Spanders
Distributed Spanning Expanders

Shlomi Dolev * Nir Tzachar †

October 30, 2007

Abstract

We consider self-stabilizing and self-organizing distributed construction of a spanner that forms an expander. The following results are presented.
• A randomized technique to reduce the number of edges of an arbitrary expander graph, while preserving expansion properties and incurring an additive stretch factor of $O(\log n)$.
• Given the randomized nature of our algorithms, a monitoring technique is presented for ensuring the desired results. The monitoring is based on the fact that expanders have a rapid mixing time and the possibility of examining the rapid mixing time by $O(n)$ short ($O(\log n)$ length) random walks even for non regular expanders.
• We then employ our results to construct a hierarchical sequence of *spanders*, each of them an expander spanning the previous graph. Such a sequence of spanders may be used to achieve different quality of service assurances in different applications.
• A snap-stabilizing reset algorithm for message passing systems is presented and utilized by the monitoring algorithm. We note that such a reset algorithm may be used as a snap-stabilizer in message passing systems.

1 Introduction

Distributed computing and communication networks research tries to define spanning subgraphs, such as spanning (BFS, DFS) tree algorithms, or a spanner graph that preserves a stretch factor between the shortest path in the original graph and the spanning graph. Dynamic and fault-tolerant algorithms, such as self-stabilizing spanning trees, were extensively investigated. However, distributed fault-tolerant algorithms for defining and maintaining a prominent class of graphs, *expander* graphs, were not thoroughly explored.

In this paper we consider *edge expanders*. A graph $G = (V, E)$ is an edge expander if there exists a constant c, such that for each set S of vertices (where $|S| < |V|/2$) it follows that $|E(S, \overline{S})|/|S| > c$

Expander graphs are perfect as a basis for communication networks; they are highly connected and symmetric in nature while being sparse and have a diameter in $O(\log n)$. Moreover, expander graphs are robust for dynamic changes; for example,[removal of a constant number, k, of nodes from the network may disconnect only additional $O(k)$ nodes.

*Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel, dolev@cs.bgu.ac.il. Partially supported by IBM, Israeli Ministry of Science, Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences.

†Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel, tzachar@cs.bgu.ac.il. Partially supported by Deutsche Telekom and the Lynn and William Frankel Center for Computer Sciences.

Self-stabilization is an important property of systems and in particular of distributed systems. A self-stabilizing algorithm is an algorithm which can be started in an arbitrary state and is guaranteed to converge to a legal behavior in a predetermined time. Such algorithms come in handy when unforeseen errors in the distributed system occur, after which the system may recover without any outside intervention.

The class of self-organizing systems forms a very important subclass of the class of self-stabilizing systems. Self-organizing systems converge in sub-linear time and react to dynamic changes even faster. The fault tolerant properties of expanders, the small diameter, and the communication cost (small number of edges) are very appealing to serve as self-organizing systems, as we demonstrate in this paper.

Our contribution. Self-stabilizing and self-organizing distributed algorithms for distributed construction of *spanders* are presented. Given an expander graph, G, a *spander*, S, is a subgraph of G such that S is an expander using a subset of the edges of G.

• First we consider the case in which the communication graph is a complete graph. In this simple case, a distributed random choice of a constant number of edges from each node results, with high probability, in an expander.

• Then we turn to consider the case of a communication graph that contains an expander, namely, has a certain expansion parameter. We show that when the spander edges are chosen using a binomial distribution, the obtained expansion is proportional to the expansion of the original graph, reducing the number of edges by the same factor.

• We introduce randomized methods for monitoring and verifying the diameter of the spander, the number of edges, and the cover time within $O(\log\ n)$ time (measured as the longest happened before chain), and communicating at most $O(n\ (\log\ n)^2)$ bits.

• We present a snap-stabilizing message-passing reset algorithm, which is employed by the monitoring algorithm. We further note that the reset algorithm may be used as a snap-stabilizer in message passing systems.

• To further reduce the number of edges while converging to a spander, we suggest using a hierarchical structure of spanders obtained by repeatedly applying the spander definition and monitoring, while verifying the total edge reduction factor. This construction results in spanders with different quality of service, which forms a tradeoff between the number of edges versus the confidence of expansion.

Related work. To the best of our knowledge, there is a limited number of works that address the problem of distributed expander construction. In [11], the authors propose to construct an expander graph by composing a sufficient number of Hamiltonian cycles. The proposed construction makes the following assumptions: the communication network is an overlay network (two nodes can directly communicate as long as their identifiers are known to each other), the algorithm starts from a predefined graph of at least three nodes and nodes wishing to join the graph must send a special message to a node that has already joined the graph. Unfortunately, the proposed algorithms cannot be started in an arbitrary state and therefor are not self-stabilizing.

A different approach for distributed construction of expanders is proposed in [14]. The authors suggest using uniform sampling to select, for each node, a set of expander-neighbors. The goal in [14] is to construct an "almost" regular graph, where each node maintains a list of expander neighbors of size between $d - \epsilon$ and $d + \epsilon$, where d is the desired degree of the almost regular graph. When a node v elects a node u to be its neighbor, v sends a JOIN message to u. However, u might already have more than $d + \epsilon$ neighbors, so u throws one neighbor, chosen uniformly, from u's neighbor list and inserts v instead. u then sends a LEAVE message to the thrown neighbor, w, to notify w to remove u from w's neighbor list. As a result, it is not straightforward to predict whether the given algorithm converges or oscillates among different

incomplete graphs.

The definition of self-organization we employ in this text was first presented in [9]. However, the system model which was used differs from ours; the authors assumed the system supports *hyperlinks*. That is, given a path between two nodes, u and v, a direct link between u and v may be established. Moreover, the communication overhead of such links is assumed to be negligible. In contrast, our model assumes a more conventional system, in which hyperlinks cannot be defined. The algorithms presented in this text achieve self-organization by employing the characteristics of the underlying expander graph.

Paper organization. In Section 2 we lay out the system settings. Spander extraction techniques are discussed in Section 3 and in Section 4 we present expansion monitoring protocols. In Section 5 we discuss the hierarchical construction of spander graphs using the results presented in the preceding sections.

2 System Settings

The *system* is defined by a *communication graph*, $G = (V, E)$, where V is a set of *nodes* $(v_1, v_2 \ldots, v_n)$ and E is a set of undirected *communication links*; if $(v, u) \in E$ then v and u can communicate by sending *messages* of bounded size to each other. Message sizes are restricted to $O(\log n)$ bits. We further assume that each communication link is a bounded capacity FIFO queue. Let lc denote the capacity of the links. When messages are sent over a full link, we assume that one of the messages (either already in the link, or the new one) is lost. We only require that messages which are sent infinitely often are received infinitely often. We present data–link algorithms to ensure that communication over such links is snap-stabilizing (see Section 4.3.1).

Nodes may join and leave the system at any time. We make no distinction between a node that leaves and a node that crashes, assuming that both can be detected by neighboring nodes in a timely fashion (e.g., checking the voltage in data link channels).

A configuration c of the system is a tuple $c = (S, L)$; S is a vector of states, $\langle s_1, s_2, \cdots s_n \rangle$, where the state s_i is a state of node v_i; L is a vector of *link states* $\langle l_{i,j}, \cdots \rangle$ for each $(i, j) \in E$. A *link* $l_{i,j}$ is modeled by a FIFO queue of messages that are waiting to be received by v_j and the content of the queue is the state of the link. Whenever v_i sends a message m to v_j, m is enqueued in $l_{i,j}$ (if the link is full, an arbitrary message in the queue will be dropped). Also, whenever v_j receives a message m from v_i, m is dequeued from $l_{i,j}$. A node changes its state according to its transition function (or program). A transition of node v_i from a state s_j to state s_k is called an *atomic step* (or simply a step) and is denoted by a. A step a consists of local computation and terminates with either a single send or a single receive operation.

The system is asynchronous, meaning that there is no correlation between the non-constant rate of steps taken by the nodes. We model our system using the interleaving model. An *execution* is a sequence of global configurations and *steps*, $\mathcal{E} = \{c_0, a_0, c_1, a_1, \ldots\}$, so that configuration c_i is reached from c_{i-1} by a step a_i of one node v_j. The states changed in c_i, due to a_i, are the one of v_j (which is changed according to the transition function of v_j) and possibly the state of a link attached to v_j. The content of a link state is changed when v_j sends or receives a message during a_i. An execution $\mathcal{E}$ is *fair* if every node executes a step infinitely often in $\mathcal{E}$. Within the scope of self-stabilization we consider executions that are started in an arbitrary initial configuration.

A *task* is defined by a set of executions called *legal executions* and denoted LE. A configuration c is a *safe configuration* for a system and a task LE if every fair execution that starts in c is in LE. A system is self-stabilizing for a task LE if every infinite execution reaches a safe configuration in relation to LE. We sometimes use the term "the algorithm stabilizes" to note that the algorithm has reached a safe configuration with regard to the legal execution of the

corresponding task.

To measure time we use the notion of *communication rounds*: a communication round (or just a round) is a sequence of atomic steps such that each node has taken at least one atomic step during this sequence. If this atomic step involves a send operation of a message m over link l, then we require that the atomic step which corresponds to a receive of a message from l, which was sent during this sequence of atomic steps, will also appear in the sequence. Such time measurements are appropriate when a protocol involves the entire system. When measuring the time complexity of a protocol which involves only a subset of nodes in the system, we use the notion of the *happened before* relation (see [12]). We then say that the time complexity of the protocol is the longest chain of happened before relation induced by the protocol.

A distributed algorithm is termed *self-organizing* ([9]) if it satisfies the following properties: (1) the algorithm is self-stabilizing, (2) convergence time to a safe configuration, $s(n)$, is in $o(n)$, and (3) after reaching a safe configuration, convergence time following a dynamic change, $d(n)$, is in $o(s(n))$.

A distributed algorithm is termed *snap-stabilizing* if the algorithm stabilizes following the first request by any node and before, or simultaneously with, a notification arriving to the requesting node at the completion of the request.

3 Expander Extraction

3.1 The Complete Graph

Consider the following generation of an expander: given a set of nodes, $|V| = n$, for each $v \in V$ choose d neighbors, independently at random. We wish to show that with overwhelming probability the resulting graph is a good *vertex* expander:

$$Prob[\min_{S \subset V, |S| \leq \frac{n}{2}} \frac{|\Gamma(S) \setminus S|}{|S|} < c] < o(1)$$

for some constant c. It is also known that a vertex expander is also a good edge expander.

To prove the above, we follow the proof in [13]. First fix a set $S \subset V$ of size s. Let $\Gamma(S) = \{u \in V | \exists v \in S, (u,v) \in E\}$ be the set of all neighbors of S. We wish to bound the probability that there exists a set T of size $t < c \cdot s$, such that $\Gamma(S) \setminus S \subseteq T$. If no such set exists, then it holds that $\frac{|\Gamma(S) \setminus S|}{|S|} > c$. Furthermore, if there exists such T of size smaller than $c \cdot |S| - 2$, then there exists such T' of size exactly $c \cdot |S| - 1$. As a result, the probability that S is "bad" is:

$$\begin{aligned} Prob\big[S \text{ is bad}\big] &= Prob[\exists T : \Gamma(S) \setminus S \subseteq T] \\ &\leq Prob\big[\exists T : |T| = c \cdot |S| - 1 \wedge \Gamma(S) \setminus S \subseteq T] \\ &\leq \sum_{|T| = c \cdot |S|} Prob[\Gamma(S) \setminus S \subseteq T] \\ &= \binom{n}{c \cdot s} \cdot \Big(\frac{s + c \cdot s}{n}\Big)^{d \cdot s} \end{aligned}$$

Using the above we show that the probability that the aforementioned construction has "bad" expansion is in $o(1)$.

$$
\begin{aligned}
Prob\Big[\min_{S\subset V,|S|\le\frac{n}{2}}\frac{|\Gamma(S)\setminus S|}{|S|}<c\Big] &= Prob\Big[\exists S\subset V,|S|\le\frac{n}{2}\text{ such that }\frac{|\Gamma(S)\setminus S|}{|S|}<c\Big]\\
&\le \sum_{s=1}^{n/2} Prob\Big[\exists S\subset V,|S|=s\text{ such that }\frac{|\Gamma(S)\setminus S|}{|S|}<c\Big]\\
&\le \sum_{s=1}^{n/2}\binom{n}{s}\binom{n}{c\cdot s}\Big(\frac{s+c\cdot s}{n}\Big)^{d\cdot s}\\
&\le \sum_{s=1}^{n/2}\Big(\frac{n\cdot e}{s}\Big)^{s}\Big(\frac{n\cdot e}{c\cdot s}\Big)^{c\cdot s}\Big(\frac{s+c\cdot s}{n}\Big)^{d\cdot s}\\
&= \sum_{s=1}^{n/2}\left[\Big(\frac{s}{n}\Big)^{d-c-1}\Big(\frac{e}{c}\Big)^{c}\cdot e\cdot(1+c)^{d}\right]^{s}\\
&\le \sum_{s=1}^{\log n}\left[\Big(\frac{\log n}{n}\Big)^{d-c-1}\Big(\frac{e}{c}\Big)^{c}\cdot e\cdot(1+c)^{d}\right]^{s}+\\
&\quad \sum_{s=\log n}^{n/2}\left[\Big(\frac{1}{2}\Big)^{d-c-1}\Big(\frac{e}{c}\Big)^{c}\cdot e\cdot(1+c)^{d}\right]^{s}\\
&\le \sum_{s=1}^{\log n}q^{s}+\sum_{s=\log n}^{n/2}r^{s}\quad\text{for } r,q<<1\\
&\le \frac{q}{1-q}+\frac{r}{1-r}=o(1)
\end{aligned}
$$

If the initial graph is complete, a simple random choice of edges yields a good expander with high probability. Moreover, the average degree of such an expander is constant and the maximal degree is in $O(\log n/\log\log n)$ with very high probability.

To realize such a construction in a distributed manner, the need for a monitoring tool which validates the expansion of the spander arises. In the next section we will discuss such a self-organizing monitoring algorithm.

3.2 An Arbitrary Expander

In some cases, the initial topology of the graph is not known. The only input may be that the initial graph is a good expander. Yet, a spander construction needs to be carried out under such constraints; one may wish to use the technique presented in the previous section to define a spander over *any* given expander graph. However, when choosing a constant number of neighbors at each node the same technique fails. As a counter example consider the following graph: let $G=(V,E)$ be a regular (degree) expander graph. Now, let V_1 be a set of half the nodes in V and $V_2=V\setminus V_1$. Consider the following graph, $G^*=(V,E^*)$, where $E^*=E\cup\{(v,u):v,u\in V_i(i=1,2)\}$. Since adding edges can only increase the expansion of a graph, G^* is a good expander (at least as good as G). Suppose we then proceed to generate a constant degree expander from G^* by choosing for each node a constant number of neighbors, independently at random. It is easy to see that, with a positive probability, we will get a disconnected graph, and with an overwhelming probability the resulting graph will not be a good expander.

Taking the previous paragraph into account, we still wish to reduce the number of edges of an arbitrary expander, without sacrificing the expansion property. When considering a graph $G=(V,E)$ such that each cut in the graph contains enough edges, one may notice that if each edge is selected using a constant probability, then with very high probability all the cuts will remain large – thus keeping the expansion of the graph.

Lemma 3.1. *Let $G = (V, E)$ be a graph with edge expansion $c \cdot \log n$, where $n = |V|$. The graph $G^* = (V, E^*)$, such that $P[(u, v) \in E^*] = p$ has edge expansion of $d \cdot p \cdot \log n$ with overwhelming probability for an appropriate p and d.*

Proof. First consider a set $S \subset V$ such that $|S| = s \leq \frac{n}{2}$. Since G is an edge expander, we know that $|E(S, \overline{S})| > s \cdot c \cdot \log n$. Let us now calculate the probability that such a cut is "small" in G^*, using Chenroff's bound for the cumulative distribution function of the Binomial distribution:

$$\begin{aligned} P\left[|E_{G^*}(S, \overline{S})| < d \cdot s \cdot p \cdot \log n\right] &\leq exp(-\frac{(c \cdot s \cdot p \cdot \log n - d \cdot s \cdot p \cdot \log n)^2}{2 \cdot c \cdot s \cdot p \cdot \log n}) \\ &= exp(p \cdot \log n(d \cdot s - \frac{d^2 \cdot s}{2} - \frac{c \cdot s}{2}) \\ &= n^{p \cdot s\left(d - \frac{d^2}{2} - \frac{c}{2}\right)} \end{aligned}$$

Now, using the union bound, and denoting a "bad" cut as a cut S such that $|E_{G^*}(S, \overline{S})| < d \cdot s \cdot p \cdot \log n$, the probability that no such "bad" cut exists is :

$$\begin{aligned} P\left[\text{!}\exists\text{a "bad" set S}\right] &\leq \sum_{1 \leq s \leq n/2} \binom{n}{s} n^{p \cdot s\left(d - \frac{d^2}{2} - \frac{c}{2}\right)} \\ &\leq \sum_{1 \leq s \leq n/2} n^{s \cdot (1 + p(d - \frac{d^2}{2} - \frac{c}{2}))} \\ &= O\left(n^{1 + p(d - \frac{d^2}{2} - \frac{c}{2})}\right) \end{aligned}$$

It is easy to see that by choosing appropriate values for c, d, p (for example, $p = 1/2, d = 1, c \geq 7$) we get that the probability of failure is less than $O(n^{-0.5})$ □

Realizing such a construction in a distributed manner is very simple; in order for each edge to be chosen with probability p, each node must choose each of its adjacent edges with probability $1 - \sqrt{1 - p}$ (hence, the probability the edge is not chosen is: $\sqrt{1 - p} \cdot \sqrt{1 - p} = 1 - p$).

4 Expansion Monitoring

Assuming that, with high probability, we can construct an expander, sometimes it might be necessary to evaluate our construction; we may wish to check that the resulting graph is a good enough expander (or even whether it forms a connected component) and whether enough edges were removed.

If message sizes, memory, and processing power of a single node are not restricted, it is easy to collect the entire topology of the graph at each node and check if the resulting graph is an expander. When restricting message sizes, memory requirements at each node, and convergence time to $O(\log n)$, such solutions are no longer feasible.

In the remainder of this section we tackle the task of monitoring the result of our construction when message sizes are limited to $O(\log n)$.

4.1 Monitoring by Random Sampling

Our first approach to monitoring is presented for the sake of building intuition, as the time it takes for detection is exponential. The monitoring is done by sampling sets of nodes from the

graph and calculating the expansion for each such set. If a set of nodes is found to be of small expansion, a reset procedure will follow.

Sets are sampled in the following manner: a node repeatedly starts a randomized propagation of information with feedback (PIF) flooding of the graph, which defines the set of nodes. Each of these sampled nodes will report back the number of neighboring nodes which were not selected. This information will then propagate back to the initiating node which will then calculate the expansion of the selected set.

The main observationis given in Lemma 4.1.

Lemma 4.1. *Let $G = (V, E)$ be a graph. If there exists a set of nodes, $S \subset V, |S| \leq \frac{|V|}{2}$, such that $\frac{|E(S,\overline{S})|}{|S|} \leq c$ for some c, then there exists a subset $S' \subseteq S$, $|S| \leq \frac{n}{2}$, $\frac{|E(S',\overline{S'})|}{|S'|} \leq c$ and S' induces a connected sub graph of G.*

Proof. The set S can be decomposed in the following fashion: $S = \bigcup_i S_i$, $\forall i, j : E(S_i, S_j) = \emptyset$ and $\forall i : S_i$ induces a connected sub-graph, where each S_i is maximal (vertex-wise). Such a decomposition is easily obtained using greedy selection. Assume to the contrary that none of these subsets satisfies Lemma 4.1. It follows that for each subset S_i, $\frac{|E(S_i,\overline{S_i})|}{|S_i|} > c$. Now:

$$\begin{aligned} \frac{|E(S,\overline{S})|}{|S|} = \frac{\sum_i |E(S_i,\overline{S_i})|}{\sum_i |S_i|} &\leq c \\ c < \frac{\sum_i c \cdot |S_i|}{\sum_i |S_i|} &\leq c \end{aligned}$$

it follows that $c < c$, which is a contradiction. Hence, one of the sets S_i must satisfy Lemma 4.1. □

More details on the way to implement analogous self-stabilizing monitoring will be presented in the next section, as part of the description of the efficient monitoring scheme.

4.2 Mixing Rate Based Monitoring

Calculating the expansion of a graph is NP hard. In order to calculate an approximation of the expansion, one needs to collect the entire topology of the network and perform a costly computation. A different approach to estimate the expansion of a graph using a distributed algorithm is based on the rapid mixing rate of expander graphs.

Expander graphs are rapidly mixing graphs ($O(\log n)$ mixing rate), and it follows that the cover time of such graphs is also short ($O(n \log n)$). We can employ this fact in the following way: assume a node, v, wishes to check if the graph is rapidly mixing. v will start a random walk of length $O(n \log n)$ and attach a random color to this walk, chosen from a large enough domain. Further, v associates three counters of $O(\log n)$ bits each to the random walk; one to limit the number of hops taken by the token, one to count the nodes discovered by the walk, and one to count the edges of the graph. Each time the walk visits a node (Figure 1) for the first time (line 1), the node increments the node counter by 1 and the edges counter by the number of its neighbors. Afterwards, the token is transfered to a random neighbor (line 8). When the walk terminates (line 6), the counters are examined, either by the last node or by routing a message with the counters back to v. In case the walk covered less than n nodes, which implies that the graph is not rapidly mixing, or there are too many edges in the graph (relative to the original graph, which implies that the construction is not as productive as desired) a reset is initiated.

When n is unknown, the protocol needs to be slightly adjusted; since counting the nodes cannot be used to determine coverage, each node remembers the last color of a token traversing

```
N = list of neighbors
L = an upper limit on the length of the walk
last_seen = ∅

Receive(color, node_counter, edge_counter, length):
1   if color ∉ last_seen then
2   |   last_seen ← last_seen ∪ {color}
3   |   node_counter ← node_counter + 1
4   |   edge_counter ← edge_counter + |N|
5   fi
6   if length > L then
7   |   Report counters to initiating node
8   else
9   |   choose u ∈ N uniformly at random
10  |   Send(u, color, counter, length + 1)
11  fi
```

Figure 1: Rules for node u.

it. After the random walk is terminated at node v, v initiates a flooding of the network to check if all nodes were colored by the same color of the token. If the flooding detects a node which has not been colored, a reset procedure will ensue.

To speed up the detection rate we need to use a shorter random walk. To achieve this, v may send out $O(n)$ random walks in parallel instead of one, each of length $O(\log n)$, and attach the same color to all of these walks. This elegant technique was suggested in [2] and analyzed for regular expanders. Here we extend the analogy for the non-regular case.

Each of the $O(n)$ random walks will hold counters in a fashion similar to the single random walk solution (see Figure 1). These random walks will cover the graph with high probability (see Lemmas 4.2 and 4.3). To count the number of visited nodes, the final counters must be routed back to v (we will elaborate more on that in Section 4.3.4). If the total number of nodes visited is less than n then, with very high probability, the graph is not a good expander. Analogously to the discussion above, coloring and checking nodes can be used for the case in which n is not known.

In case the graph is not a good expander, we argue that initiating the random walks from a random edge results in failure with probability larger than half (see Lemma 4.4).

The running time of this monitoring algorithm is $O(\log n)$ and the message complexity is $O(n \log n)$ bits.

Lemma 4.2. *Let G be a connected, non bipartite graph, such that the second largest eigenvalue of the transition matrix of the random walk on G is λ and the expansion of the graph is at least $c \cdot \log n$.*

- *Let $s = \frac{5 \cdot \log n + 2 \log 1/c}{1-\lambda}$. The probability of a random walk of length $2 \cdot s$, starting at any vertex $u \in V$, to visit a given vertex $v \in V$, is at least $\frac{s}{\frac{2n^2}{c \log n} + 2s + \frac{2n^2 \lambda}{c \log n (1-\lambda)}}$.*
- *Assuming $d_{max}/d_{min} \leq \log^m n$ for some constant m and $s = \frac{\log 2cn + 2m \log \log n}{\log 1/\lambda}$, the probability to visit a given vertex $v \in V$ is at least $\frac{s}{nc(\log^m n)\frac{1+\lambda}{1-\lambda} + 2s}$.*

Proof. We will follow the proofs presented in [2]. Fix u as the node from which the random walks start and fix a node, v. Let Y_i, $s \leq i \leq 2s$, be indicator random variables such that $Y_i = 1$ if the walk visited v at step i. Let $Y = \sum_{i=s}^{2s} Y_i$ be the sum of these random variables. We need to show that $P[Y > 0] > \frac{s}{\frac{2n^2}{c \log n} + 2s + \frac{2n^2 \lambda}{c \log n (1-\lambda)}}$.

Using the Cauchy-Schwarz inequality, we can see that

$$(\sum_{j>0} P[Y=j])(\sum_{j>0} j^2 P[Y=j]) \geq (\sum_{j>0} P[Y=j])^2$$

$$P[Y>0] = (\sum_{j>0} P[Y=j]) \geq \frac{(\sum_{j>0} P[Y=j])^2}{(\sum_{j>0} j^2 P[Y=j])} = \frac{(E(Y))^2}{E(Y^2)}$$

From now on, we will focus on estimating both $E(Y)$ and $E(Y^2)$. From linearity of expectation, $E(Y) = \sum_{i=s}^{2s} E(Y_i)$ and for each i, $E(Y_i)$ equals the probability that the walk which started at u visits v precisely at step i. This probability is $A^i z(v)$, where A is the transition matrix of the random walk and z is the indicator vector with 1 in the coordinate of u and 0 elsewhere.

We now use the results presented in [6, 15] which bound the mixing rate of a random walk on a graph. Let u be the starting node, denote by $P^t(u,\cdot)$ the distribution of the walk at time t and let π be the stationary distribution over A. Define:

$$\Delta_u(t) = \max_{S \subseteq V} |P^t(u,S) - \pi(S)| = \frac{1}{2} \sum_{v \in V} |P^t(u,v) - \pi(v)|$$

and

$$\tau_v(\epsilon) = \min\{t : \forall t' \geq t, \Delta_v(t') \leq \epsilon\}$$

Now, the bound on the mixing time is given in the following inequality:

$$\tau_v(\epsilon) \leq (1-\lambda)^{-1}(\log \pi(v)^{-1} + \log \epsilon^{-1})$$

Since G has expansion of at least $c \log n$, for each $v \in V$ we get that $\pi(v) \geq \frac{c \log n}{n^2}$. It easily follows that $\tau_u(\frac{c \log n}{4n^2}) \leq s$. As a result, we get that after a walk of length i steps ($s \leq i \leq 2s$) $|P^i(u,v) - \pi(v)| \leq \frac{c \log n}{2n^2}$. It immediately follows that each entry in $A^i z$ is at least $\frac{c \log n}{2n^2}$. Furthermore, we can conclude that

$$E(Y) \geq \frac{s \cdot c \cdot \log n}{2n^2}$$

We now wish to bound $E(Y^2)$. From the definition of Y we get:

$$\begin{aligned} E(Y^2) = E((\sum_{i=s+1}^{2s} Y_i)^2) &= \sum_{i,j} E(Y_i Y_j) = \sum_i E(Y_i^2) + \sum_{i \neq j} E(Y_i Y_j) \\ &= \sum_i E(Y_i) + 2 \sum_{s<i<j<2s} E(Y_i Y_j) \end{aligned}$$

Now, $E(Y_i Y_j)$ is exactly the probability that the walk visits v at step i and then at step j ($i < j$), which is the probability of visiting v at step i times the probability of returning to v after a walk of length $j - i$. According to [8], we know that:

$$4\|P^t(x,\cdot) - \pi\|^2 \leq \frac{1-\pi(x)}{\pi(x)} \lambda^{2t}$$

Given that $\pi(x) \geq \frac{c \log n}{n^2}$, it follows that the probability of returning to v after k steps is at most $\frac{c \log n}{n^2} + \frac{\lambda^k}{2}$.

It then follows that

$$\begin{aligned} E(Y^2) &= E(Y) + 2\sum_{s<i<j<2s} E(Y_i)P(Y_j|Y_i) \\ &\leq E(Y) + 2\sum_{s<i<j\leq 2s} E(Y_i)(\frac{sc\log n}{n^2} + \sum_{k>0}\frac{\lambda^k}{2}) \\ &= E(Y)(1 + \frac{2sc\log n}{n^2} + \frac{\lambda}{1-\lambda}) \end{aligned}$$

Combining all of the above, we get that

$$P[Y>0] \geq \frac{E(Y)^2}{E(Y^2)} = \frac{E(Y)}{1 + \frac{2sc\log n}{n^2} + \frac{\lambda}{1-\lambda}} = \frac{s}{\frac{2n^2}{c\log n} + 2s + \frac{2n^2\lambda}{c\log n(1-\lambda)}}$$

In some cases we may have a bound on the ratio between the maximal and minimal degrees in the graph. In such cases, we may achieve tighter results; assume that $d_{max}/d_{min} \leq \log^m n$ for some constant m (this implies the $|E| \in \tilde{O}(n)$). Note that it is possible to check this ratio assumption locally, verifying that the d_{min} is big enough when d_{max} is known. In case any node finds a problem, a reset procedure can be used to reconstruct the graph.

Using the above bound, we get that $\pi_{min} \geq \frac{1}{cn\log^m n}$. We will further employ the fact that for each two vertices, u, v, $|P^t(u,v) - \pi(v)| \leq \lambda^k \frac{d_{max}}{d_{min}}$ (see [10], Chapter 10) and that assuming that all the eigenvalues of P are positive we have that $|P^t(u,v) - \pi(v)| \geq \lambda^k$; this can easily be achieved by transforming the random walk in the following fashion: for each node, v, stay in the node with probability $1/2$ and move to a neighbor with probability $\frac{1}{2d(v)}$. It then follows that all eigenvalues are positive and λ of the modified transition matrix is exactly $\lambda/2$ of the original matrix.

Using the information above, it then follows that a random walk of length $2s$, where $s = \frac{\log 2cn + 2m\log\log n}{\log 1/\lambda}$, starting at node u, will visit an arbitrary node v with a probability of at least $\frac{1}{2cn\log^m n}$. It follows that

$$E(Y) \geq \frac{s}{2cn\log^m n}$$

Now,

$$\begin{aligned} E(Y^2) &= E(Y) + 2\sum_{s<i<j<2s} E(Y_i)P(Y_j|Y_i) \\ &\leq E(Y) + 2\sum_{s<i<j\leq 2s} E(Y_i)(\frac{s}{cn\log^m n} + \sum_{k>0}\lambda^k) \\ &= E(Y)(1 + \frac{2s}{cn\log^m n} + \frac{2\lambda}{1-\lambda}) \end{aligned}$$

Combining all of the above, we get that

$$P[Y>0] \geq \frac{E(Y)^2}{E(Y^2)} = \frac{s}{nc(\log^m n)\frac{1+\lambda}{1-\lambda} + 2s}$$

□

Lemma 4.3. *Let G be a graph with expansion at least $c\cdot\log n$ and $d_{max}/d_{min} \leq \log^m n$ for some constant m. $k \in O(n\log^m n)$ random walks started from the same node in the graph, v, each of length $2s$, cover the entire graph with overwhelming probability (at least $1 - \frac{1}{n}$).*

Proof. For each node u, the probability that it is not visited by a given random walk is less than $1 - \frac{s}{nc(\log^m n)\frac{1+\lambda}{1-\lambda}+2s}$. The probability that none of the k random walks visit u is less than $(1 - \frac{s}{nc(\log^m n)\frac{1+\lambda}{1-\lambda}+2s})^k$. When $k \approx O(n \log^m n)$, we get that this probability is smaller than $\frac{1}{n^2}$. Using the union bound, we get that the probability that all nodes are covered is larger than $1 - \frac{1}{n}$. □

Following lemma 4.3, we now know that if a graph is a good expander, the short random walks we use will cover the graph with high probability. However, we also wish to investigate the case in which a graph is not a good expander. The following lemma illustrates that when a graph has less than constant expansion, the short random walks used will not cover the graph with a probability of at least half.

Lemma 4.4. *Let $G = (V, E)$ be a graph, such that that there exists $S \subset V, |S| \leq \frac{|V|}{2} = \frac{n}{2}$ for which $|E(S, \overline{S})| < \frac{|S|}{8 \log^2 n}$. $n \log n$ random walks started from a uniformly chosen edge, each of length $O(\log n)$, will not cover the entire graph with a probability of at least $\frac{1}{2}$.*

Proof. Let $(u, v) \in E$ be a directed edge, chosen uniformly at random. For each edge $e \in E$ define X_e as a random variable counting the number of times one of the random walks traverses e. Start the $n \log n$ random walks from v. Now, since (u, v) was chosen uniformly, which is the stationary distribution of the edges of the graph, for each directed edge $e \in E$ we have $E(X_e) = \frac{n \log n}{|E|}$. Let $Y = \sum_{e \in E(S,\overline{S})} X_e$ (where e is a directed edge). From linearity of expectation we get that

$$\begin{aligned} E(Y) &= \frac{2n \log n \cdot |E(S, \overline{S})|}{|E|} \\ &\leq \frac{2n \log n |S|}{8n \log^2 n} = \frac{|S|}{4 \log n} \end{aligned}$$

From Markov's inequality, we get that with probability at least $\frac{1}{2}$, the cut is not crossed more than $\frac{|S|}{2 \log n}$ times (in both directions). Assuming $v \notin S$, and since each walk is of length $\log n$, we cover at most $\frac{|S|}{2}$ nodes within S. If $v \in S$, we get that we cover even less of $\overline{S}$, since $|S| \leq |\overline{S}|$. It follows that with a probability of at least $\frac{1}{2}$, we do not cover the entire graph. □

4.3 Self-Stabilizing Distributed Monitoring

Given the monitoring technique presented in the previous section, we wish to discuss a self-stabilizing distributed monitoring algorithm to monitor the mixing rate of a graph. The algorithm will be based on repeatedly selecting a random edge from the graph and starting random walks from this edge, according to the results in the previous section. Whenever a problem is observed in the graph, a snap-stabilizing reset will be utilized to reset the monitoring algorithm and to rebuild the current graph (if it is not a good spander).

4.3.1 Snap-Stabilizing Data Link

Throughout the text we assume the use of a snap-stabilizing data-link layer. We consider several algorithms which can be used to realize a snap-stabilizing data-link layer.

• **Spontaneous receiver**. When the receiver may send duplicated answers for a single frame sent (or the duplication may be caused by the underlying physical layer), we suggest the following algorithm (see [9]): when the sender, s, wishes to send a frame, f, to the receiver, r, s will send f to r repeatedly and attach a sequence number to f. At first, s will attach the number 1 to f, and repeatedly send f to r with sequence number 1. Once s receives an acknowledgment from r upon the receipt of f with sequence number 1, s will repeatedly to send f with sequence number 2. s will keep incrementing the sequence number of f each time s receives an acknowledgment on the current sequence number, until s reaches $2 \cdot lc + 1$ times (where lc is the bound on the link capacity). At this point f is assured to be delivered at r.

When r receives a frame f with a sequence number $2 \cdot lc + 1$, following a frame with a different sequence number, r will deliver the frame upwards in r's network stack.

• **Non-spontaneous receiver**. When frames are not duplicated (either by r or by the physical layer), we suggest the following algorithm: s will first repeatedly send f to r, but will mark f with 0. s will then count the number of acknowledgments it receives from r. When s has received $2 \cdot lc + 1$ acknowledgments, s will then mark f with 1 and repeat the process. After s receives $2 \cdot lc + 1$ additional acknowledgments for f, f is assured to be delivered at r.

When r receives a frame f with a mark 0 that is immediately followed by a frame with a mark 1, r will deliver the frame upwards in r's network protocol stack.

The algorithm for the non-spontaneous receiver is more efficient both communication-wise and time-wise.

4.3.2 Snap-Stabilizing Message Passing Reset

We present a reset procedure for message passing networks. The reset is executed on a graph G, which in our case is of diameter $d \in O(\log n)$. We assume the use of a snap-stabilizing data-link protocol for passing messages between nodes. A similar technique appears in [3, 5, 7], although here we present and prove *snap-stabilization and termination*, while fitting anonymous networks. The reset procedure for a single node, u, appears in Figure 2.

The reset procedure is based on bounded counters of $3d$ at each node. Each node u maintains a reset counter. To initiate a reset, u just sets u's counter to zero (line 1). The technique used by the reset procedure is that a node, u, will only increment u's counter after u is certain that all of u's neighbors have counters which are greater than or equal to u's counter. This is achieved by first saving the counter values received from the neighbors (line 3) and assigning u's counter with the minimal value among the counters of all of u's neighbors, plus one (line 14).

u must also repeatedly broadcast u's counter value to u's neighbors. The broadcast is based on the snap stabilizing data-link algorithm presented above. The SnapSend procedure used in line 17 ensures that the counter value C is sent to the neighbor v using a snap-stabilizing data-link algorithm, thus ensuring the delivery of the new value. The receiving node of the SnapSend procedure piggy backs its own counter value while sending acknowledgments, which is returned from the SnapSend procedure on the initiator side. This ensures that upon the termination of a single SnapSend procedure both ends hold the other side's counter values that existed during the SnapSend execution.

The $flag$ variable is used to ensure that u will not reassign u's counter before updating all of u's neighbors with u's current value. This property is vital to our proof, and establishes a happened before relation between counter updates across the graph.

The proof is based on the following observation: when a node, v, assigns 0 to its counter (starting a reset), this value will propagate in the graph, causing other nodes to adopt a counter value not greater than their distance to v. First, v's immediate neighbors will set their counters to at most 1. Afterwards, v's neighbors' neighbors will set their counter to (at most) 2, and so

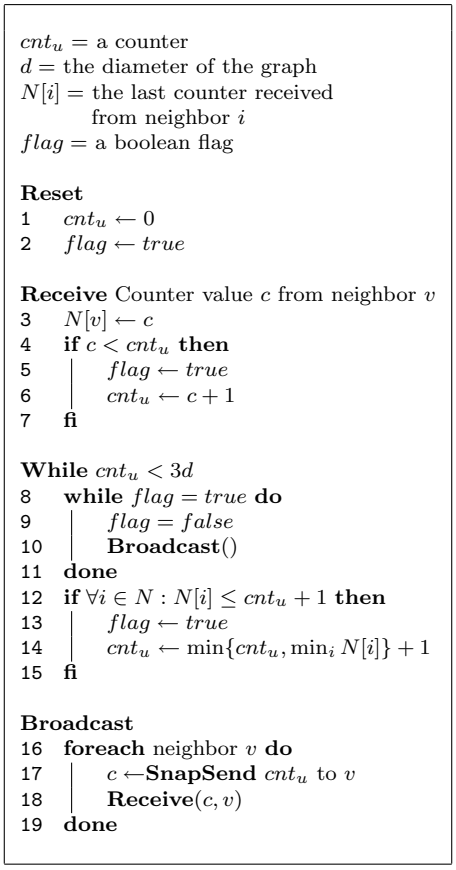

Figure 2: Snap-stabilizing reset procedure for u.

on. We then show that when v reaches $2d$, each node can not exceed $3d$.

Definition 4.1. Given an execution $\mathcal{E}$, a *happened before path* during $\mathcal{E}$ between a node u and a node v is a path in the graph induced by a happened before relation between u and v. Assume the happened before relation is $u \rightsquigarrow p_1 \rightsquigarrow p_2 \rightsquigarrow \ldots \rightsquigarrow p_i \rightsquigarrow v$, where u sent a message to p_1, p_1 sent a message to p_2 after receiving the message from u, etc. The *happened before path* is defined to be $u, p_1, p_2, \ldots, p_i, v$.

Lemma 4.5. *Let c_v be a configuration in which a node, v, has started the reset (line 1). For every node u, and in every fair execution, there exists a configuration c' such that u receives a counter value (line 2) in c' and a happened before relation between the atomic step of v in c_v, which started the reset, and the atomic step of u in c' exists. Moreover, there exists such a happened before relation and a configuration c_u such that the happened before path induced by the happened before relation forms a shortest path from v to u.*

Proof. The proof is by induction over the distance between u and v. When $d(u, v) = 0$, the claim obviously holds as $u = v$. Assume that the claim holds for all $p \in V$ such that $d(p, v) < d(u, v)$. Since the graph is connected, there exists a node p such that $d(u, v) > d(p, v)$ and $(u, p) \in E$. According to the induction assumption, there exists a configuration c_p satisfying the Lemma. Let c_u be the first configuration following c_p in which u receives a counter value from p which was sent in a configuration following c_p. Clearly, c_u satisfies the conditions of the Lemma. □

Lemma 4.6. *Let c_v be a configuration in which a node, v, has started the reset (line 1). For a node u, let c_u be the earliest configuration satisfying Lemma 4.5. The counter value of u in c_u, $cnt_u(c_u)$, may not exceed $d(v, u)$, the distance between v and u.*

Proof. By induction on $d(v,u)$. When $d(u,v)=0$, the assumption clearly follows since $v=u$ and $c_u=c_v$. Assume that the claim holds for all values smaller than i. Let u be a node such that $d(v,u)=i$ and mark c_u as the first configuration which satisfies Lemma 4.5. Let w be u's neighbor such that $d(w,v)=i-1$ and w is u's predecessor in the happened before relation which defined c_u. Let c_w be the configuration which satisfies Lemma 4.5 with regards to w. It follows, from the inductive assumption, that $cnt_w(c_w)\leq i-1$. Since w repeatedly broadcasts its counter value, and will only increase w's counter value after the broadcast phase is completed at least once (this is due to the $flag$ variable), there exists a configuration c', between c_w and c_u, in which u received a value $i-1$ or less from w for the first time. It follows that $c'=c_u$, and since u adopts the counter value (plus one) received if the value is smaller than its own, we conclude that u's counter value may not exceed $(i-1)+1=i$ in c_u. □

Lemma 4.7. *Let c_v be a configuration in which a node, v, has started the reset (line 1). For a node, u, let c_u be the earliest configuration satisfying Lemma 4.5, and let w be a node on the happened before path from v to u. For every fair execution, and in every configuration c' between c_w and c_u, $cnt_w(c')\leq d(u,v)+d(u,w)$.*

Proof. Let u be a node and c_u be a configuration as defined in Lemma 4.5. Let w be a node on the happened before path from v to u. The proof is by induction on $d(u,w)$. For $d(u,w)=0$, it follows that $u=w$ and the claim clearly holds from Lemma 4.6. Assume the claim is correct for all values smaller than i.

Let x be a node on the happened before path between u and w, such that $d(u,x)+1=d(u,w)$. First note that according to Lemma 4.6, we know that $cnt_w(c_w)\leq d(w,v)$. From the induction assumption, it follows that in every configuration c' between c_x and c_u $cnt_x(c')\leq d(u,v)+d(u,x)$. Furthermore, in each such c' it clearly follows that $cnt_w(c')\leq cnt_x(c')+1$, since w always adopts the lowest counter value in its neighborhood plus one. As a result, in every configuration c' between c_x and c_u, we get that $cnt_w(c')\leq d(u,v)+d(u,x)+1=d(u,v)+d(u,w)$.

According to Lemma 4.6, $cnt_w(c_w)\leq d(w,v)$. Note now that w will only increment its counter in consecutive steps, and will make sure w communicates with all of w's neighbors afterwards; this is due to the $flag$ used in the algorithm. This in turn ensures that between c_w and c_x w may increment w's counter by no more than 1. Hence, in every configuration c' between c_w and c_x we get that $cnt_w(c')\leq d(w,v)+1$.

To conclude, we get that in every configuration c' between c_w and c_u, $cnt_w(c')\leq d(u,v)+d(u,w)$. □

Lemma 4.8. *Let c_v be a configuration in which a node, v, has started the reset (line 1) and u be a node such that for each node w c_w preceded c_u. It follows that in every configuration c' between c_w and c_u, $cnt_w(c')\leq 3d$.*

Proof. Assume, towards contradiction, that there exists a node w and a configuration c' such that c' is between c_w and c_u and $cnt_w(c')>3d$. Consider the shortest path between v and w, $(v,v_1,v_2,\ldots,v_k,w)$ induced by Lemma 4.5. It follows that $cnt_{v_k}(c')>3d-1$, since w and v_k have exchanged counter values. In a similar way, we can show that $cnt_v(c')>3d-i-1$. Now, since $i\leq d-1$, it follows that $cnt_v(c')>2d$. However, according to Lemma 4.7, we know that in each configuration between c_v and c_u, the counter value of v may not exceed $2d$, which yields the contradiction. □

Lemma 4.9. *Let c_v be a configuration in which a node, v, started a reset. In every fair execution there exists a configuration c', which follows c_v after at most d communication rounds, such that for each node u it holds that $cnt_u(c')<3d$.*

Proof. The proof follows from the fact that after d communication rounds there is a happened before relation between v and any nodes u on a shortest path between v and u. Moreover, from Lemma 4.8 it follows that all nodes have counter values less than $3d$. □

Lemma 4.10. *Liveness: let c be a configuration in which at least one node has a counter value less than $3d$. Let v be a node such that $cnt_v(c) < 3d$ and for every node u $cnt_u(c) \geq cnt_v(c)$. Let c' be a configuration following c after d communication rounds, during which no node has started a reset. For every node $u \in V$, $cnt_u(c') \geq cnt_v(c) + 1$.*

Proof. The proof is by the fact that every node v with a minimal counter will not set v's $flag$ to $true$, and hence will finish the broadcast phase and advance v's counter value by 1. □

To ensure the termination of the reset algorithm, we enable nodes to start a reset only when their counter value is larger than $3d$. Such a restriction gives rise to the following: assume a node, v, has started the reset. Following Lemma 4.9, we will reach a configuration in which all counter values are less than $3d$. Combining that with Lemma 4.10, and assuming that once a successful reset is performed no new reset will be started for a long time, such that nodes reaching $3d$ will not start a reset for a long enough period, the reset algorithm will terminate. Namely, all nodes' counters will reach $3d$.

Theorem 4.1. *The reset protocol in Figure 2 is snap-stabilizing.*

The snap-stabilization paradigm requires that once one node initiates the (reset) algorithm, the algorithm will terminate successfully; namely, the initiator receives a notification on the termination and no more messages are sent. The termination property of the snap-stabilizing reset is a rare property in the scope of self-stabilizing algorithms (see [4]). The reset algorithm is snap-stabilizing since once a node, v, starts the reset algorithm, each node in the system will go through a reset, namely lower its counter value below $3d$, before v receives an indication on the reset termination.

In the context of our monitoring algorithm presented in the following section, a leader must be elected following the reset, and a BFS tree rooted in the leader needs to be defined. We suggest the following algorithm: once the counter of a node, v, reaches $3d$, v will start broadcasting to v's neighbors v's candidate for the BFS leader and the distance to the candidate, starting with v: $(v, 0)$. v will repeatedly collect the BFS leaders of v's neighbors. If there exists a BFS leader with a lower identifier than v's current BFS leader or one of v's neighbors is closer to the leader than v, v will adopt this BFS leader, mark the node from which v received this BFS leader as v's predecessor in the BFS tree, and add one to the distance to the BFS leader. Once any BFS leader has a distance larger than d, v can safely discard this BFS leader.

It is easy to see that this leader election algorithm, when started after a Reset, will elect a leader, define a BFS tree rooted in the leader, and establish correct distances. After further $2d$ communication rounds (due to the liveness property, see Lemma 4.10) all nodes will agree on one leader, the node with the lowest id in the system. If we let nodes continue the counting of the reset algorithm, starting to execute the leader election algorithm at $3d$ and up to $5d$, the leader can start a new monitoring session once its counter reaches $5d$. In the sequel, once the counter reaches $5d$, the tree structure remains fixed in the sense that parent pointers are not changed.

4.3.3 Monitoring Algorithm for a Single Node

We next present a self-stabilizing expansion monitoring algorithm. In fact, given the self-stabilizing reset presented earlier, a simpler non-stabilizing version can be used, as long as there

is a local predicate for checking the consistency of the monitoring that triggers the reset. Still, the self-stabilizing monitoring may be of independent interest. Next, we consider an instance of the algorithm which is started from one node, which we denote ml (for monitoring leader).

When designing the self-stabilizing version of the monitoring algorithm, we may notice two problems which we need to solve; one, we need to ensure that tokens can be routed back to ml once they have traveled the required length. Second, if ml assumes that a monitoring session has started, and none of ml's tokens exist in the graph, we need to prevent ml from waiting forever. To overcome both of these obstacles, we use the same repeated token sending mechanism.

```
N = list of neighbors
L = an upper limit on the length of a walk
tokens[] = an array, holding the last tokens received
answers[] = an array, holding the last hop of a token (if exists)

Start Walks
1   choose a random color c
2   for i ∈ tokens do
3   |   choose a random neighbor p
4   |   tokens[i] ← (c, 0, 0, 0, p)
5   done

Receive(color, index, origin, hop_counter, node_counter, edge_counter) from v
6   choose a random neighbor p
7   if tokens[index].color ≠ color
8   |    tokens[index] ← (color, hop_counter, node_counter + 1,
9   |                     edge_counter + |N|, p)
10  else if tokens[index].hop_counter < hop_counter then
11  |   tokens[index] ← (color, hop_counter, node_counter,
12  |                     edge_counter, p)
13  fi
14  if answers[index].color = color then
15  |   Send (i, answers[index]) to v
16  fi

Repeatedly
17  foreach i ∈ tokens do
18  |   (color, hop_counter, node_counter, edge_counter, p) ← tokens[i]
19  |   if hop_counter < L then
20  |   |   Send (color, i, v, hop_counter + 1, node_counter, edge_counter) to p
21  |   else
22  |   |   answers[i] ← (color, node_counter, edge_counter)
23  |   fi
24  done

Receive(index, answer) from v
25  (c, hop_counter, node_counter, edge_counter, p) ← tokens[index]
26  if color = answer.color then
27  |   answers[index] = answer
28  fi
```

Figure 3: Self-stabilizing monitoring algorithm for u.

Each node u will follow the algorithm presented in Figure 3: ml will add a serial number to each token it sends, e.g., since ml sends n tokens initially, they will be consecutively numbered, $t_1, t_2, \ldots, t_n$. Each node u records not only the color of the token, but also, for each token t_i, to which neighbor, w, u sent token t_i following the last arrival of t_i in u (lines 8 and 11). We call this the *forward pointer* of t_i at u. u then repeatedly sends all the tokens u has received to ensure delivery (line 20). We use the repeated send technique to argue concerning the termination of each monitoring phase.

To ensure the delivery of the tokens back to ml, when they have reached the end of their random walk, we need the following mechanism; when a random walk terminates (reached its

maximum hop count), the node in which the walk terminated saves the information collected by the walk (line 11). Each time a node u receives a token (whether for the first time or not), if u currently holds an answer to the token, u forwards the answer to the sender of the token (line 15).

We next show that the traversal of a single token sent by ml will terminate, and the information contained in the token (node count and edge count) will be propagated back to ml. The proof considers one token, t_i. Denote by k the length of the random walks according to the previous section ($k \in O(\log n)$).

Definition 4.2. We say that a node v contains a token t_i in configuration c if one of the following conditions holds:
• v has received t_i in c for the first time
• v has a forward pointer for t_i (with a hop count of $j < k$), which points at u, and the maximal hop count of t_i in u (if it exists) is smaller than j.

Lemma 4.11. *Let c_0 be a configuration, and T_c be the set of all instances of t_i in a configuration c (t_i might appear twice due to faults). Further, let c_k be a configuration following c_0, in a fair execution, after the first k communication rounds. Assuming ml has not initiated a new monitoring phase using a new color, then $T_{c_k} = \emptyset$.*

Proof. Nodes repeatedly forward token messages to their neighbors. As a result, if at configuration c_i node v contained a token, than after one communication round v would have forwarded the token and increased the token's hop counter. Define $H(T_c)$ as the minimal hop counter of a token in T_c, and c_{i+1} as the first configuration following a single communication round. It immediately follows that $H(T_{c_i}) < H(T_{c_{i+1}})$. Now, since tokens are terminated after reaching a large enough hop counter, k, we get that after k rounds, starting in c_0, there can be no tokens in c_k. □

It easily follows from Lemma 4.11 that for any configuration c_j, such that $j \geq k$, if ml does not initiate a new monitoring phase, no node will change its forward pointer that is related to t_i.

Lemma 4.12. *Under the notation of Lemma 4.11, in c_k there exists a path from ml to a node t, in which t_i terminated, which is defined by the forward pointers of t_i starting from ml. Moreover, this path will not change unless ml initiates a new monitoring phase.*

Proof. Let c_m, $c_0 \leq c_m \leq c_k$, be the last configuration in which ml contained t_i (if no such m exists, set $m = 0$). Since ml does not hold t_i between c_m and c_k, ml's forward pointer for t_i will not change.

Let v_1 be the node to which ml's forward pointer points. Let c_{m_1} be the last configuration in which v_1 contained t_i. There exists such a configuration c_{m_1}, since according to Lemma 4.11 there are no tokens in c_k. Since v_1 does not hold t_i between c_{m_1} and c_k, v_1's forward pointer for t_i will not change.

Using the same argument, we define a sequence of nodes, $r = v_0, v_1, v_2, \ldots, v_l$, such that v_j's forward pointer for t_i points at v_{j+1} in c_k, and will not change until ml starts a new monitoring phase. □

Lemma 4.13. *Under the notation of Lemma 4.11, after further k communication rounds at most, ml will receive a notification for the termination of t_i.*

Proof. The proof is by induction, showing that after each round, the notification advances at least one node backwards on the path defined by Lemma 4.12. □

The next theorem follows immediately from the lemmas above. In particular it follows that after the first time *ml* initiates a new monitoring phase using a new color, the monitoring phase will complete successfully. Thus,

Theorem 4.2. *The monitoring algorithm is self-stabilizing.*

4.3.4 Global Monitoring Algorithm

Given the monitoring algorithm and the reset procedure above, we develop a self-stabilizing expansion monitoring algorithm. The algorithm is based on repeated invocation of the monitoring algorithm for a single node, where each instance of the monitoring is carried by a different node.

To coordinate between the nodes, and to ensure only one monitoring session is active at a time, we employ a single leader in the system, with a BFS tree rooted in this same leader. Repeatedly, the BFS leader, bl, selects a directed edge, $\langle u, v\rangle$, uniformly among all directed edges, and informs v, the edge's endpoint, to start a new monitoring session as the monitoring session leader. If a specific percentage of monitoring sessions failed, bl may conclude that the graph is not a good expander and initiate a reset to rebuild the graph.

We next detail the specific techniques used to realize the global monitoring algorithm.

• **Self-Stabilizing BFS tree**. The BFS tree is assumed to be defined. Nodes will repeatedly inspect the status of the tree, and upon finding any error will initiate a reset. Namely, each node v repeatedly checks that v's neighbors have the same leader v has. In case v is the leader, v also checks whether its distance to the BFS root is 0. Otherwise, when v is not a leader, v checks whether v's parent in the BFS is closer to the leader than v by exactly one. Once any node (either a leader or a non-leader node) detects a violation of the above assertions, the node will initiate a snap-stabilizing reset, ensuring that a single BFS leader will be elected and a correct BFS tree will be constructed.

To facilitate communications between the BFS leader and the rest of the nodes, the BFS leader will repeatedly color the BFS defined by the parent child relation, using randomly selected colors. Such a self-stabilizing coloring technique using broadcast and convergecast over a tree has been well investigated, see [7].

When coloring the BFS tree with a new color, the BFS leader may piggy back messages on the messages used for coloring, and nodes may piggy back information back to the BFS leader with their replies.

• **Selecting the next monitoring node**. Each monitoring session must start from a directed edge, chosen uniformly. To select the edge, the BFS leader, bl, will employ the coloring of the tree defined above.

During and after the construction of the BFS tree each node v will be repeatedly notified by its children on se_u, the sum of the edges connected to all nodes in the subtree rooted in each child u. v will repeatedly define se_v to be $\Delta_v + \sum_{u \text{ child of } v} se_u$, where Δ_v is the number of edges directly connected to v. In particular, a leaf u will repeatedly notify its parent with Δ_u. Moreover, each node maintains a *local* ordering of the node's children.

Each node u associates a set of natural numbers between 1 and se_u with u's children and with u itself, so that each number is associated with a single node and each node v receives a set of numbers, N_v, of cardinality se_v so that $N_v = \{i_1, i_2, \ldots, i_{se_v}\}$.

The BFS leader, bl, initiates the selection of the new node by first selecting, uniformly at random, a natural number k between 1 and se_l. If k is associated with bl, bl is the new monitoring leader. Otherwise, let v be bl's child with which k is associated, and let $k = i_j \in N_v$. bl will inform v (by piggy-backing on the coloring algorithm's messages) that v should continue the process with the number j.

When a node v receives a message that v should select the next monitoring leader using the number k, v first checks with which node k is associated. If k is associated with v itself, v selects itself as the new monitoring leader. Otherwise, assume k is associated with a child, u, such that $k = i_j \in N_u$. v will notify u that u should continue the process with the number j. Note that the selection algorithm always terminates and selects a node with a uniform distribution between all edges' endpoints.

At the end of the coloring of the BFS tree, *bl* learns the identity of the node which was selected as the next monitoring session. Moreover, the natural number, k, which *bl* has randomly chosen, uniquely identifies the next monitoring leader. *bl* will then initiate a new coloring of the tree, in which *bl* broadcasts the identity of the new monitoring leader. Next, we describe the mechanism used to ensure that the monitoring leader indeed performs the monitoring.

• **Ensuring a monitoring leader exists**. *bl* must ensure that the current monitoring leader is active. There are several ways to achieve this, amongst them are using the coloring to communicate with the current monitoring leader. A different way, which does not involve the coloring algorithm is by routing messages directly to the monitoring leader, down the BFS tree, by employing the same mechanism used to select the monitoring leader; as *bl* remembers k, the randomly chosen natural number which identifies the monitoring leader, *bl* can route messages to the monitoring leader which can then send replies up the tree towards *bl*. To ensure self-stabilization, each time *bl* sends a message, *bl* will attach a random color to the message and expect a reply containing the same random color.

If *bl* does not receive a confirmation that a monitoring leader exists, *bl* will perform a reset to reinitialize the BFS tree.

• **Detecting bad expansion**. There exists a small constant c, such that if the monitoring algorithm (for a single node) indicated that the graph is not a good expander at least $c/2$ times when running c monitoring sessions consecutively (from different nodes), then with probability $1-o(1)$ the graph is not a good expander. This follows from a simple expansion of the conditional probability that the graph is not a good expander given that the monitoring algorithm indicated so in more than half the times of c successive sessions, and that the probability of a wrong answer on a good expander and the probability of getting a good expander a priori are both $1 - o(1)$.

bl will employ this constant to check the last c invocations of the monitoring algorithm. If more than $c/2$ invocations failed, *bl* will initiate a reset to rebuild the graph. We wish to draw the reader's attention to the fact that monitoring results may not be reused; e.g., once c successive monitoring sessions are finished, a new set of successive monitoring sessions will start.

Note that one may employ more than one monitoring session in parallel, boosting the speed of the detection time.

• **Stretch factor**. The diameter of the resulting spander graph is in $O(\log n)$. This, in turn, implies that the spander has an additive stretch factor in $O(\log n)$.

5 Distributed Hierarchical Spanner Construction

Given a graph $G = (V, E)$ satisfying the conditions of Lemma 3.1 (namely, having an edge expansion of at least $c \cdot \log n$), we wish to define a hierarchical structure of spanders such that each spander will have fewer edges than the one before it (and, as a by-product, smaller expansion). This hierarchical construction can then be used to ensure quality of service in a wide variety of applications; the system can automatically adjust the communication graph used in order to achieve its goals, taking into account the underlying structure of the chosen graph and the fitting number of edges versus the probability of expansion.

We propose a self-stabilizing and self-organizing distributed algorithm based on the techniques shown above. Namely, we propose to distributively define a sequence of spanders, $\{G_i\}_{i=0}^{k}, G_0 = G$, where k is an input to the construction algorithm, such that each graph g_i results from G_{i-1} by applying the algorithm portrayed in Section 3.2. Furthermore, to ensure that each spander is indeed up to the standard, we apply the monitoring technique portrayed in Section 4.2 by starting a monitoring phase from each node and for each spander in the hierarchy. If one node discovers that a given spander G_i is either not sparse enough or not a good expander, it will start a snap-stabilizing reset which restarts the construction from G_i onwards.

It immediately follows that the proposed hierarchical construction is self-stabilizing; since each graph of the hierarchy is constantly monitored by a self-stabilizing monitoring protocol, once one of the graphs is found to be faulty the snap-stabilizing reset protocol will tear down the hierarchy from the faulty graph forward (using the reset procedure). The construction will then resume from the latest graph known to be a good expander. It follows, according to the fair composition technique ([7], Chapter 2.7), that the hierarchical construction will eventually stabilize.

• **Maximal hierarchy height**. When considering possible values for k, the height of the hierarchy, several considerations come to mind; assuming the probability of edge selection in each level of the hierarchy is p, at each level i there are at most $p^i|E_0|$ edges. In fact, we check that the number of edges in each level is not too large. As a result, when $i \in O(\log n)$ the spander construction will fail. This implies that at most $O(\log n)$ levels in the hierarchy can exist and therefore appropriate memory and communications resources for this number of levels are needed.

• **Self-organization**. The construction is self-organizing; the construction of the graph is a local computation, carried by each node locally and independently of neighboring nodes. Each instance of the monitoring protocol takes $O(\log n)$ communication rounds to complete. Taking into account that following a constant number of successful monitoring phases, with very high probability the graph is a good expander, we then get that a hierarchy of height k will stabilize in $O(k \log n)$ communication rounds, if a reset procedure is invoked at every level of the hierarchy in the worst case.

Another property of self-organizing algorithms is fast reaction time to topology changes, after the algorithm has converged. Consider a node, v, joining G_0. If G_0 will remain a good expander and satisfy the conditions of Lemma 3.1, then the following simple procedure, carried by v, will ensure that each graph in the hierarchy will be a good expander: v will select each edge of G_i ($i > 0$) to be in G_{i+1} with probability p (the same probability as in the construction algorithm); each G_i can still be seen as a random graph drawn from the probability space induced by our construction algorithm, from which it follows that each graph is a good expander with high probability.

Now, consider the effect v has on the monitoring algorithm. We first require that v will join the BFS tree by selecting the neighbor with the shortest distance to the BFS leader as v's BFS parent. v will then notify v's parent with Δ_v, such that the sum of degrees held by each node on the path from v to the BFS leader will also be updated. Next, v will check that the height of the BFS tree is not too big, say, bigger than $2 \log n$. If the height is too big, v will initiate a reset to rebuild the BFS tree.

Overall, if the pattern of nodes joining G_0 is symmetric, meaning each node joining the graph is connected to other nodes using a uniform distribution, then the number of nodes joining the network between two successive resets is large. Hence, the expected, in fact, amortized, convergence time following a join is short and in $O(1)$.

Next, consider a node, v, leaving the graph or crashing. If the basic graph, G_0, remains

an expander with the required properties as in Lemma 3.1 even after v is removed, then from a similar argument as the one for nodes joining the network, each graph G_i in our hierarchy will remain a good expander with high probability. Let us now consider the implications of v's crash on the monitoring algorithm.

Each of v's neighbors in G_0 will detect that v has crashed. Each node u such that v was u's parent in the BFS tree will need to select a different parent, and update this new parent with se_u. Moreover, v's parent, w, will need to deduce se_v from se_w. Regarding the current monitoring session, each node v will need to check if v is currently holding a token which it needs to forward to a crashed node. If so, v will need to forward the token to a different node.

If v was the monitoring leader, then a new monitoring leader will be selected by the BFS leader. In the case that v is itself the BFS leader, a new reset procedure will be started.

Overall, the correct behavior of the monitoring algorithm will not be harmed. The total communication round which it would take to update the BFS tree following a crash is expected to be in $O(1)$ when the probability of node failure is uniform.

Furthermore, the same argument holds when edges are removed or added, as long as G_0 satisfies Lemma 3.1. As a result, the construction algorithm converges in an expected (amortized) $O(1)$ communication rounds following a topology change.

• **Hierarchical reset**. Consider the reset protocol presented earlier, which we perform on the original graph, G, when building the hierarchy. The message complexity of the reset algorithm is in $O(d \cdot |E|)$, where d is the diameter of the graph and E is the set of edges. When G is sparse, the reset algorithm is efficient. However, when G is dense, i.e., $|E| \in O(n^2)$, the communication complexity of the reset algorithm is in $O(n^2)$.

To lower the communication cost of the reset algorithm we propose to perform the reset algorithm associated with the monitoring algorithm of G_{i+1} on G_i. However, one problem may arise; we need to ensure that G_i is indeed connected. Otherwise, the reset algorithm will fail to reset the entire graph. To ensure that G_i is connected, we augment the reset protocol in the following way; the BFS leader will count the nodes in the tree in addition to coloring the tree. If the number of nodes is too small, or differs from the number of nodes in $G = G_0$, then a reset on G_{i-1} will need to be started to rebuild G_i. Alternatively, the BFS of G_0 will be used to update the leader identity of each level.

Theorem 5.1. *The proposed hierarchical distributed expander construction algorithm is self-stabilizing and self-organizing.*

To sum, the convergence time of a hierarchy of height k is in $O(k \log n)$ and the memory required at each node, imposed by the monitoring algorithm, is in $O(kn \log n)$.

Acknowledgements. It is a pleasure to thank Eitan Bachmat and Noga Alon for helpful discussions. In particular, some of the techniques were explored during and after discussions with Noga.

References

[1] Alon, N., private communications.

[2] Alon, N., Avin, C., Koucky, M., Kozma, G., Lotker, Z., Tuttle, M. R. "Many Random Walks Are Faster Than One". Unpublished.

[3] Arora, A., Dolev, S., and Gouda, G. M., "Maintaining Digital Clocks in Step", *Parallel Processing Letters*, Vol. 1, No. 1, pp. 11–18, 1991.

[4] Arora, A. and Nesterenko, M., "Unifying stabilization and termination in message-passing systems", *Distributed Computing*, Vol. 17, No. 3, pp. 279-290, 2005.

[5] Awerbuch, B., Varghese, G. "Distributed program checking: a paradigm for buildingself-stabilizing distributed protocols". *FOCS 1991: 32nd Annual Symposium on Foundations of Computer Science*.

[6] Fan R. K. C., "Spectral Graph Theory", *Providence, RI*: American Mathematical Society. ISSN 0160-7642; no 92.

[7] Dolev, S., *Self-stabilization*, MIT Press, 2000.

[8] Diaconis, P., Stroock, D., "Geometric Bounds for Eigenvalues of Markov Chains", *the Annals of Applied Probability*, Vol. 1, No. 1(Feb., 1991), pp 36–61.

[9] Dolev S., Tzachar N., "Empire of Colonies: Self-stabilizing and Self-organizing Distributed Algorithms". *OPODIS 2006: Principles of Distributed Systems, 10th International Conference*. pp 230–243.

[10] Behrends. E., "Introduction to Markov Chains, with special emphasis on rapid mixing". *Veiweg*, 2000.

[11] Law, C., Siu, K.-Y., "Distributed construction of random expander networks", *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE , vol.3, pp 2133–2143. April 2003

[12] Lamport, L. "Time, clocks, and the ordering of events in a distributed system". *Commun. ACM* 21, 7 (Jul. 1978), pp. 558-565.

[13] Motwani R., Raghavan P., "Randomized Algorithms", chapter 5.3, Cambridge university press, 2006.

[14] Reiter, M.K., Samar, A., Wang, C., "Distributed construction of a fault-tolerant network from a tree," *24th IEEE Symposium on Reliable Distributed Systems*. SRDS 2005. pp 155–165. October 2005

[15] Sinclair, A., "Improved Bounds for Mixing Rates of Marked Chains and Multicommodity Flow", *LATIN '92: Proceedings of the 1st Latin American Symposium on Theoretical Informatics*, 1992. pp 474–487.