A Near-Optimal Distributed Fully Dynamic Algorithm for Maintaining Sparse Spanners

[Extended Abstract]

Michael Elkin
Department of Computer Science, Ben-Gurion University of the Negev,
Beer-Sheva, Israel.
elkinm@cs.bgu.ac.il *

ABSTRACT

Currently, there are no known explicit algorithms for the great majority of graph problems in the *dynamic distributed message-passing model*. Instead, most state-of-the-art dynamic distributed algorithms are constructed by composing a static algorithm for the problem at hand with a simulation technique that converts static algorithms to dynamic ones. We argue that this powerful methodology does not provide satisfactory solutions for many important dynamic distributed problems, and this necessitates developing algorithms for these problems *from scratch*.

In this paper we develop a *fully dynamic* distributed algorithm for maintaining *sparse spanners*. Our algorithm improves drastically the *quiescence time* of the state-of-the-art algorithm for the problem. Moreover, we show that the quiescence time of our algorithm is optimal up to a small constant factor. In addition, our algorithm improves significantly upon the state-of-the-art algorithm in *all efficiency parameters*, specifically, it has smaller quiescence message and space complexities, and smaller local processing time. Finally, our algorithm is self-contained and fairly simple, and is, consequently, amenable to implementation on unsophisticated network devices.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on Discrete Structures; G.2.2 [**Graph Theory**]: Network Problems

General Terms

Algorithms

*This research has been supported by the Israeli Academy of Science, grant 483/06. Additional funding was provided by the Lynn and William Frankel Center for Computer Sciences.

PODC'07, August 12–15, 2007, Portland, Oregon, USA.

Keywords

Spanners, Distributed Dynamic Algorithms

1. INTRODUCTION

In the *message-passing model* of distributed computing a communication network is modeled by an unweighted graph, and each vertex of this graph hosts a processor. The vertices communicate through network links, modeled by graph edges. Devising algorithms for the message-passing model (henceforth, *algorithms*) is an active area of research [1]-[14], [30]-[34]. (See [32] for an excellent survey.) The main motivation for the study of this model is the urge to devise real-life protocols for the growing body of communication networks, particularly the Internet. However, ever since the theoretic study of distributed algorithms started in the early eighties, it was commonly understood [7, 14, 1, 3, 8, 10, 4] that real-life networks are inherently *dynamic*, i.e., the communication links may crash and revive at will. The necessity to devise algorithms that are sufficiently robust to work properly in an unstable environment became even more apparent in view of such recent developments in the world of telecommunications as *ad-hoc, sensor, and wireless networks*, which are all inherently unstable.

Moreover, the resurgence of these novel network architectures dictates certain limitations on algorithms that can be used. In particular, these algorithms have to use *limited computational resources*, such as local processing time and space, just because the processors in these networks may well be *uncapable* of undertaking a heavy computational task. Even more importantly, these algorithms need to be *simple*, due to inherent computational limitations of (cheap) processors that are supposed to execute them.

Currently, there are no known explicit dynamic distributed algorithms for the great majority of distributed graph problems. Instead, most dynamic distributed algorithms are constructed by composing a static distributed algorithm for the problem at hand with a simulation technique that converts static algorithms to dynamic ones. (There are important exceptions. See the paragraph titled "Related Work".) However, the currently known simulation techniques all suffer from significant drawbacks. The simulation technique of [1, 12] has time and space overheads which are super-linear in the number of vertices n; the technique of [14] has message and space overheads which are super-linear in n. Awerbuch et al. [10] has only polylogarithmic time, message, and space overheads, but entails a large overhead in local pro-

cessing. More importantly, the technique of [10] is extremely complex, and incorporates several very involved procedures. In particular, it uses a composition of a *reset procedure* [1, 11], an algorithm for constructing *sparse neighborhood covers* [12], a *bootstrap* technique [12], and a *local rollback* algorithm [14]. Consequently, the algorithm of [10] is unsuitable for running on simple network devices.

These considerations raise the need to develop dynamic distributed algorithms for basic algorithmic problems *from scratch*, or to develop alternative simulation techniques. In this paper we follow the first avenue, and develop an *extremely efficient* fully dynamic distributed algorithm for maintaining sparse *spanners*. Informally, graph *spanners* can be thought of as sparse skeletons of communication networks that approximate to a significant extent the metric properties of the respective networks. Spanners serve as an underlying graph-theoretic construct for a great variety of distributed algorithms. Their most prominent applications in this context include *synchronization* [5, 33, 12, 10], *routing* [34, 13, 38], *approximate distance computation* [21, 23], and *online load balancing* [9].

The state-of-the-art distributed *static* algorithm for constructing sparse spanners is a distributed variant of the algorithm of Baswana and Sen [16]. (An explicit description of this algorithm can be found in Baswana et al. [15].) For an arbitrary positive integer parameter t, and arbitrary undirected unweighted n-vertex graph, this algorithm constructs $(2t-1)$-spanner with expected $O(t \cdot n^{1+1/t})$ edges. (See Section 2 below for the definition of spanner.) The running time of this algorithm is $O(t)$, its message complexity is $O(|E| \cdot t)$, and its space requirement for a vertex v running the algorithm is $O(deg(v) \cdot \log n)$ bits, where $deg(v)$ is the degree of the vertex v.

The common way to measure performance of distributed dynamic algorithms is through *quiescence* complexities. Specifically, suppose that all topology updates stop occurring at a certain time α, and that the algorithm reaches a state in which the (distributed) structure that it is maintaining starts again to satisfy the properties of the problem at hand at time $\beta \geq \alpha$. The worst-case difference $\beta - \alpha$ is called the *quiescence time complexity* of the the algorithm, and the worst-case number of messages that are sent during the time period $[\alpha, \beta]$ is called the *quiescence message complexity* of the algorithm. (See Section 2 below for a more precise definition.)

Running the algorithm of Baswana and Sen [16, 15] on top of the simulation technique of Awerbuch et al. [10] results in a dynamic distributed algorithm for maintaining a $(2t-1)$-spanner of expected size $O(t \cdot n^{1+1/t})$ with quiescence time complexity of $O(t \cdot \log^3 n)$, quiescence message complexity of $O(t \cdot |E| \cdot \log^3 n)$, and space requirement of $O(deg(v) \cdot \log^4 n)$.

Our Results: We devise a fully dynamic asynchronous algorithm for maintaining a $(2t-1)$-spanner of the same expected size that improves the result of [16, 10] described above in *all efficiency parameters*. In particular, our result drastically reduces *the quiescence time* from $O(t \cdot \log^3 n)$ to $3t$. [1] The quiescence message complexity of our algorithm is $O(t \cdot |E|)$, and its space requirement is $O(deg(v) \cdot \log n)$ bits. The expected processing time-per-edge of our algorithm is $O(1)$. See Table 1 for a concise comparison between the different efficiency parameters of our algorithm with the state-of-the-art algorithm of [16, 10].

We also show that the quiescence time of our algorithm is near-optimal. Specifically, we show that for any constant parameter t any distributed algorithm that maintains sparse $(2t-1)$-spanners of near-optimal size in a fully dynamic setting has quiescence time greater or equal to $\lfloor \frac{2}{3}t \rfloor$. In fact, our lower bound applies even for *static synchronous* algorithms that are allowed to send messages of *arbitrarily large* size, while messages sent by our algorithm are all of size $O(\log n)$. Moreover, under Erdős girth conjecture [24] our lower bound can be improved to $t-1$. The lower bound extends also to super-constant values of t. We remark also that devising a dynamic algorithm with better complexity guarantees than that of our algorithm would also improve the state-of-the-art *static* algorithms for the problem.

To summarize, our algorithm compares very favorably to the state-of-the-art benchmark algorithm that combines the simulation technique of [10] with the distributed variant of [16]. No less important is that our algorithm is reasonably simple, in contrast to the simulation algorithm of [10]. The incremental variant of our algorithm is *extremely simple*, and can be hard-wired even in the most primitive network devices. The general variant of our algorithm is more complex, but nevertheless, is amenable for implementation on unsophisticated network devices.

Our Techniques: Our main technical contribution is a novel approach to handle edge crashes. The standard approach is the rollback technique [7, 1, 14, 8, 11, 10, 4], which requires each vertex to maintain some part of the history of communication, and to undo certain operations from this history once an edge crash occurs. Maintaining history is a significant burden on the processors of the network. In the context of ad-hoc, sensor, and wireless networks the processors are usually uncapable of carrying this burden. Our approach is based on looking for a "replacement" for every crashing edge. While our algorithm also undoes certain operations, the history of communication is never explicitly maintained. Rather the list of operations to be undone is *deduced from the current state of affairs*. We believe that this method of undoing operations without maintaining an explicit history of communication is the key to the extreme robustness and efficiency of our algorithm.

Additional features of our algorithm: Our algorithm has a number of additional features. First, not only that its quiescence time is at most $3t$, but if at a time α edges stop crashing but are still allowed to appear, still at time $\alpha + 3t$ the spanner maintained by the algorithm will provide a stretch guarantee of $2t-1$ for all edges of the graph present in the network at time α. (Edges that appeared at some time β, $\beta > \alpha$, will be taken care of by time $\beta + 3t$.)

Second, our algorithm behaves even better in a purely incremental and purely decremental environments. In the incremental case the quiescence time of our algorithm becomes $2t$ instead of $3t$. In addition, this quiescence time decreases further if the sets of edges that appear in the network possess a certain convenient structure. In particular, if the set F of edges that appear in the network is a matching, then our algorithm takes care of all edges F within *one single round* or *two time units* after these edges appear, depend-

[1] In many cases the parameter t is constant, and it is always true that $t = O(\log n)$. Also, the constant hidden by the O-notation in $O(t \cdot \log^3 n)$ is at least one order of magnitude greater than 3.

	Quiescence Time Complexity	Quiescence Message Complexity	Space Requirement	Local Processing Time-per-edge
[16, 10]	$O(t \cdot \log^3 n)$	$O(t\|E\| \cdot \log^3 n)$	$O(deg(v) \cdot \log^4 n)$	Not clear
New	$3t$	$O(t\|E\|)$	$O(deg(v) \cdot \log n)$	Expected $O(1)$
l.b.	$\lfloor \frac{2}{3} t \rfloor$	$\Omega(\|E\|)$	$\Omega(deg(v))$	$\Omega(1)$

Table 1: A comparison between the state-of-the-art algorithm of [16, 10] and our new dynamic distributed algorithm. The space requirement is for a specific vertex v. The shortcut "l.b." stands for "lower bounds".

ing on whether the network is synchronous or asynchronous, respectively. The expected size of the spanner keeps to be bounded by $O(t \cdot n^{1+1/t})$ all through this process. Note that the set F may contain as many as $n/2$ edges appearing all around the network, and moreover, the topology updates may arrive in *multiple bursts*. Nevertheless, they are processed by the algorithm *on the fly*. This result extends also to more complex topology update edge sets than matchings. Specifically, if the update edge set F has maximum degree Δ then it is processed by the algorithm within 2Δ rounds or time units. In the decremental setting our algorithm performs better if each of the update sets F of crashing edges has cardinality $o(n^{1/t})$. In this case the *expected* quiescence time of our algorithm is $1 + o(1)$ time units. We remark, however, that the bound of $3t$ on the quiescence time of our algorithm requires no assumptions. Assumptions are needed only if one is interested in yet stronger bounds.

Finally, the incremental variant of our algorithm has expected quiescence message complexity of $O(|E|)$. In particular, this result improves the state-of-the-art *static* distributed algorithm of Baswana and Sen [16, 15] whose message complexity is $O(t \cdot |E|)$.

The Structure of the Paper: Even though all our results apply to asynchronous setting, in this extended abstract we only consider the synchronous setting. Terminology and definitions that are used throughout the paper are presented in Section 2. In Section 3 we describe an incremental variant of our algorithm. In Section 4 we overview its extension to the fully dynamic setting.

Related Work: In a companion paper [22] similar techniques are used for devising efficient streaming and fully dynamic centralized algorithms for maintaining sparse spanners. Dynamic distributed algorithms for *shortest paths* computation were studied in [6, 26, 27, 36, 18]. Dynamic routing in trees was studied in [2, 3, 17, 28, 29]. Dynamic broadcast and slide mechanisms were studied in [7] and [4], respectively.

2. PRELIMINARIES

For a parameter α, $\alpha \geq 1$, called a *stretch parameter*, a subgraph G' of the graph $G = (V, E)$ is called an *α-spanner* of G if for every pair of vertices $x, y \in V$, $dist_{G'}(x, y) \leq \alpha \cdot dist_G(x, y)$, where $dist_G(u, w)$ denotes the distance between vertices u and w in G. Also, for a fixed value of $t = 1, 2, \ldots$, we say that a subgraph G' *spans* an edge $e = (v, u) \in E$, if $dist_{G'}(v, u) \leq 2t - 1$.

In the message-passing model a processor v can send and receive messages over each edge $e = (v, u)$ adjacent to v in the graph. Communication links of the network are assumed to have a *limited capacity*, and this is modeled by assuming that the size of each single message is at most $O(\log n)$. For convenience, we assume that every vertex v has a unique identifier $I(v) \in [n] = \{1, 2, \ldots, n\}$. (Henceforth, for any positive integer k, the set $\{1, 2, \ldots, k\}$ is denoted $[k]$, and the set $\{0, 1, \ldots, k\}$ is denoted $[(k)]$.)

In the *synchronous model* the communication occurs in *discrete rounds*. On each round each vertex v is allowed to send messages to all its neighbors in the graph G. A message sent by a vertex v over an edge $e = (v, u)$ on round R arrives to its destination u before round $R + 1$ starts. In the *asynchronous model* each vertex maintains its own clock, and clocks of different vertices may disagree. Nevertheless, the vertices are assumed to work locally in a manner similar to the one in which they work in the synchronous model. Specifically, on each cycle of its clock ("round") each vertex v processes the messages that it has received since the last time it was in the receiving mode (the beginning of the previous clock cycle), and sends new messages that result from this processing. Messages sent from a vertex to its neighbor arrive within finite but unpredictable time period, called the *time unit*.

The *time complexity* of a synchronous (respectively, asynchronous) algorithm $\mathcal{A}$ is the maximum number of rounds (resp., time units) that an execution of $\mathcal{A}$ lasts. The *message complexity* is the maximum overall number of messages sent during an execution of $\mathcal{A}$. The *space complexity* is the maximum number of bits used by a certain vertex at a single moment during an execution of $\mathcal{A}$. The *local processing time-per-round* is the maximum duration of a time period that some vertex spends on processing data *locally* on a single clock cycle (or round). In algorithms that we will consider each vertex v processes edges adjacent to v separately, and thus it makes sense to consider also the *local processing time-per-edge*.

In a dynamic setting edges of the graph are allowed to appear and disappear (henceforth, *crash*) at any time. The challenge in this context is to design algorithms that are robust to this ever-changing environment. An *incremental* (respectively, *decremental*) dynamic algorithm is an algorithm that can handle edge appearances (resp., crashes) but not crashes (resp., appearances). A *fully dynamic* algorithm is an algorithm that can handle both appearances and crashes. The usual way to measure performance of dis-

tributed dynamic algorithms is through *quiescence time and message* complexities, that is, the time and message requirements after the last topological change. More precisely, if at a certain time α the network stops experiencing topology updates, then it is required that at a certain later time β, $\beta \geq \alpha$, all vertices running the algorithm reach a certain final state. In a final state every vertex v knows which of the edges adjacent to it belong to the spanner, and which do not. Moreover, at this point v knows that it is in a final state, and stops sending messages. Later, after leaving the final state, it may resume sending messages. [1] The worst-case possible duration $\beta - \alpha$ of the time period $[\alpha, \beta]$ is called the *quiescence time* complexity of the algorithm. The worst-case possible number of messages sent during this time period is called the *quiescence message* complexity of the algorithm.

Throughout the paper we consider topology updates that involve only appearances and crashes of *edges*. Our algorithm can cope with *vertex* appearances and crashes as well. The algorithm handles a vertex crash exactly as it handles the crash of all edges adjacent to the crashing vertex. Vertex appearances are handled analogously. When an edge $e = (u, v)$ crashes or appears, both its endpoints u and v are notified by the underlying link-layer protocol.

We assume that the number L of crashes of edges of certain type is at most polynomial in n. If this assumption does not hold, our guarantee on the size of the spanner maintained by the algorithm would grow by a factor $\frac{\log L}{\log n}$. We believe that this limitation is not really restrictive because one can use fresh coin tosses after, say, n^{10} rounds elapse. Our algorithm is sufficiently robust to guarantee that this switch to new coins does not require to coordinate the processors (in the asynchronous scenario).

We also assume that a message is lost only if the edge through which it was sent crashes. Moreover, in our model messages sent through a fixed edge $e = (v, u)$ in a fixed direction arrive to their destination in the First-In-First-Out (FIFO) manner. This assumption can be eliminated at the expense of using a more complicated analysis.

In addition, vertices are assumed to know the value of the parameter t. We also assume that the number of vertices n is known in advance, in the beginning of computation. This assumption can be weakened so that only an upper bound $\hat{n}$ on the number of vertices wil be known at the beginning of computation, but then $\hat{n}$ would have to be plugged in all our bounds instead of n. Also, our algorithm can be adapted so that the vertices will be only assumed to know some common value $q > 1$, and will not be assumed the number of vertices n or an estimate $\hat{n}$. This variant of the algorithm maintains $O\left(\frac{\log n}{\log q}\right)$-spanner of size $O(n \cdot \frac{\log n}{\log q} \cdot q)$, and its quiescence time complexity is $O\left(\frac{\log n}{\log q}\right)$. Note that in this variant of the algorithm the stretch of the spanner depends on n, but a near-optimal tradeoff between the stretch of the spanner and its size is maintained at all times, regardless of the fluctuations of n. (Let $\gamma = 1 + 1/t$ denote the exponent in the size $O(t \cdot n^{\gamma})$ of the spanner. Then the tradeoff is optimal if $(\gamma - 1)t = \Theta(1)$.) For an illustration, suppose that in the beginning vertices know the value q, set (by an operator that knows n and t) as $q = n^{1/t}$. As a result the algorithm constructs an $O(t)$-spanner with expected size $O(t \cdot n^{1+1/t})$. Suppose that the number of vertices n is permanently growing. Then the stretch $t = O\left(\frac{\log n}{\log q}\right)$ grows logarithmically in n, but the exponent $\gamma - 1 = \frac{\log q}{\log n} = 1/t$ decays inverse-logarithmically in n.

Finally, we assume that the input graph, the topology updates and their order, are all chosen by a *non-adaptive adversary* obliviously of the coin tosses of the algorithm. Intuitively, this setting can be thought of as a fair simultaneous 2-player game between the algorithm and the adversary. The game takes place right before the beginning of the execution. The algorithm tosses all coins that it will use during the execution, and the adversary decides what is the input, which topology updates will take place, and at which time moments. This model captures the situation in which edges and vertices appear and disappear in a way that depends on a current structure of the graph, but does *not* depend on the current structure of the spanner that the algorithm maintains. Although this assumption is not completely realistic, we believe that our model reflects truthfully many practical situations. In addition, this assumption is significantly weaker than an assumption of some stochastic model for topology updates, and the latter is very common in the literature (see [35], and the references therein). Finally, all existing (static or dynamic) distributed algorithms for constructing spanners that run in time smaller than the diameter of the network and use messages of bounded size [19, 16, 15] rely on a similar assumption.

[1] Note that this setting is different the self-stabilizing scenario [20] in which messages are being sent at all times.

3. THE INCREMENTAL ALGORITHM

We start this section with an overview of the *incremental* variant of our algorithm. We then describe its extension to the fully dynamic setting. As was already mentioned, in this extended abstract we only consider *synchronous* setting.

As a part of the initialization step, each vertex picks a non-negative integer *radius* $r(v)$ for every vertex v of the graph from the *truncated geometric probability distribution* given by $\mathbb{P}(r = k) = p^k \cdot (1 - p)$, for every $k \in [(t - 2)]$, and $\mathbb{P}(r = t - 1) = p^{t-1}$, with $p = n^{-1/t}$. Note that this distribution satisfies $\mathbb{P}(r \geq k + 1 \mid r \geq k) = p$ for every $k \in [(t - 2)]$. Employing this probability distribution for distributed algorithms was introduced in [31]. In the context of spanners related distributions were used in [19, 37, 16, 25].

During an execution of the algorithm, every vertex v maintains the variable $P(v)$, called the *label* of v, initialized as the identifier of v, $I(v)$. The labels of vertices may grow as the execution proceeds, and they accept values from the set $\{1, 2, \ldots, n \cdot t\}$. A label P in the range $i \cdot n + 1 \leq P \leq (i+1)n$, for $i \in [(t - 1)]$, is said to be a label of *level* i; in this case we write $L(P) = i$. The value $B(P)$ is given by $B(P) = n$ if n divides $P(v)$, and by $B(P) = P(v) \mod n$, otherwise. This value is called the *base value* of the label P. See Figure 1. The vertex $w = w_P$ such that $I(w) = B(P)$ is called the *base vertex* of the label P. A label P is said to *exist* if the level $L(P)$ of P is no greater than the radius of the base vertex w_P, i.e., $L(P) \leq r(w_P)$. The label P is called *selected* if $L(P) < r(w_P)$. Note that for a label P to be selected, it must satisfy $L(P) \leq t - 2$.

An alternative way to see a label P is as a pair $(B(P), L(P))$ such that $P = n \cdot L(P) + B(P)$. Intuitively, the base value $B(P)$ determines the identity of a *tree* τ to which the vertex v labeled by P belongs currently, and the value $L(P)$ is equal to the distance between v and the root w_P of τ.

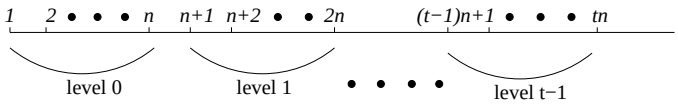

Figure 1: Labels of different levels. A label $in + j$ has base value j and level i.

The radius $r(w_P)$ of w_P reflects the maximum distance to which the label of w_P is allowed to propagate in the network during an execution of the algorithm. When a label P is adopted by a vertex from one of its neighbors, the level of P is incremented, and its base value is preserved. However, the level of P is not allowed to grow beyond $r(w_P)$.

One of the basic primitives of the algorithm is comparing labels. We say that the labels $P(v)$ and $P(v')$ of the vertices v and v', respectively, satisfy the relation $P(v) \succ P(v')$ if and only if either $P(v) > P(v')$ or $(P(v) = P(v')$ and $I(v) > I(v'))$. Note that for every two vertices v and v', either $P(v) \succ P(v')$ or $P(v') \succ P(v)$.

We remark that the way that we define and manipulate labels is closely related to the way it is done in Feigenbaum et al. [25].

Every vertex maintains an edge set $Sp(v)$, initialized as an empty set. During the execution the algorithm inserts some edges into $Sp(v)$, and never removes them. In other words, in the incremental variant of the algorithm the sets $Sp(v)$ grow monotonely as the execution proceeds. It is useful to think of the sets $Sp(v)$ as divided into two disjoint subsets $T(v)$ and $X(v)$, $Sp(v) = T(v) \cup X(v)$. The set $T(v)$ is called the set of *tree edges* of the vertex v, and the set $X(v)$ is called the set of *cross edges* of the vertex v. During the execution the algorithm constructs implicitly a *tree cover* of the graph. Edges of this tree cover are maintained in the sets $T(v)$. In addition, the spanner will also contain some edges that connect different trees of the tree cover; these edges are maintained in the sets $X(v)$. Each edge e that is inserted into the set $T(v)$ will also be labeled by the label of v at the time of the insertion. An insertion of an edge $e = (v, u)$ into the set $T(v)$ will cause v to change its label to the label of u plus n, that is, $P(u) + n$. The edge e will also be labeled by this label.

In addition, every vertex v maintains a table $M(v)$. These tables are initially empty. Each table $M(v)$ is used to store all the base values of levels P such that there exists at least one neighbor z of v that was labeled by P at some point of the execution of the algorithm, and such that the edge (v, z) was inserted into the set $X(v)$ at that point of the execution. The vertex v also maintains a variable $ttl(v)$, standing for the "time-to-live", which reflects the maximum distance to which the current label of v, $P(v)$, is allowed to propagate. Finally, the vertex v maintains a counter $ctr(v)$, which records the number of rounds that elapsed since the last topology update detected by v. The vertex v is active only as long as $ctr(v) \leq 2t$. Once the counter becomes greater than $2t$, the vertex becomes dormant, and its only activity from that point on is to monitor edges adjacent to it. Once a dormant vertex detects an appearance of a new edge, it resets its counter, and becomes active again.

Next, we describe the algorithm itself. In the beginning of the execution, or upon detecting an appearance of a new edge, the vertex v invokes the Procedure *Round* for $2t$ rounds in a row. During these rounds v computes its set $Sp(v)$ which is the output of the algorithm. In other words, $Sp = \bigcup_{v \in V} Sp(v)$ is the sparse spanner returned by the algorithm. Note that the Procedure *Round* accepts no input parameters, and consequently, it runs in the same way on every round of the algorithm. In addition, there is just one possible type of messages that can be sent by vertices that run the algorithm. (These are messages $(P(v), ttl(v))$, where v is the sender of the message.) This uniformity of the algorithm makes it extremely robust to dynamic changes of the networks.

The Procedure *Round* iterates over all messages $(P(u), ttl(u))$ that the host vertex v received in the current round. For each such a message the algorithm tests whether the condition $P(u) \succ P(v)$ holds. The messages are considered in an arbitrary order. If this is not the case, the message is skipped. Otherwise, the algorithm tests whether $P(u)$ is a selected label. If $P(u)$ is a selected label, the edge e is inserted into the set of tree edges $T(v)$ of v, and the vertex v adopts the label $P(u)+n$. Intuitively, this adoption of label means that the vertex v joins the tree τ to which the vertex u currently belongs as a leaf-child of u. If $P(u)$ is not a selected label, it means that the vertex u is not allowed to acquire children in the tree τ. In this case the procedure tests whether the base value $B(P(u))$ is stored in the table $M(v)$. Intuitively, if this is the case then there is already an edge in $X(v)$ connecting v to the tree τ. If it is not the case, then the edge e is inserted into the set $X(v)$, the base value $B(P(u))$ is inserted into the table $M(v)$, and the label of v does not change. If $B(P(u))$ is already stored in $M(v)$ then nothing needs to be done, because there is another edge $e' = (v, u') \in X(v)$ that connects v to the tree τ. Along with a subpath of τ connecting u with u', the edge e' guarantees that the constructed spanner spans the edge e. See Figure 2.

The pseudo-code of the algorithm is provided below. The initialization step is described by the first two lines.

1: $P(v) \leftarrow I(v)$; $ctr(v) \leftarrow 1$
2: Select $r(v)$ from the distribution described above; Set $ttl(v) \leftarrow r(v)$
3: On every round do (lines 4-9 in an infinite loop)
4: **if** some edge $e = (v, z)$ appeared on previous round **then**
5: $\quad ctr(v) \leftarrow 1$
6: **end if**
7: **if** $(ctr(v) \leq 2t)$ **then**
8: $\quad$ Invoke Procedure *Round*; Set $ctr(v) \leftarrow ctr(v) + 1$
9: **end if**

Procedure *Round*:

1: Go over all received messages in an arbitrary order and do
2: **while** $\exists$ message $(P(u), ttl(u))$ with $P(u) \succ P(v)$ **do**
3: $\quad$ **if** $(ttl(u) > 0)$ **then**
4: $\quad\quad P(v) \leftarrow P(u) + n$; $ttl(v) \leftarrow ttl(u) - 1$; $Sp(v) \leftarrow Sp(v) \cup \{e\}$

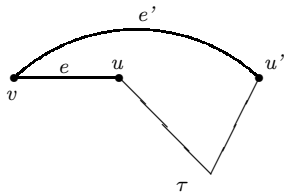

Figure 2: The edges e and e' connect the vertex v to the tree τ. If $e' \in X(v)$ then there is no need to insert e into the spanner because $\tau \cup \{e'\}$ spans e.

5: **else if** $B(P(u)) \notin M(v)$ **then**
6: $M(v) \leftarrow M(v) \cup \{B(P(u))\}$; $Sp(v) \leftarrow Sp(v) \cup \{e\}$
7: **end if**
8: **end while**
9: Send to all your neighbors the message $(P(v), ttl(v))$

We say that an edge $e = (w, w') \in E$ is *scanned* by the algorithm if at some point of the execution, either the algorithm run by w (meaning that $v = w$) substitutes $u = w'$ and passes the condition of the while loop (and gets into the loop), or the algorithm run by w' (meaning that $v = w'$) does the same with $u = w$. In the former (respectively, latter) case we say that the edge is scanned by w (resp., w'). We say that a vertex w *reads* an edge $e = (w, w')$ on round j, if either the vertex w scans it on round j or if it discovers that $P(w') \prec P(w)$ (and thus does not scan it).

Since each radius $r(v)$ is at most $t-1$, all vertex labels have levels in the set $[(t-1)]$. Also, whenever the label $P(v)$ of a vertex v grows, the level $L(P(v))$ of the label grows as well. Consequently, the label of a given vertex may grow on at most $t-1$ different rounds. Hence for every edge $e = (w, w')$, there are at most $2t-2$ rounds on which either $P(w)$ or $P(w')$ grows. It follows that in any $2t$ consequent rounds on which the edge e is present, there will necessarily be at least one round other than the first round on which both labels $P(w)$ and $P(w')$ stay unchanged. On this round either w or w' (the endpoint whose label is smaller) will scan the edge e. It is not hard to see that once the edge is scanned, it is either inserted into the spanner, or the spanner contains a path of length $2t-1$ connecting its endpoints. We conclude that the algorithm maintains a $(2t-1)$-spanner.

For analyzing the size of the spanner we observe that an edge e adjacent to a vertex v is added to the set $T(v)$ only when the level $L(P(v))$ of v grows. Consequently, at all times $|T(v)| \leq t-1$, and the number of all tree edges in the spanner is never greater than $n \cdot (t-1)$. We use the notion "backbone" to refer to this set of edges. It is easy to see that any edge of the graph has a probability of at most $p = n^{-1/t}$ to belong to the backbone.

To analyze the number of non-backbone (or cross) edges that belong to the spanner, we partition each edge set $X(v)$ to t subsets $X^{(i)}(v)$, $i \in [(t-1)]$, defined as follows. For an index $i \in [(t-1)]$, let $X^{(i)}(v)$ denote the set of edges (u, v), with $L(P(u)) < t-1$, inserted into $X(v)$ during the period of time that $L(P(v))$ was equal to i. Obviously, $X^{(t-1)}(v) = \emptyset$, and the expected number of edges (u, v) with $L(P(u)) = t-1$ inserted into $X(v)$ is at most the expected number of existing distinct labels of level $t-1$, which is $O(n/p^{t-1}) = O(n^{1/t})$.

Moreover, the expected size of $X^{(i)}(v)$ for $i \in [(t-2)]$ is bounded by $O(p^{-1}) = O(n^{1/t})$ as well. Intuitively, for $X^{(i)}(v)$ to contain at least h edges, for some integer $h \geq 1$, the vertex v needs to read h unselected labels with distinct bases *in a row*, without reading even a single selected label in-between. However, each label is selected with probability p, and for labels of distinct bases these events are independent. Hence the expected value of h is $O(p^{-1}) = O(n^{1/t})$. We conclude that for every vertex v, the expected size of $X(v)$ is $O(t \cdot n^{1/t})$, and hence the expected size of the spanner is $O(t \cdot n^{1+1/t})$.

Also, with minor modifications this algorithm achieves expected quiescence message complexity of $O(|E|)$. (Note that a trivial worst-case bound on the message complexity is $O(t \cdot |E|)$.) Specifically, we need Procedure *Round* to process messages $(P(u), ttl(u))$ with unselected $P(u)$ before it processes messages with selected $P(u)$. In addition, in the new variant of the algorithm a vertex v that scans an edge $e = (v, u)$ will notify the other endpoint u that the edge e is scanned, and the two vertices will stop sending messages along this edge.

It is easy to verify that the analysis above can be carried through for this variant of the algorithm too. In addition, this variant of the algorithm satisfies the following property. Consider an edge $e = (v, u)$ such that the larger label among $P(v)$ and $P(u)$ is not selected in the beginning of a fixed round R, and such that the edge e was not scanned before round R. This edge will necessarily be scanned by the algorithm on round R. Since for any edge e that is not scanned in the beginning of round R, its probability to satisfy this property is $1-p$, it follows that each such an edge e is scanned on round R with probability at least $1-p$. Hence the expected number of rounds that a given edge e is not scanned is at most $\frac{1}{1-p} = O(1)$, and consequently, the expected number of messages sent through e is at most $O(1)$ as well. By linearity of expectation, the expected number of messages sent by the algorithm is $O(|E|)$. We remark that by a similar, though somewhat more involved, argument it can be shown that the number of messages sent by the algorithm is $O(|E|)$ with high probability.

Next, we argue that the quiescence time decreases drastically if we restrict the sequence of topology updates to a certain convenient structure. In particular, if the set of edges added on each round forms a matching, then the quiescence time becomes *one single round*.

Suppose that all edges already present in the network were scanned before the update set F arrives. Consider a vertex v that detects a new edge $e = (v, u) \in F$, adjacent to v. Since F is a matching, there is at most one such an edge for every vertex v. Consequently, either v or u (whoever has smaller label) scans e on the same round it appears. It follows that as long as on every round the set of new edges

appearing on this round forms a matching, the algorithm takes care of each appearing edge on the same round on which it appears. Moreover, these considerations extend to the scenario when each vertex has degree at most Δ in the update set F. Specifically, in this case the quiescence time of the algorithm becomes $\min\{2t, 2\Delta - 1\}$.

Finally, we show that the bound of $2\Delta - 1$ cannot be improved using our algorithm as is. Consider the following directed graph $\hat{F}$. The vertex set of the graph consists of the "central vertex" v, the vertices $u(1), u(2), \ldots, u(\Delta)$, the vertices $z(1), z(2), \ldots, z(\Delta)$, and the vertices $w(1,1), w(1,2), \ldots, w(1,\Delta-2), w(2,1), \ldots, w(2,\Delta-2), \ldots, w(\Delta,1), \ldots, w(\Delta,\Delta-2)$. The edge set contains the oriented edges $\{\langle u(i), v\rangle \mid i \in [\Delta]\}$, the edges $\{\langle z(i), w(i,j)\rangle \mid i \in [\Delta], j \in [\Delta-2]\}$, the edges $\{\langle u(i), w(i,j)\rangle \mid i \in [\Delta], j \in [\Delta-2]\}$, and, finally, $\{\langle z(i), u(i)\rangle \mid i \in [\Delta]\}$. See Figure 3.

We next describe an execution scenario of our algorithm. Suppose that the undirected underlying graph F of $\hat{F}$ is the update edge set provided to the algorithm on round R. An edge is oriented from a vertex x to a vertex y if and only if $P(x) \prec P(y)$. Suppose that on round R the labels of $V(\hat{F})$ satisfy the order relationship determined by the directed graph $\hat{F}$. The maximum degree of F is Δ.

On the first round (after detecting the edges of F) each vertex $u(i)$, $i \in [\Delta]$, scans the edge $(u(i), w(i,1))$, and each vertex $z(i)$, $i \in [\Delta]$ scans the edge $(z(i), w(i,1))$. (The vertex v scans nothing as all its adjacent edges are incoming.)

On the second round each $u(i)$ scans the the edge $(u(i), w(i,2))$, and each $z(i)$ scans $(z(i), w(i,2))$, etc., for each round $j = 1, 2, \ldots, \Delta - 2$, on round j the edges $(u(i), w(i,j))$ and $(z(i), w(i,j))$ for all $i \in [\Delta]$ are scanned. Hence after $\Delta - 2$ rounds we are left with the graph F' in which only the edges $\{(z(i), u(i)), (u(i), v) \mid i \in [\Delta]\}$ are present.

Moreover, suppose that on round $\Delta-2$ each edge $(z(i), u(i))$ "switches orientation". This can happen if each $z(i)$, $i \in [\Delta]$, adapted the label of the "late" $w(i, \Delta - 2)$, and this label is greater than that of $u(i)$. The resulting edge set is $\{\langle u(i), z(i)\rangle, \langle u(i), v\rangle \mid i \in [\Delta]\}$.

On the $(\Delta - 1)$st round every vertex $u(i)$ scans edges $(u(i), z(i))$, $i \in [\Delta]$, and labels of $u(i)$ grow sufficiently for all the remaining edges $(u(i), v)$ to switch orientation. We then obtain the star $\{\langle v, u(i)\rangle \mid i \in [\Delta]\}$. Now v takes its time scanning each of the Δ edges adjacent to it one by one. Consequently, in this scenario $2\Delta - 1$ rounds are required to the algorithm to scan all edges of a set F of maximum degree Δ, and the bound $2\Delta - 1$ is tight.

4. THE FULLY DYNAMIC ALGORITHM

We next extend our algorithm to the fully dynamic setting. This is done in two stages. First, we consider the *semi-decremental scenario* in which an edge $e = (w, w')$ is allowed to crash only if it does not belong to the backbone. Next we extend the algorithm to the setting that allows arbitrary edge crashes.

To motivate the semi-decremental setting further we note also that when an edge $e = (w, w')$ is scanned, it joins the backbone only if the larger of the two labels $P(w)$ and $P(w')$ is selected. The probability of the latter event is at most $p = n^{-1/t}$. Hence the edge e belongs to the backbone with probability at most p, and thus the the semi-decremental setting is quite powerful by itself. In addition, an algorithm for this setting provides a good starting point for the most general setting.

The main modification needed to adapt the incremental algorithm to the semi-decremental setting has to do with the way that a vertex v acts when it scans an edge $e = (v, w)$ and discovers that another edge $e' = (v, w')$ connecting v to the same tree was already scanned. In the incremental variant of the algorithm v plainly discards the edge e. In the semi-decremental setting this course of action is problematic because the edge e' may later crash, and thus v needs to keep the edge e as a replacement edge for e'.

For this end each vertex v maintains a set $M(v)[B]$ for every base value B that belongs to the set $M(v)$. This set contains all edges $e = (v, w)$ that were scanned by v, and such that v discovered that $B(P(w)) \in M(v)$. Such edges are said to be *dropped by* v, and the set of those edges is denoted $D(v)$. Upon a crash of an edge $e' = (v, w') \in X(v)$, the vertex v replaces e' by an arbitrary edge $e = (v, w)$ from the set $M(v)[B]$, if $M(v)[B] \neq \emptyset$. Interestingly, if $M(v)[B]$ is empty it means that there are no longer any edges *in the graph* connecting v to the tree B. Thus, in this case there is no problem with the fact that the spanner contains no such edges. If this is the case, the value B is removed from $M(v)$. Also, if a dropped edge $e = (v, w) \in M(v)[B]$ crashes, it is removed from the set $M(v)[B]$.

Observe that since every non-backbone edge $e = (v, w)$ is scanned either by v or by w, then either it belongs to $X(v) \cup X(w)$, or there exists a base value B such that $e \in M(v)[B]$ or $e \in M(w)[B]$. Consequently, these simple rules capture all possible crashes of non-backbone edges. Remarkably, no matter how many edges of this type crash simultaneously, their crashes can be processed *locally*, with no communication whatsoever! This is particularly surprising since the great majority of all edges do not belong to the backbone. Indeed, the overall number of backbone edges is at most $n \cdot (t-1)$, and the probability of any particular edge to belong to the backbone is at most $p = n^{-1/t}$.

Also, we remark that essentially by the same argument as in the incremental setting, the expected quiescence message complexity of our algorithm in the semi-decremental setting is $O(|E|)$.

Next, we overview the way our algorithm handles crashes of backbone edges. When no backbone edges crash, vertex labels grow monotonely as the execution proceeds. This is no longer the case in the most general scenario. In particular, if a vertex v detects that the edge (v, u) connecting v to its parent u in one of the trees to which v belongs crashes, then the label P that v adopted from u earlier becomes meaningless. This is because the label P indicates that the vertex v belongs to the same tree as u does, and once the edge (v, u) crashed this is no longer the case. In this scenario the vertex v has to drop its current label, and retrieve the label it had before scanning the edge e.

To perform this retrieval, v maintains a table $A(v)$ of *active* labels. This table may contain at most one label for each of the t levels. A label P is called an *active* label for v if it belongs to $A(v)$, or, equivalently, if v was labeled by P at some point, and did not drop the label since then. In addition to dropping the label P, the vertex v drops also all its active labels that are greater than P. Intuitively, the rationale for dropping these labels is that they were acquired on the grounds that v was labeled by P, and once P became obsolete there is no reason to keep these other labels.

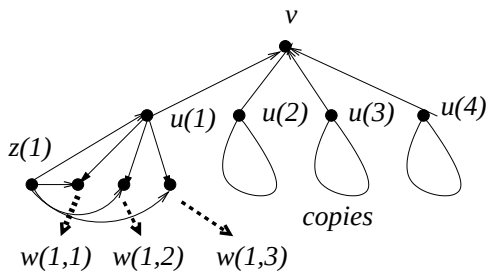

Figure 3: The graph $\hat{F}$ with $\Delta = 4$. The three "sacks" depict three isomorphic copies of the graph induced by the vertices $z(1), u(1), w(1,1), w(1,2), w(1,3)$.

Additionally, for every edge $e = (v, w)$ scanned by the vertex v, the vertex v maintains either the label $P(v) = P(w) + n$ that it acquired from w while scanning e, or, if it did not acquire any label at that time, the label $P(w)$ of w at the time that v scanned e. This label will be referred to as the label of e. The vertex v also maintains a status variable indicating whether this edge was inserted into $T(v)$, $X(v)$, or $D(v)$. Depending on the set to which it was inserted, the edge e will be referred to as a T-edge, X-edge, or D-edge, respectively.

In the scenario that was described above, after the vertex v drops all labels greater or equal to P, it also sends the message *CRASH* over all T-edges adjacent to v that connect it to its neighbors in trees to which it no longer belongs. It also sends the same message over the X- and D-edges that were scanned when v was labeled by one of the dropped labels. (Obviously, if one of these edges itself crashes, nothing is, and nothing can be, sent through it. The algorithm is, however, sufficiently robust to handle these situations. The details are provided in the full version of the paper.)

A child z of v in one of these trees treats the message *CRASH* received over the edge (v, z) in the same way it would have treated an actual crash of the edge (v, z). In other words, the subroutine that we described for v is now invoked recursively in z, and in all other children of v in trees to which it no longer belongs. We will refer to this event by saying that the vertex z *experiences a super-crash* of the edge (v, z).

The parents of v in all these trees, as well as its neighbors that are connected to v via X- or D-edges of the particular type described above, treat the message *CRASH* received from v in a different way. Unlike the children of v that propagate the message *CRASH*, these neighbors update their local data structures to reflect the fact that the edge connecting them to v can no longer be used for reaching the trees to which v stopped belonging. Further details can be found in the full version.

The key property of this algorithm that is used in the analysis is that the message *CRASH* never propagates beyond the $(t-1)$-neighborhood of its originator. Consider again the vertex v that detects a crash of the backbone edge (u, v), labeled by P. Let $\{T_1, T_2, \ldots, T_q\}$, for some integer q, $1 \leq q \leq t - L(P)$, be the set of trees in which v has an active label larger or equal to P. The message *CRASH* originated in v makes its children in $T_1, T_2, \ldots, T_q$ to experience super-crashes of edges connecting them to v. The crucial observation is that the distance of a child z of v in a tree T_i, $i \in [q]$, from the root r_i of T_i is at least $L(P) + 1$. This is because for every index $i \in [q]$, the label P_i satisfies $L(P_i) \geq L(P)$, and the level of the label of z in T_i is precisely $L(P_i) + 1$. The upper bound of $t - 1$ on the propagation radius of the message *CRASH* follows now directly from the fact that all trees maintained by the algorithm are of radius at most $t - 1$.

The small locality radius of the algorithm can by itself serve a strong evidence that the algorithm has a small quiescence time. Indeed, intuitively, within t rounds every vertex that should be "concerned" with a certain topology update knows about it, and can undertake appropriate actions. The rigorous analysis is, however, significantly more involved for a number of reasons. First, many edge crashes may occur simultaneously, and yet many other edges may crash while the algorithm takes care of those edges that have just crashed. These crashes may prevent a proper dissemination of the crash information, which, in turn, may prevent vertices from undertaking appropriate actions for correcting the spanner. Second, the edges in our model have a very limited capacity ($O(\log n)$ bit per message), and this capacity limitation by itself may cause severe bottlenecks that could prevent a proper dissemination of crash information.

Finally, we describe a variant of our algorithm in which vertices need to know neither the number of vertices n nor an estimate $\hat{n}$ of this number. Instead the vertices are assumed to know a common value q, $q > 1$. In this variant the probability p is set as $p = q^{-1}$, and each radius $r(v)$ is selected from the geometric distribution given by $\mathbb{P}(r(v) = k) = p^k \cdot (1 - p)$, for $k = 0, 1, 2, \ldots$. (The main difference from the previous variant of the algorithm is that the geometric distribution is used instead of the truncated geometric one.)

To analyze this variant we observe that with high probability, for every vertex $v \in V$, $r(v) = O\left(\frac{\log n}{\log q}\right)$. More specifically, for any $\epsilon > 0$, $\mathbb{P}\left(\forall\ v,\ r(v) \leq (1+\epsilon) \cdot \frac{\log n}{\log q}\right) \geq 1 - n^{-\epsilon}$. Consequently, by previous argument, with high probability the algorithm maintains $O\left(\frac{\log n}{\log q}\right)$-spanner. Also, for each vertex v and level $i = 0, 1, 2, \ldots, r(v)$, the expected number of X-edges that v scans while $L(P(v)) = i$ is q. Hence the expected number of X-edges adjacent to v in the spanner is $O\left(q \cdot \frac{\log n}{\log q}\right)$, and the expected size of the spanner is $O\left(n \cdot q \cdot \frac{\log n}{\log q}\right)$. The quiescence time complexity of the algorithm is, by previous argument, $O(\max\{r(v) \mid v \in V\})$, which is at most $O\left(\frac{\log n}{\log q}\right)$, with high probability.

We remark also that if one allows an additional factor of $O((\log n)^{1-1/t})$ (or $O(\log n)$ for the last variant of the algorithm) then all bounds on the *expected* size of the spanner can be replaced by bounds that apply *with high probability*.

5. OPEN QUESTIONS

We present a sample of open questions that are related to our study.

1. Perhaps the most obvious open question is to close the constant gap between our upper and lower bounds of $3t$ and $\frac{2t}{3}$, respectively, on the best possible quiescence time of a dynamic algorithm for maintaining sparse spanners.

2. Another avenue is to weaken the assumptions on computational model that are needed for our result. Specifically, in this paper the adversary that determines the topology updates is assumed to be non-adaptive and oblivious to the coin tosses of the algorithm. Devising an algorithm with properties similar to that of our algorithm in the presence of a more powerful adversary appears to be a challenging problem.

3. Our algorithm is randomized, and randomization is crucial for our analysis. Derandomizing our result is a challenging problem as well.

4. An interesting open problem is to improve our bound on the expected size of constructed spanners from $O(t \cdot n^{1+1/t})$ to $O(n^{1+1/t})$. Note, however, that this problem is open even for *static* algorithms.

5. To this day there are only a handful of distributed algorithms for basic graph-theoretic problems that are provably applicable to a dynamic environment. We hope that our study will stimulate further development of this important area. Particularly, it would be very interesting to devise dynamic distributed algorithms for such basic problems as *MST*, *MIS*, and vertex cover.

Acknowledgements

The author is grateful to Eitan Bachmat, Shlomi Dolev, Joan Feigenbaum, David Peleg, Seth Pettie, Oded Regev, Liam Roditty, and Jian Zhang for helpful and stimulating discussions. The author also thanks the support of Lynn and William Frankel Center for Computer Sciences.

6. REFERENCES

[1] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th Symp. on Foundations of Computer Science*, pages 358–370, 1987.

[2] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks. Local management of a global resource in a communication network. In *Proc. of the 28th IEEE Annual Symp. on Foundations of Computer Science*, pages 347–357, 1987.

[3] Y. Afek, E. Gafni, and M. Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In *Proc. of 30th IEEE Annual Symposium on Foundations of Computer Science (FOCS-89)*, pages 370–375, 1989.

[4] Y. Afek, E. Gafni, and A. Rosen. The slide mechanism with applications in dynamic networks. In *Proc. of the 11th ACM Symp. Principles of Distributed Computing*, pages 35–46, 1992.

[5] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 4:804–823, 1985.

[6] B. Awerbuch, I. Cidon, and S. Kutten. Communication-optimal maintenance of replicated information. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 492–502, 1990.

[7] B. Awerbuch and S. Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16:381–396, 1986.

[8] B. Awerbuch, O. Goldreich, and A. Herzberg. A quintitive approach to dynamic networks. In *Proc. of the 9th ACM Symp. on Principles of Distributed Computing*, pages 189–203, 1990.

[9] B. Awerbuch, S. Kutten, and D. Peleg. Online load balancing in a distributed network. In *Proc. 24th ACM Symp. on Theory of Comput.*, pages 571–580, 1992.

[10] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. E. Saks. Adapting to asynchronous dynamic networks. In *Proc. of the 24th Annual ACM Symp. on Theory of Computing*, pages 557–570, 1992.

[11] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local cheking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 268–277, 1991.

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

[13] B. Awerbuch and D. Peleg. Routing with polynomial communication-space tradeoff. *SIAM J. Discrete Mathematics*, 5:151–162, 1992.

[14] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 206–220, 1988.

[15] S. Baswana, T. Kavitha, K. Mehlhorn, and S. Pettie. New constructions of (a,b)-spanners and additive spanners. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 672–681, 2005.

[16] S. Baswana and S. Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(n^{1+1/k})$ size in weighted graphs. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 384–396. Springer, 2003.

[17] S. Cicerone, G. D. Stefano, and M. Flammini. Static and dynamic low-congested interval routing schemes. *Theoretical Computer Science*, 276:315–354, 2002.

[18] S. Cicerone, G. D. Stefano, D. Frigione, and U. Nanni. A fully dynamic algorithm for distributed shortest paths. *Theoretical Computer Science*, 297:83–102, 2003.

[19] E. Cohen. Fast algorithms for t-spanners and stretch-t paths. *SIAM J. Comput.*, 28:210–236, 1999.

[20] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[21] M. Elkin. Computing almost shortest paths. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 53–62, 2001.

[22] M. Elkin. A streaming and fully dynamic centralized algorithm for maintaining sparse spanners. In *Proc. of the Internat. Colloq. Autonata, Languages, and Progr. (to appear)*, 2007.

[23] M. Elkin and J. Zhang. Efficient algorithms for constructing $(1+\epsilon, \beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18:375–385, 2006.

[24] P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and Applications (Proc. Sympos. Smolenice)*, pages 29–36, 1964.

[25] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: The value of space. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, pages 745–754, 2005.

[26] P. Humblet. Another adaptive distributed shortest path algorithm. *IEEE Transactions on Communications*, 39(6):995–1003, 1991.

[27] G. F. Italiano. Distributed algorithms for updating shortest paths. In *Proc. Intern. Workshop on Distributed Algorithms*, pages 200–211, 1991.

[28] A. Korman and D. Peleg. Labeling schemes for weighted dynamic trees. In *Proc. 30th Int. Colloq. on Automata, Languages and Programming*, pages 369–383, 2003.

[29] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. *Theory of Computing Systems*, 37:49–75, 2004.

[30] F. Kuhn, T. Moscibroda, and R. Wattenhofer. The price of being near-sighted. *Proc. of Symp. on Discrete Algorithms*, pages 980–989, 2006.

[31] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. *Combinatorica*, 13:441–454, 1993.

[32] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

[33] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. on Comput.*, 18:740–747, 1989.

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

[35] W. B. Powell, P. Jaillet, and A. Odoni. Stochastic and dynamic networks and routing. In *M. O. Ball and T. L. Magnanti and C. L. Monma and and G. L. Nemhauser (editors) Network Routing: volume 8 of Handbooks in Operations Research and Management Science*, pages 141–295, 1995.

[36] K. V. S. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13:235 – 257, 1992.

[37] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. of the 33rd ACM Symp. on Theory of Computing*, pages 183–192, 2001.

[38] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. of the 13th Symp. on Parallelism in Algorithms and Architectures*, pages 1–10, 2001.