Omega(n)$) amount of space at some of the network nodes. Moreover, average degree covers are not sufficient for an efficient solution to the k-pairs shortest paths problem, because the "imbalance" (high and low overlaps) cannot be amortized over many queries. (Unlike the algorithms in this paper, other constructions of average degree covers [1, 15] cannot be run for multiple iterations to produce a small maximum degree cover.)

However, as Afek et al. [1] and Cohen [15] have pointed out, in the special case of the *all-pairs shortest paths application*, an average degree cover suffices and leads to a better tradeoff between running time and approximation; in particular, the constant of approximation can be improved by a constant factor.

1.2 Structure of this extended abstract

In Section 2, we provide the graph-theoretic definition of neighborhood covers [11]. In Section 3, an overview of the new fast algorithm to construct the sparse neighborhood covers data structure is presented. A formal presentation of the algorithm that constructs the data structure is given in Section 4, and the analysis of the algorithm is in Section 5. Finally, in Section 6 we present two sample applications.

2 Network decompositions: definitions

The *sparse neighborhood covers* data structure was first introduced by [11] in the context of distributed computing. This paper achieves the first near-linear time construction of this data structure, thereby making it feasible for sequential computation. We define a sparse neighborhood cover as follows:

A (β, r)*-neighborhood cover* is a collection of sets (also called *clusters*) of vertices $S_1, \ldots, S_l$, with the following properties:

1. $\forall v, \exists i$ s.t. $N_r(v) \subseteq S_i$, where $N_r(v) = \{u | dist(u,v) \leq r\}$.
2. $\forall i, Diam(S_i) \leq O(\beta r)$, where $Diam(S_i) = \max_{u,v \in S_i}(dist_{S_i}(u,v))$. [2]

$N_r(v)$ is referred to as the r-*neighborhood* of v. A (β, r)-neighborhood cover is said to be *sparse*, if each vertex is in at most $\beta n^{1/\beta}$ sets.

[2] $dist_{S_i}(u,v)$ is the minimum distance from u to v in the induced subgraph S_i; $dist(u,v)$ (without the subscript) is the minimum distance in G.

We say we have a *sparse neighborhood covers* data structure when we have a family of sparse neighborhood covers for fixed β and different values of r. Typically, the applications construct a sparse neighborhood cover for $O(\log n)$ values of r, where r doubles each time from 1 to n. (This blows up all running times in Theorem 5.6 and the associated corollary by $O(\log n)$.)

Typically we set $\beta = \log n$. Then a sparse $(\log n, r)$-neighborhood cover is a collection of sets that contain all r-neighborhoods such that the diameter of the sets is bounded by $O(r \log n)$, and each vertex is contained in at most $c \log n$ sets, where $c > 0$. We remark that the diameter/sparsity tradeoff in this definition is tight to within a constant factor; there exist graphs for which any $(\log n, r)$-neighborhood cover places some vertex in at least $\Omega(\log n)$ sets [20].

Awerbuch and Peleg [11] presented the first (sequential) construction for a sparse neighborhood cover. However, their construction takes sequential time $O(nE)$ and essentially calculates all-pairs shortest paths by growing BFS trees from every vertex to achieve the cover!

3 Building the data structure: an overview

This section presents an overview of the algorithm. For clarity of exposition, we assume throughout this section that we have set $\beta = \log n$.

The algorithm builds the cover in phases: each phase produces a new subcollection of sets to be added to the cover. All sets added to the cover will be low diameter, and each vertex will appear in at most one set per phase. In addition, a constant fraction of the vertices whose r-neighborhoods remain uncovered by any set already in the cover, will have their r-neighborhoods placed in some set constructed in the current phase. After $O(\log n)$ iterationss, since each vertex appears in at most one set per phase, each vertex will appear in at most $O(\log n)$ sets. Since a constant fraction of the remaining vertices have their r-neighborhoods covered in each phase, in $O(\log n)$ phases we have covered the r-neighborhood of all vertices in the graph.

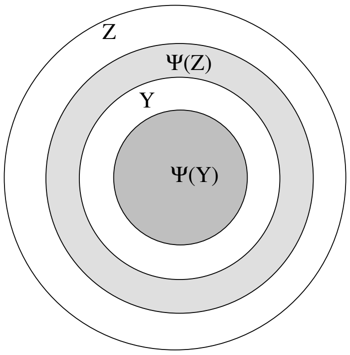

Figure 1: Cluster 1 is grown iteratively; Y, $\Psi(Z)$, and Z all grow in successive bands of width r around $\Psi(Y)$. (Note that Y, $\Psi(Z)$ and Z in this picture are balls, not empty rings, and contain by definition all nodes within their borders.) If stopping conditions are not met, at the next level Y becomes the old Z, $\Psi(Y)$ grows to $\Psi(Z)$ and the new Z contains the $(2r)$th neighborhood of the new Y. $\Psi(Z)$ consists of those nodes whose r-neighborhood lies in Z. The stopping conditions are guaranteed to be met in $O(\log n)$ levels.

Fix a phase of the algorithm, and let U be the collection of vertices whose r-neighborhood remains uncovered. Let $\mathcal{Y}$ denote the subcollection of sets we will place in the cover in the current phase.

We now sketch how to construct the sets $\mathcal{Y}$, in near-linear time. Low diameter sets Y are constructed one at a time, in a breadth-first manner, and added to $\mathcal{Y}$. The key technique is a new "guarded" breadth first search technique, which carefully controls how new sets are grown around existing sets in $\mathcal{Y}$. In particular, a two-layer "shield" is grown around each set Y. The shield serves a dual purpose: it keeps different sets in $\mathcal{Y}$ from overlapping, and it also insures that several sets do not duplicate work by performing breadth-first search forays again and again into the same area of the graph.

Section 3.1 describes how the algorithm will construct a single Y in the cover, and Section 3.2 shows how new non-overlapping sets Y grow around existing Y in the collection $\mathcal{Y}$. The code, details, and a formal analysis of the algorithm appear in Sections 4 and 5.

3.1 Growth of a single set

We show how to grow a single set Y for the cover. Y is grown iteratively, in layers, and outer layers form a shield around Y. We keep track of four layers around an "internal kernel" called $\Psi(Y)$. $\Psi(Y)$ consists of those nodes whose r-neighborhood is fully subsumed in the set Y; Z is the $2r$th neighborhood of Y; and $\Psi(Z)$ consists of those nodes whose r-neighborhood is subsumed by Z (see Figure 1). Note that $\Psi(Y) \subset Y \subset Z$ and $\Psi(Y) \subset \Psi(Z) \subset Z$. We will sometimes refer to the set Z as the set Y's associated *cluster*.

The set Y starts growing as follows: $\Psi(Y)$ is initially just the single node v, and Y is its r-neighborhood. If all stopping conditions are not met, at the next iteration, $\Psi(Y)$ becomes the old $\Psi(Z)$ and Y becomes the old Z, and the algorithm grows the BFS to construct a new Z which contains the $(2r)$th neighborhood of the new Y.

The first stopping condition says that the number of still uncovered nodes in $\Psi(Y)$, the interior of the set Y, must be a constant fraction of those in $\Psi(Z)$. Another says that the number of still uncovered nodes in Y must be a constant fraction of those in Z. These two conditions help guarantee that the cover is sparse, as will be argued below. A third stopping condition makes sure that the BFS exploration of the outer Z layer does not involve too much computation. Define $\lambda(v)$ to be the degree of node v in the graph G. Let $\lambda(V) = \sum_{v \in V} \lambda(v)$. The final stopping condition says that $\lambda(Y)$ is a constant fraction times $\lambda(Z)$. We argue below that all three stopping conditions are guaranteed to be met simultaneously in $O(\log n)$ levels. Thus, the diameter of a Y will be at most $O(r \log n)$.

3.2 A subcollection of sets

In this section, we show how a set Y and its cluster grow around previous non-overlapping Y from the same subcollection $\mathcal{Y}$ in the cover $\mathcal{X}$. A new cluster is always grown starting from a node v which lies outside the $\Psi(Z)$ of any previous Y. Notice that this insures that the r-neighborhood of v lies outside any previous Y.

When a new cluster then grows into territory already occupied by a previous cluster, its BFS growth is curtailed by the previous cluster's layers. Each of the successive shields around an existing

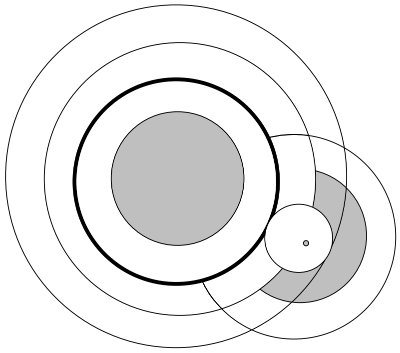

Figure 2: Cluster 2 begins growing with $\Psi(Y)$ any single node outside a previous $\Psi(Z)$. Notice this ensures that Y, the r-neighborhood of $\Psi(Y)$, will not overlap with previous Y's. Previous Y's form an impenetrable barrier (bold line) that nothing else can enter. In addition, Cluster 2's $\Psi(Z)$ (striped region) does not enter previous $\Psi(Z)$.

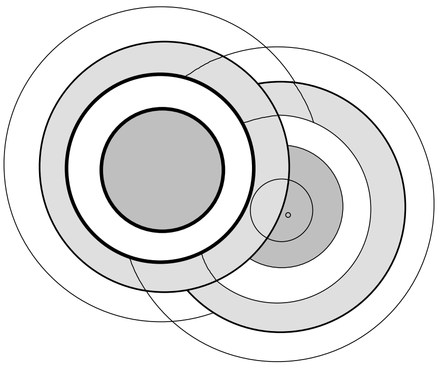

Figure 3: The second layer of Cluster 2 (i.e. the stopping condition was not met in the first layer.) Notice that the new $\Psi(Y)$ does not contain all of Y from the previous layer, since, $\Psi(Y)$ is the old $\Psi(Z)$, which does not extend through a previous cluster's $\Psi(Z)$. Even though Y and Z do not extend into the territory of a previous cluster's Y layer, Y contains the r-neighborhood of $\Psi(Y)$, and Z contains the r-neighborhood of $\Psi(Z)$.

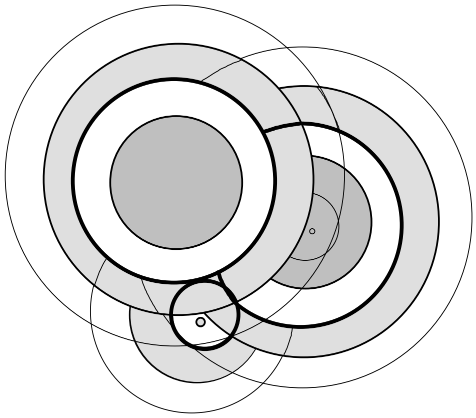

Figure 4: The third cluster begins to grow. As before, the kernels $\Psi(Y)$ are shaded, the sets Y are marked with a bold line, $\Psi(Z)$ is striped, and Z is the outside ring. Ignore for a moment when a node participates in the outer Z ring; then every node participated in the BFS's at most twice: a node can lie in at most one set Y, and prior to being placed in Y, it could lie in at most one $\Psi(Z)$. The BFS done in the Z layer, we end up charging to the Y cluster for which it was grown. When subcollection $\mathcal{Y}$ is complete, every node appears in a $\Psi(Z)$, for some cluster Z, and the stopping conditions ensure that a constant fraction of the nodes appear in $\Psi(Y)$ for some Y. Thus every node appears in at most one Y, and a constant fraction have their r-neighborhoods contained in Y as well.

cluster permit a different degree of penetration: No cluster is allowed to grow into a previous cluster's Y (this will ensure that the sets put in $\mathcal{Y}$ are disjoint). (See Figure 2.) A cluster's $\Psi(Z)$ does not grow into a previous $\Psi(Z)$. The Z level is entirely permeable. (We control the cost of repeated BFS forays into previous Z instead by a stopping condition on the growth of a single cluster.) We point out, to aid comprehension, that the relation $Y \subseteq \Psi(Z)$ may not hold for clusters grown after the first cluster (see Figure 3).

Figures 2 – 4 give snapshots of how new clusters grow. We stop growing clusters in the subcollection $\mathcal{Y}$ when every node is in $\Psi(Z)$ for some cluster Z.

This ends the overview of the algorithm. The next section gives precise code for the procedures, and Section 5 proves correctness, and gives the running times for general β.

4 Details of building the data structure

In this section, the algorithm `All_Cover` is introduced. Given β and r, `All_Cover` constructs a sparse (β, r)-neighborhood cover (see Section 2 for the definition.) For the applications, β is typically fixed to control the tradeoffs between complexity and the quality of the approximation (we typically set $\beta = \log n$), and then `All_Cover` is called with $O(\log n)$ different r's to produce $O(\log n)$ different sparse neighborhood covers.

`All_Cover` is built from two intermediate subprocedures: Procedure `Cluster` (see Figure 5), which grows a single set to be placed in the cover, and Procedure `Cover`, which calls `Cluster` to produce a subcollection $\mathcal{Y}$ of the sets in the Cover $\mathcal{X}$ (see Figure 6).

The input to the algorithm is a graph $G = (V, \mathcal{E})$ (where $|V| = n$ and $|\mathcal{E}| = E$) and integers $r, \beta \geq 1$. The output collection of cover clusters, $\mathcal{X}$, is initially empty. The algorithm maintains the set of "remaining" nodes R. These are the nodes whose neighborhoods are not yet subsumed by the constructed cover. Initially $R = V$, and the algorithm terminates once $R = \emptyset$. The code for algorithm `All_Cover` appears in Figure 7.

> $\Psi(Z) \leftarrow \{v\}$
> $Z \leftarrow \{N_r(v)\}$
> **repeat**
> $\Psi(Y) \leftarrow \Psi(Z)$
> $Y \leftarrow Z$
>   Perform a multi-origin BFS w.r.t. Y
>   to depth $2r$ in $G(V \setminus \bigcup \mathcal{Y})$
>   Add all nodes encountered to Z
> $\Psi(Z) \leftarrow \{v \mid v \in U \cap Z,\ dist(v, Y) \leq r\}$
> **until** $|Z| \leq n^{1/\beta} \cdot |Y|$
>   **and** $\lambda(Z) \leq n^{1/\beta} \cdot \lambda(Y)$
>   **and** $|\Psi(Z)| \leq |R|^{1/\beta} \cdot |\Psi(Y)|$
> **return** $(\Psi(Y), Y, \Psi(Z), Z)$

Figure 5: Procedure `Cluster`$(R, U, v, \mathcal{Y})$.

The algorithm operates in at most $\beta n^{1/\beta}$ phases. Each phase consists of the activation of the procedure `Cover`(R), which adds a subcollection of output clusters $\mathcal{Y}$ to $\mathcal{X}$ and removes the set of nodes $\Psi(R)$ whose r-neighborhood appears in some $Y \in \mathcal{Y}$ from R.

5 Analysis of the algorithm

In this section, we first analyze the properties of the procedures in Section 4 and prove that algorithm `All_Cover` constructs a sparse neighborhood cover. The complexity analysis then follows in Section 5.2.

5.1 Correctness of the algorithm

Lemma 5.1 A set Y produced by procedure `Cluster` will have diameter at most $O(4\beta r)$.

Proof: Recall the three stopping conditions on the growth of a cluster. If the first or second stopping conditions to fail to be met, this means that $|Z| \geq n^{1/\beta} \cdot |Y|$ in the case of the first stopping condition or $|\Psi(Z)| \geq |R|^{1/\beta} \cdot |\Psi(Y)|$ in the case of the second. Since each time the stopping condition fails, we increase what is inside by a factor of $n^{1/\beta}$, each of these stopping conditions can fail at most β times. The third stopping condition says that $\lambda(Z \setminus Y)$ is less than or equal to $n^{1/\beta}$ times

```
R ← V                          /* R is the collection of remaining (unsubsumed) balls */
𝒳 ← ∅                          /* 𝒳 is the output cover */
repeat
    (Ψ(R), 𝒴, 𝒵) ← Cover(R)    /* invoke procedure Cover */
    𝒳 ← 𝒳 ∪ 𝒴
    R ← R \ Ψ(R)
until R = ∅
return 𝒳
```

Figure 7: Algorithm All_Cover.

```
U ← R
Ψ(R) ← ∅.
𝒴, 𝒵 ← ∅.
while U ≠ ∅ do:
    Select an arbitrary node v ∈ U.
    (Ψ(Y), Y, Ψ(Z), Z) ← Cluster(R, U, v, 𝒴)
    Ψ(R) ← Ψ(R) ∪ Ψ(Y).
    U ← U \ Ψ(Z).
    𝒴 ← 𝒴 ∪ {Y}.
    𝒵 ← 𝒵 ∪ {Z}.
end-while
return (Ψ(R), 𝒴, 𝒵)
```

Figure 6: Procedure Cover(R)

$\lambda(Y)$, and since the sum of the degrees of the entire graph is at most $E \leq n^2$, the third stopping condition can fail at most 2β incremental times. Therefore, the number of iterations of the growth process for which *any* of the stopping conditions fail is at worst 4β, and so, for one of the first $4\beta+1$ steps at which we might grow a cluster, *all* three stopping conditions will hold, and the cluster will stop growing. Since each time we grow Y, we add an additional distance r to the radius, the diameter of the sets is bounded by $O(4\beta r)$. □

Lemma 5.2 The r-neighborhood of any vertex in a cluster's Y layer is contained in $\Psi(Z)$ for that cluster, or some previous cluster's $\Psi(Z)$ layer.

Proof: `Cluster` adds a vertex in Y's r-neighborhood to $\Psi(Z)$, unless it is missing from set U. U is initially the entire graph G, and procedure `Cover` only deletes those nodes from set U which lie in some previous cluster's $\Psi(Z)$ layer. □

Lemma 5.3 For all $w \in G$, each invocation of procedure `Cover` places w into at most one set Y.

Proof: By contradiction. Suppose w was placed in Y_1 and Y_2, and without loss of generality, let Y_1 be constructed by `Cover` before Y_2. Note that since $w \in Y_1$, then $w \in \Psi(Z)$ for Y_1. When `Cover` constructs Y_2, it picks some center vertex v and invokes procedure `Cluster`. We consider three cases: w is v, w is in the r-neighborhood of v, and w is not in the r-neighborhood of v. w cannot be v, since w is in $\Psi(Y)$ for Y_1, and thus `Cover` has removed w from the set U of vertices from which it chooses centers v. In fact, since v is outside $\Psi(Z)$ for all previous Y, its entire r-neighborhood cannot contain anything in a previous Y, by Lemma 5.2. Thus w is not in the r-neighborhood of v, and so it will not be placed into Y when `Cluster` is initialized. As `Cluster` grows cluster Y_2, by construction, it will only place vertices w into Y_2 that it first places in the Z layer. However, members of previous Y are barred from entering into any subsequent Z layer, by construction. □

Lemma 5.4 An $O(n^{1/\beta})$ fraction of the vertices whose r-neighborhood still remains to be covered (i.e. vertices in R) have their r-neighborhoods covered each time `All_Cover` calls procedure `Cover`. Thus procedure `Cover` is invoked $O(\beta n^{1/\beta})$ times.

Proof: The vertices that lie in $\Psi(Y)$ for some cluster Y are the vertices that have their r-neighborhoods covered. A stopping condition guarantees that for each cluster, a $O(n^{1/\beta})$ fraction of the vertices in $\Psi(Z)$ (in R) lie in $\Psi(Y)$. Summing over all clusters produced in one invocation of `Cover`, since every vertex in R is in some $\Psi(Z)$ (the termination condition for `Cover`), and the Y, and hence the $\Psi(Y)$ are all disjoint (by Lemma 5.3) gives the result. Hence, after $O(\beta n^{1/\beta})$ iterations of procedure `Cover`, all vertices have their r-neighborhoods covered, and algorithm `All_Cover` returns the cover. □

Theorem 5.5 Given a graph $G = (V, \mathcal{E})$ (where $|V| = n, |\mathcal{E}| = E$), and integer parameters $\beta, r \geq 1$, algorithm `All_Cover` constructs a sparse neighborhood cover.

Proof: By Lemma 5.4, `All_Cover` will cover all neighborhoods after $O(\beta n^{1/\beta})$ iterations of procedure `Cover`. All clusters in the cover are constructed by procedure `Cluster`, and hence are low-diameter, by Lemma 5.1. Finally, by Lemma 5.3, each vertex appears in at most one set for each invocation of procedure `Cover`, and by Lemma 5.4, `Cover` is invoked at most $O(\beta n^{1/\beta})$ times. Thus each node is in at most $O(\beta n^{1/\beta})$ sets, and the cover is sparse. □

This ends the proof of correctness; we now turn to complexity.

5.2 Complexity analysis

We compute the cost of checking the stopping conditions and then performing the appropriate BFS explorations to construct $\mathcal{Y}$. The stopping conditions need to be checked at most $O(\beta)$ times for each cluster, since the conditions will be met in $O(\beta)$ levels. To check the stopping conditions, we count nodes in Z, $\forall Z \in \mathcal{Z}$. Recall one of the stopping conditions insures that for all Z the number of nodes in Z is less than a $O(n^{1/\beta})$ fraction times the number of nodes in Y. Therefore, $O(\sum_{Z\in\mathcal{Z}} |Z|) = O(n^{1/\beta} \sum_{Y\in\mathcal{Y}} |Y|) = O(n^{1+1/\beta})$, since the Y are disjoint. Thus, the cost of checking the stopping conditions to construct $\mathcal{Y}$ is $O(\beta n^{1+1/\beta})$.

Now we turn to the BFS's. Ignore for a moment for all clusters, the cost of the BFS exploration of the final layer $Z \setminus \Psi(Z)$. Then for the construction of $\mathcal{Y}$, each node participates at most twice in BFS's: each edge is placed in at most one set Y, and prior to being placed in Y, it could lie in at most one $\Psi(Z)$. Thus without the Z layer, each edge has been examined at most twice in the construction of $\mathcal{Y}$.

Now consider the complexity of examining edges in Z. Notice that when $r = 1$ this is trivial, since every edge in $Z \setminus \Psi(Z)$ has an endpoint in $\Psi(Z)$, and every node is in a unique $\Psi(Z)$. For $r > 1$, bounding the amount of work done in Z is controlled by means of another stopping condition on the growth of individual clusters. In particular, if we have done too much work on exploring the layer $Z \setminus Y$, we are then obligated to grow Y to contain Z.

Recall $\lambda(v)$ is the degree of node v in the graph G, and $\lambda(V) = \sum_{v\in V} \lambda(v)$. The final stopping condition on clusters ensures $\lambda(Z \setminus Y)$ is less than or equal to $O(n^{1/\beta})$ times $\lambda(G)$ which is equal to $2E$. Together with the stopping condition on the size of Z, this gives $\sum_{Y\in\mathcal{Y}} |Z \setminus Y| \leq O(n^{1/\beta} E)$.

Constructing $\beta n^{1/\beta}$ subcollections $\mathcal{Y}$ to complete the cover $\mathcal{X}$, we thus obtain the following theorem:

Theorem 5.6 Procedure `All_Cover` producs a sparse neighborhood cover in time $O((E\beta + n\beta^2)n^{2/\beta})$.

Setting $\beta = \log n$, we obtain the following corollary:

Corollary 5.7 Procedure `All_Cover` produces a sparse neighborhood cover in time $O(E \log n + n \log^2 n)$.

6 Example applications

6.1 Routing

As a typical example of how to apply the network decomposition data structure in the distributed domain, we focus on a classic network application: *online* construction of adaptive routing schemes with compact tables [4, 5, 21]. Routing schemes have been around since [18, 19], but they have not been efficiently constructible until this paper. Many of the proposed schemes rely on the assumption of the existence of a data structure that can be constructed using small space [4, 22, 13, 8, 11]. However, the more efficient schemes of [8, 11] required linear space per node to set up the data structure. This was a serious bottleneck, particularly in the adaptive scenario, since routing tables have to be recomputed every time there is a change in network topology (or traffic conditions). The space-efficient construction of sparse covers in this paper provides the missing piece.

More specifically, prior to this paper, the best adaptive routing schemes appeared in [4, 5]. However, these schemes feature an unfavorable (roughly speaking, exponential) tradeoff between their communication overheads and space requirements; using these schemes one can achieve stretch and memory requirements of around $O(n^{\frac{1}{\sqrt{\log n}}})$. The more efficient schemes of [8, 11] feature a polynomial trade-off, allowing $\tilde{O}(1)$ stretch and run-time memory. However, their preprocessing space requirements were high, namely $O(n)$ per node, and in the adaptive model, this dominates the cost. The algorithms presented in this paper enable the best of both worlds, namely, the same trade-off as in [8, 11], with preprocessing requiring the same amount of space as during run-time.

6.2 Approximate shortest paths

The *k-pairs shortest paths* problem is: given an undirected graph with nonnegative edge weights, and k pairs of vertices, find the shortest paths between these pairs of vertices.

The best-known bounds for exact k-pairs shortest paths run Dijkstra's algorithm in time $O(kE \log n)$ with a binary heap implementation [16], in time $O(kn \log n + kE)$ with a Fibonacci heap implementation [17], or compute all-pairs shortest paths via matrix multiplication in $\tilde{\mathrm{O}}(M(n))$ time, where $M(n)$ is the time for matrix multiplication (currently known to be $o(n^{2.376})$); for example, see [2, 24, 3]. The bottlenecks in the running time of the exact algorithms come either from repeated application of Dijkstra's single-source shortest paths algorithm or from matrix multiplication.

As an immediate corollary of our near-linear cost construction of sparse neighborhood covers, we obtain an $O(\log n)$-approximation algorithm for this problem that runs in $\tilde{\mathrm{O}}(E + k)$ time, thereby bypassing the bottlenecks of both these approaches at the cost of producing an approximate solution. Set $\beta = \log n$. Our new algorithm is very simple:

- in the preprocessing stage, it constructs an r-neighborhood covers data structure (see Section 2 for the definition) in $\tilde{\mathrm{O}}(E)$ time using our algorithm, and
- shortest path quieries are answered in $\tilde{\mathrm{O}}(1)$ time and with logarithmic precision by simply determining the smallest cluster containing the desired pair of vertices (through a binary search).

Our improvement in running time is due to our ability to construct a sparse neighborhood covers data structure in near-linear time. A tradeoff can be achieved between quality of the approximation and running time; for example, a 32β approximation takes $O((E\beta + n\beta^2)n^{2/\beta} \log n + k\beta n^{1/\beta} \log \log n)$ time, and, in the sequential setting, the approximation constant can be further improved by eliminating some of the stopping conditions of the algorithm. Note that we can achieve a better running time than known exact algorithms even for paths that are only a constant factor times the length of the shortest path. Further work in this direction is reported in [15].

Acknowledgments

We thank Yehuda Afek, Edith Cohen, Tom Leighton and Moti Ricklin for stimulating discussions.

References

[1] Y. Afek and M. Riklin. Sparser: A paradigm for running distributed algorithms. *J. of Algorithms*, 1991. Accepted for publication.

[2] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all-pairs shortest path problem. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, Oct. 1991.

[3] N. Alon, Z. Galil, O. Margalit, and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, Oct. 1992.

[4] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Compact distributed data structures for adaptive network routing. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 230–240, May 1989.

[5] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *J. of Algorithms*, 11:307–341, 1990.

[6] B. Awerbuch, A. Baratz, and D. Peleg. Efficient broadcast and light-weight spanners. Unpublished manuscript, May 1993.

[7] B. Awerbuch, B. Patt, D. Peleg, and M. Saks. Adapting to asynchronous dynamic networks with polylogarithmic overhead. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 557–570, 1992.

[8] B. Awerbuch and D. Peleg. Routing with polynomial communication-space trade-off. Technical Memo TM-411, MIT, Lab. for Computer Science, Sept. 1989.

[9] B. Awerbuch and D. Peleg. Efficient distributed construction of sparse covers. Technical Report CS90-17, Weizmann Institute, July 1990.

[10] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.

[11] B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[12] B. Awerbuch and D. Peleg. Routing with polynomial communication - space trade-off. *SIAM J. on Discr. Math.*, 1991. To appear. Also in Technical Memo TM-411, MIT, Sept. 1989.

[13] A. E. Baratz and J. M. Jaffe. Virtual circuits in large computer networks. *Computer Networks*, pages 27–37, Dec. 1986.

[14] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. In *Proc. 8'th ACM Symposium on Computational Geometry*, 1993.

[15] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. these proceedings, 1993.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[17] M. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. of the ACM*, pages 596–615, 1987.

[18] L. Kleinrock and F. Kamoun. Hierarchical routing for large networks; performance evaluation and optimization. *Computer Networks*, 1:155–174, 1977.

[19] L. Kleinrock and F. Kamoun. Optimal clustering structures for hierarchical topological design of large computer networks. *Computer Networks*, 10:221–248, 1980.

[20] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 320–330. ACM/SIAM, Jan. 1991.

[21] D. Peleg. Distance-dependent distributed directories. *Info. and Computation*, 1993. Also in Tech. Report CS89-10, The Weizmann Institute, May 89.

[22] D. Peleg and E. Upfal. A tradeoff between size and efficiency for routing tables. *J. of the ACM*, 36:510–530, 1989.

[23] S. Rao. Finding small edge cuts in planar graphs. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 229–240, 1992.

[24] R. Seidel. On the all-pairs-shortest-path problem. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 745–749, May 1992.