Linearization: Locally Self-Stabilizing Sorting in Graphs

Melih Onus* Andrea Richa† Christian Scheideler‡

Abstract

We consider the problem of designing a distributed algorithm that, given an arbitrary connected graph G of nodes with unique labels, converts G into a sorted list of nodes. This algorithm should be as simple as possible and, for scalability, should guarantee a polylogarithmic runtime as well as at most a polylogarithmic increase in the degree of each node during its execution. Furthermore, it should be self-stabilizing, that is, it should be able to eventually construct a sorted list from any state in which the graph is connected. It turns out that satisfying all of these demands at the same time is not easy.

Our basic approach towards this goal is the so-called linearization technique: each node v repeatedly does the following with its neighbors:

- for its left (i.e., smaller) neighbors $u_1, \ldots, u_k$ in the order of decreasing labels, v replaces $\{v, u_1\}, \ldots, \{v, u_k\}$ by $\{v, u_1\}, \{u_1, u_2\}, \ldots, \{u_{k-1}, u_k\}$, and
- for its right (i.e., larger) neighbors $w_1, \ldots, w_\ell$ in the order of increasing labels, v replaces $\{v, w_1\}, \ldots, \{v, w_\ell\}$ by $\{v, w_1\}, \{w_1, w_2\}, \ldots, \{w_{\ell-1}, w_\ell\}$.

As shown in this paper, this technique transforms any connected graph into a sorted list, but there are graphs for which this can take a long time. Hence, we propose several extensions of the linearization technique and experimentally evaluate their performance. Our results indicate that some of these have a polylogarithmic performance, so there is hope that there are distributed algorithms that can achieve all of our goals above.

1 Introduction

Due to their many applications, peer-to-peer systems have recently received a lot of attention both inside and outside of the research community. Peer-to-peer systems are usually dynamic in nature, that is, peers continuously enter and leave the system. Also, peers may fail. In order to handle such a situation, distributed algorithms are needed that can quickly and reliably recover a peer-to-peer system from any inconsistent state. Systems with such a property are also known to be self-stabilizing. In general, self-stabilization of a system guarantees that regardless of a current system's state, the system will converge to a legal state in a finite number of steps. Before presenting previous work on self-stabilization, we start with some basic definitions.

DEFINITION 1.1. *A* system *is a pair $S = (C, \rightarrow)$ where C is a set of states of the system and $\rightarrow$ is a binary transition relation on C. A* computation *of S is a non-empty sequence $(c_1, c_2, \ldots)$ such that for all $i \geq 0$, $c_i \in C$ and $c_i \rightarrow c_{i+1}$.*

In distributed systems the above mentioned c_i expresses a global state, which is the concatenation of the local states of every process and the contents of every communication channel. A commonly used definition of self-stabilization is:

DEFINITION 1.2. *A system S is* self-stabilizing *with respect to a set of* legal *states $L \subset C$ if for any state $c \in C$, any computation of the system starting with c will eventually reach a state in L.*

The subset L is problem-specific. Often, it is assumed that L is *closed*, i.e. every non-faulty move from a state in L ends in a state in L. An algorithm is said to be self-stabilizing if it promotes self-stabilization of the system. A *locally self-stabilizing* algorithm is a self-stabilizing algorithm that runs locally at each node and that only assumes *local* knowledge of the network by the nodes (e.g., the algorithm does not assume that nodes know or have an estimate on the number of nodes in the network).

It turns out that the crux of designing truly self-stabilizing algorithms for many structured overlay peer-to-peer networks such as Chord and skip graphs is to find a self-stabilizing protocol that can convert an arbitrary connected graph into a sorted list or ring. For scalability, such algorithms should converge to a sorted list in at most a polylogarithmic number of rounds. A distributed algorithm for arranging the nodes of an arbitrary connected graph into a sorted list in a polylogarithmic number of rounds and with polylogarithmic work for each node is already known [1], but the algorithm is not locally self-stabilizing. In fact, under certain conditions it can run into a deadlock (i.e.,

*Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-8809, USA. Email: melih@asu.edu

†Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-8809, USA. Email: aricha@asu.edu. This work was supported in part by NSF CAREER award no. 9985284.

‡Institute for Computer Science, Technical University of Munich, 85748 Garching, Germany. Email: scheideler@in.tum.de

the nodes cannot escape a certain invalid state). Hence, there exists the need for developing truly (locally) self-stabilizing algorithms that can converge to a sorted list from an arbitrary initial state in polylogarithmic time.

We are now ready to formally state the problem we consider in this paper. We are given an arbitrary connected graph $G(V, E)$, $|V| = n$, with node labels which induce a total order on V. For nodes u and v, we say that $u < v$ (resp., $u > v$), whenever the label of node u is smaller (resp., greater) than that of node v. For simplicity, we assume that all the node labels are distinct. We consider the problem of designing a distributed, locally self-stabilizing algorithm that converts G into a sorted list $H(\{v_1, \ldots, v_n\}, E_H)$, where $v_i \in V$, and $v_{i-1} < v_i$ and $\{v_{i-1}, v_i\} \in E_H$ for all i.

The idea of self-stabilization in distributed computing first appeared in a classical paper by E.W. Dijkstra in 1974 [6] in which he looked at the problem of self-stabilization in a token ring. Since Dijktra's paper, self-stabilization has been studied in many contexts, including communication protocols, graph theory problems, termination detection, clock synchronization, and fault containment. For a recent survey see, e.g., [4], and a *very* comprehensive list of papers can be found in Herman's Self-Stabilization Bibliography from 2002 [8]. For more information about self-stabilizing algorithms we also recommend the book by Dolev [7].

Self-stabilization issues have also been considered for peer-to-peer networks. In the technical report of the Chord system [10], for example, several techniques such as the weak stabilization protocol and the strong stabilization protocol are presented that allow the Chord network to recover quickly from various kinds of degenerate states which the authors call *pseudo-trees* or *loopy states*. However, the running time of these algorithms depends on the degree of distortion of the degenerate state, and can be as high as linear on the number of nodes. Moreover, no technique is given that allows the Chord network to recover from an arbitrary connected state. Instead, to prevent Chord from ever running into a state outside of pseudo-trees or loopy states, it is suggested that each node keeps $\Theta(\log n)$ successor pointers instead of just one. In this way, nodes can recover their successor pointers with high probability even if the nodes in the system fail with constant probability. Though this protects against faulty nodes, it is known that this technique will not protect against adversarial nodes [3], and therefore it is still important to understand how to recover from an arbitrary state.

In a technical report on skip graphs [2] the authors propose a self-stabilization protocol in which each node v periodically checks six locally checkable conditions. If one of these conditions is violated, it is locally repaired. It is shown that an overlay network forms a skip graph if and only if all of the six conditions are satisfied for all nodes. However, the local checking of the conditions does not work for arbitrary states but only for states in which the nodes form a degenerate version of a skip graph that may be created due to node faults or inconsistencies in concurrent join or leave operations. As for the Chord repair mechanisms, the total running time for repairing the conditions at each node depends on the distortion of the degenerate state and can be as high as linear.

Self-stabilizing protocols for organizing the nodes into a sorted ring have already been proposed in [5, 9]. In the Iterative Successor Pointer Rewiring Protocol (ISPRP) proposed in [5], each node aims at maintaining correct successor pointers. If there is a local inconsistency, i.e., there are two nodes u and v with successor pointers to a node w, w asks either u or v (depending on their label) to repair its successor pointer. In the Ring Network (RN) protocol proposed in [9], each node v periodically initiates the search for a closest successor by sending a search request to a randomly chosen neighbor. Any node w receiving such a request forwards it to the neighbor of w closest to v. This is continued until there is no such neighbor, or a neighbor is found that is closer to v than its closest successor. In the latter case, v is informed about it. Both protocols can have a very long runtime. Hence, the question remains whether there are simple self-stabilizing protocols with a polylogarithmic runtime.

1.1 Our contributions We propose linearization as the basic technique to sort any connected graph. This technique works as follows: each node v repeatedly does the following with its neighbors:

- for its left (i.e., smaller) neighbors $u_1, \ldots, u_k$ in the order of decreasing labels, v replaces $\{v, u_1\}$, $\ldots, \{v, u_k\}$ by $\{v, u_1\}, \{u_1, u_2\}, \ldots, \{u_{k-1}, u_k\}$, and

- for its right (i.e., larger) neighbors $w_1, \ldots, w_\ell$ in the order of increasing labels, v replaces $\{v, w_1\}$, $\ldots, \{v, w_\ell\}$ by $\{v, w_1\}, \{w_1, w_2\}, \ldots, \{w_{\ell-1}, w_\ell\}$.

As shown in this paper, the linearization technique is self-stabilizing and transforms any connected graph into a sorted list, but there are graphs for which this can take a long time. Even for random graphs, the pure linearization (PL) technique appears to perform poorly. Our goal is to find self-stabilizing protocols that are guaranteed to have a polylogarithmic runtime (with high probability), or at least a polylogarithmic runtime

for random graphs. Hence, we propose several extensions of the linearization technique and experimentally evaluate their performance. These are:

- Linearization with memory: Each node only linearizes the edges that it got in the previous round but also maintains ("remembers") edges to all nodes that it was connected to at any time. It uses this node list to accelerate the linearization process of the new edges. See Section 3 for details.
- Linearization with shortcut neighbors: Each node only linearizes the edges that it got in the previous round but also maintains edges to a certain set of $O(\log n)$ other shortcut nodes that it learned about over the time. Also, every node exchanges its set of shortcut nodes with every node it is connected with by a new edge. See Section 4 for details.

Though linearization with memory (LM) still has a linear runtime in the worst case, our experiments demonstrate that for random graphs the LM protocol significantly outperforms the PL protocol. LM appears to have a polylogarithmic runtime for these graphs, and the node degree seems to be polylogarithmic as well. Since linearization with shortcut neighbors (LSN) is based on the concept of a virtual space, also this protocol can perform poorly in the worst case (extremely unevenly distributed node labels), but our experiments with LSN demonstrate that it is still much better than the LM protocol for random graphs. Thus, LSN appears to be a good candidate for further, rigorous investigations towards the goal of finding a self-stabilizing protocol with polylogarithmic runtime.

1.2 Structure of the paper In Section 2 we present and analyze the pure linearization technique. Afterwards, we study the linearization with memory technique in Section 3 and the linearization with shortcut neighbors technique in Section 4. Section 5 presents and interprets the simulation results for all three techniques. The paper ends with conclusions.

2 Pure Linearization

In this section we present the pure linearization technique. We are given as input a connected undirected graph $G(V, E)$ with node labels. At any time t during the execution of the algorithm, we will let $G(V, E)$ denote the current configuration of graph G, after edges may have been inserted or removed from the original edge set during prior time steps. We start with some definitions:

DEFINITION 2.1. *At any time t, node v is a* left (resp., right) neighbor *of node u if $\{u, v\} \in E$ at time t and $v < u$ (resp., $u < v$).*

Figure 1 illustrates the left and right neighbors of a node u with label 10.

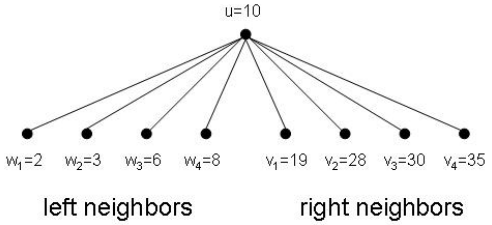


Figure 1: Left and right neighbors of node u

In the pure linearization algorithm (PL), each node repeatedly converts its left neighborhood and its right neighborhood into sorted lists at each round. More specifically, if a node u has left neighbors $w_1 < w_2 < \ldots < w_k$ at the start of a left linearization step, it converts its left neighbors into a sorted list by removing the edges $\{u, w_i\}$, $1 \leq i \leq k-1$ and adding the edges $\{w_i, w_{i+1}\}$, $1 \leq i \leq k-1$ (Figure 2(a)). Similarly, if $v_1 < v_2 < \ldots < v_j$ are the right neighbors of u at the start of a right linearization step, u removes the edges $\{u, v_i\}$, $2 \leq i \leq j$ and adds the edges $\{v_i, v_{i+1}\}$, $1 \leq i \leq j-1$ (Figure 2(b)) to the graph. Note that in a round every node first converts its left neighbors into a sorted list and then converts its right neighbors into a sorted list. If at any step there is a conflict where a node u wants to remove an edge $\{u, v\}$ and an another node w wants to add this edge $\{u, v\}$, the final outcome will be that the edge $\{u, v\}$ remains in the graph. Algorithm 1 summarizes the PL algorithm.

Algorithm 1 Pure Linearization (PL)

Each node u executes the following:

1: **for** each round **do**
2: (Left linearization step) If $w_1 < w_2 < \ldots < w_k$ are the current left neighbors of u, u removes the edges $\{u, w_i\}$, $1 \leq i \leq k-1$ and adds the edges $\{w_i, w_{i+1}\}$, $1 \leq i \leq k-1$ to the graph.
3: (Right linearization step) If $v_1 < v_2 < \ldots < v_j$ are the current right neighbors of u, u removes the edges $\{u, v_i\}$, $2 \leq i \leq j$ and adds the edges $\{v_i, v_{i+1}\}$, $1 \leq i \leq j-1$ to the graph.
4: **end for**

The following two sections provide a formal analysis of the complexity and several other properties of the PL.

2.1 Analysis. Note that the PL algorithm is memoryless, that is, it only needs to know the current state

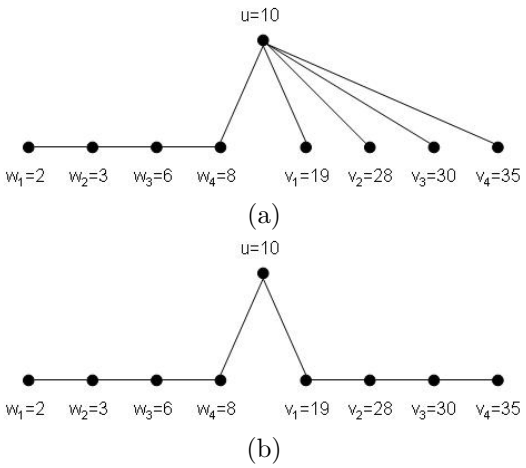


Figure 2: Neighbors of node u after one round of pure linearization.

of the connections in order to converge to a sorted list. Hence, if we can show that the PL algorithm converges to a sorted list for any connected graph, the PL algorithm is self-stabilizing. In the following two theorems we will prove matching lower and upper bounds showing that in the worst case the PL algorithm requires $(n-2)$ rounds to converge to a sorted list.

THEOREM 2.1. (LOWER BOUND) *The pure linearization algorithm requires at least* $(n-2)$ *rounds to generate a sorted list out of an arbitrary graph* G.

Proof. Consider the example graph $G(V,E)$, $E = \{\{i,i+1\}|i \in \{2,3,...,n-1\}\} \cup \{\{n,1\}\}$ illustrated in Figure 3. After the first round, the edge $\{n,1\}$ is removed and the edge $\{n-1,1\}$ is added to G. After the second round, the edge $\{n-1,1\}$ is removed and the edge $\{n-2,1\}$ is added to G. After the third round, the edge $\{n-2,1\}$ is removed and the edge $\{n-3,1\}$ is added to G. The PL algorithm will take $(n-2)$ rounds to convert G to a sorted list since after the j^{th} round, the edge $\{n-j+1,1\}$ is removed and the edge $\{n-j,1\}$ is added to G, $1 \leq j \leq n-2$. □

Figure 3: Lower bound example for PL algorithm

THEOREM 2.2. (UPPER BOUND) *After* $(n-2)$ *rounds, the pure linearization algorithm will generate a sorted list out of any graph* G.

Proof. Let $V = \{a_1, a_2, a_3, ..., a_n\}$, where $a_i < a_j$ for all $i < j$, $i,j \in \{1,2,...,n\}$. Recall that one round of the PL algorithm consists of a left linearization step followed by a right linearization step. The following three claims will be used in the proof of Theorem 2.2. The claims bound the number of left and right neighbors out of each node after each round of the PL algorithm.

CLAIM 2.1. *After* k *rounds* $a_n, a_{n-1}, a_{n-2}, ..., a_{n-k+1}$ *will each have exactly one left neighbor for every* $k = 1,2,...,n-2$.

Proof. We will prove this claim by induction on k.

Base case: $k = 1$. We will show that after the first round a_n will have exactly one left neighbor. During the left linearization step of the first round, a_n will remove all of its left neighbors except for one. Since a_n is the largest node, it does not have any right neighbors. Thus no node can add a left neighbor to a_n during a left linearization step. Hence after the left linearization step, a_n will have only one left neighbor, which we call b_n. If b_n only has one right neighbor (a_n itself) after the right linearization step, the left neighbor of a_n will not change. If b_n has more than one right neighbor, after the right linearization step, the left neighbor of a_n will change and it will be the second largest right neighbor of b_n. Thus, after the first round a_n will have only one left neighbor.

Inductive Hypothesis: After k rounds $a_n, a_{n-1}, a_{n-2}, ..., a_{n-k+1}$ will each have exactly one left neighbor, $k < n-2$.

Inductive Step: We will show that after $k+1$ rounds $a_n, a_{n-1}, a_{n-2}, ..., a_{n-k+1}, a_{n-k}$ will each have exactly one left neighbor, $k \leq n-2$.

We will first show that after $k+1$ rounds $a_n, a_{n-1}, a_{n-2}, ..., a_{n-k+1}$ will still have one left neighbor each. From the inductive hypothesis, we know that after k rounds $a_n, a_{n-1}, a_{n-2}, ..., a_{n-k+1}$ will have only one left neighbor each. Consider node a_i, $n-k+1 \leq i \leq n$ at round $k+1$. Since a_i only has one left neighbor it will not do anything during the left linearization step of this round. Since all nodes $a_j > a_i$ have only one left neighbor each, they cannot add a left neighbor to a_i. So, after the left linearization step of round $(k+1)$, a_i will still have only one left neighbor (say b_i). If b_i has no right neighbor smaller than a_i, the left neighbor of a_i will not change after right linearization step of round $k+1$. If b_i has a right neighbor smaller than a_i, the left neighbor of a_i will change and it will be the largest right neighbor of b_i which is less than a_i. Thus, after $k+1$ rounds a_i will have only one left neighbor, $n-k+1 \leq i \leq n-1$.

Secondly, we will show that after $k+1$ rounds a_{n-k} will also have only one left neighbor. From the inductive hypothesis, we know that after k rounds $a_n, a_{n-1}, a_{n-2}, ..., a_{n-k+1}$ will have only one left neighbor each. We start by showing that a_{n-k} must have at least one left neighbor after round $k+1$. Then we will show that a_{n-k} cannot have more than one left neighbor after this round. By contradiction, assume that after k rounds a_{n-k} has no left neighbor. Since all nodes greater than a_{n-k} have only one left neighbor each, all of the right neighbors of a_{n-k} must have only one left neighbor which is a_{n-k} (if a_{n-k} has no right neighbor the graph is disconnected.). Since the graph is connected, a_{n-k} must have a path to a_1. However, the right neighbors of a_{n-k} cannot have a left neighbor smaller than a_{n-k} (since a_{n-k} is their only left neighbor) and similarly the right neighbors of the right neighbors of a_{n-k} cannot have a left neighbor smaller than a_{n-k}, and so on. Therefore, there is no path between a_{n-k} and a_1 which is a contradiction.

Now we show that a_{n-k} cannot have more than one left neighbor after round $(k+1)$. We have just seen that after k rounds a_{n-k} has at least one left neighbor. During the left linearization step of round $(k+1)$, a_{n-k} will remove all of its left neighbors except one. Since all nodes $a_j > a_{n-k}$ have only one left neighbor, they cannot add a left neighbor to a_{n-k}. So, after the left linearization step, a_{n-k} will have only one left neighbor (say b_{n-k}). If b_{n-k} has no right neighbor smaller than a_{n-k}, the left neighbor of a_{n-k} will not change during the right linearization step of round $(k+1)$. If b_{n-k} has a right neighbor smaller than a_{n-k}, the left neighbor of a_{n-k} will change and it will become the largest right neighbor of b_{n-k} which is less than a_{n-k}. Hence, after $k+1$ rounds a_{n-k} will have exactly one left neighbor, completing the proof. □

Claim 2.2. *After $n-2$ rounds all nodes except a_1 will have exactly one left neighbor each.*

Proof. From Claim 2.1 we know that after $n-2$ rounds $a_n, a_{n-1}, a_{n-2}, ..., a_3$ will have one left neighbor each. We show that a_2 will also have exactly one left neighbor (a_1). After $n-2$ rounds, since the graph is connected, there must be a path between a_2 and a_1. By contradiction, assume a_2 and a_1 are not neighbors, implying that a_2 has only right neighbors. By Claim 2.1, the right neighbors of a_2 only have right neighbors and so on. Thus we cannot reach a_1 from a_2, a contradiction. □

Claim 2.3. *After $n-2$ rounds all nodes except a_n will have exactly one right neighbor each.*

Proof. This statement is the dual of Claim 2.2 and its proof is analogous. The main difference in the proof is that we start the algorithm with a left linearization, but this will not affect the proof significantly. □

We are now ready to prove Theorem 2.2. We know, by Claim 2.2 and Claim 2.3, that each node a_i has exactly one left and one right neighbor after $n-2$ rounds of the PL algorithm. In addition, we will show that the (only) left neighbor of node a_i must be a_{i-1}, $1 < i \leq n$. This is trivially true for $i = 2$. Assume that the left neighbor of a_i is a_{i-1}, and hence the (only) right neighbor of a_{i-1} is a_i, for all $1 \leq i \leq k$. This trivially implies that the left neighbor of a_{k+1} must be a_k (and the right neighbor of a_k must be a_{k+1}), and hence $a_1, ..., a_n$ is a sorted list of V. □

2.2 Other properties of PL In the following claims, we will present several properties of PL algorithm regarding the degree of the nodes and the required number of rounds.

Claim 2.4. *At each round of the PL algorithm, the degree of a node v increases by at most an additive constant.*

Proof. Assume a node u has l left and r right neighbors at the start of a round. After the left linearization step of this round, node u will have (i) at most $r+1$ left neighbors, since each of its right neighbors can add at most one more left edge to u and only one of its previous left neighbors will remain; and (ii) node u will have at most r right neighbors, since each of its right neighbors will either remain or be changed to another node. In summary, after the left linearization step, node u will have at most $r+1$ left and r right neighbors. Similarly, after the right linearization step of this round, node u will have at most $r+1$ left and $r+2$ right neighbors. Following the same argument, after the left linearization step of the next round, node u will have at most $r+3$ left and $r+2$ right neighbors, and after the corresponding right linearization step, node u will have $r+3$ left and $r+4$ right neighbors. Hence, after t left and right linearizations steps, node u will have at most $r+2t-1$ left and $r+2t$ right neighbors. □

An edge $\{u, w\}$ is said to *cross* a node v if $u \leq v \leq w$. The following claim bounds the degree of a node v by the number of edges crossing the node.

Claim 2.5. *The maximum degree of a node v during the PL algorithm is bounded by the number of edges crossing it.*

Proof. Linearization ensures that for any edge (u, w) converted into (u', w') during a linearization step: $u \leq u' \leq w' \leq w$.

3 Linearization with Memory

In this section we present an extension of the pure linearization technique called the linearization with memory (LM). In the linearization with memory algorithm, nodes remember all of their old neighbors. While placing current neighbors into sorted lists, a node u will also use its old neighbors in order to "better place" its current neighbors within the final total order, as we will show.

In the LM algorithm, nodes will place their current left and right neighbors into sorted lists by reconnecting them. When placing a left neighbor v into the sorted list, a node u will remove the edge $\{u, v\}$ and edge $\{v, w\}$ will be added, where w is the smallest node u remembers (w may be a current neighbor of node u) such that $v < w$. If v is the largest left neighbor of node u, edge $\{u, v\}$ remains. When placing a right neighbor v into the sorted list, a node u will remove the edge $\{u, v\}$ and edge $\{w, v\}$ will be added, where w is the largest node u remembers (w may be a current neighbor of node u) such that $w < v$. If v is the smallest right neighbor of node u, edge $\{u, v\}$ remains. For example, consider a node u that currently has left neighbors $v_1 < v_2 < v_3 < v_4$ and in addition remembers the nodes $w_1 < w_2 < w_3 < w_4$, where $w_1 < v_1 < v_2 < w_2 < v_3 < w_3 < w_4 < v_4 < u$. The LM algorithm would place u's left neighbors into the sorted list by removing the edges $\{u, v_1\}, \{u, v_2\}, \{u, v_3\}$ and adding the edges $\{v_1, v_2\}, \{v_2, w_2\}, \{v_3, w_3\}$, as illustrated in Figure 4. Algorithm 2 summarizes the LM technique.

Algorithm 2 Linearization with Memory (LM)

Each node u executes the following:

1: **for** each round **do**
2: Node u will remember all of its neighbors
3: For each current left neighbor v of u, u replaces edge $\{u, v\}$ by edge $\{v, w\}$, where w is the smallest node u remembers such that $v < w$, if such a node w exists. Otherwise, u keeps the edge $\{u, v\}$.
4: For each current right neighbor v of u, u replaces edge $\{u, v\}$ by edge $\{w, v\}$, where w is the largest node u remembers such that $w < v$, if such a node w exists. Otherwise, u keeps the edge $\{u, v\}$.
5: **end for**

While the LM technique alone still has a linear worst-case running time (see Corollary 3.1), its average running time seems to be polylogarithmic as the experimental results in Section 5 show, improving significantly on the average running time of the PL algorithm. The major drawback of the LM technique is that the degree of a node may increase very fast, since nodes "never forget" a neighbor (see Table 2). The linearization with shortcut neighbors (LSN), introduced in Section 4, is a refinement of the LM technique that aims at keeping the degree of the nodes lower (and at speeding up the LM process).

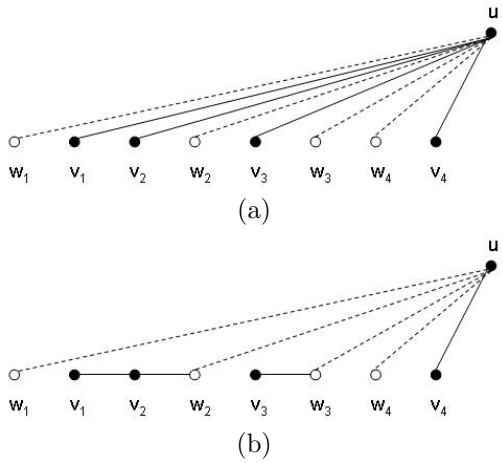


Figure 4: Node u places its current left neighbors into sorted list by reconnecting them.

We now show that LM algorithm requires $n - 2$ rounds to generate a sorted list of the nodes in the worst case. For any two nodes v and w, where $u < w$, let $|w-u|$ denote the cardinality of the set $\{v : u < v \leq w\}$.

THEOREM 3.1. *The linearization with memory algorithm still requires $n - 2$ rounds to generate a sorted list of the nodes in the worst-case.*

Proof. The LM algorithm will behave just like the PL algorithm for the graph in Figure 3 and hence the lower bound of Theorem 2.1 also applies to the LM algorithm.

The following two claims are needed to prove that the LM algorithm requires at most $n - 2$ rounds to generate a sorted list of the nodes.

CLAIM 3.1. *At any round of the algorithm, if a node u remembers a node $v < u$ (node v may be a current neighbor of u), then there exists an* increasing path *between u and v in the current graph, i.e. there exists a path $P = v_1, v_2, ..., v_p$, where $v_1 = v$, $v_p = u$, and v_i is a current left neighbor of v_{i+1}, $1 \leq i \leq j - 1$ (also implying that $v_1 < v_2 < ... < v_j$, $j \geq 2$).*

Proof. Let v be a left neighbor that u currently remembers. We will prove by induction on $|u - v|$.

Base case: $|u - v| = 1$. Node v is thus the largest left neighbor node u has seen. According to the LM algorithm, v must still be a current left neighbor of node u, and hence $P = vu$.

Inductive Hypothesis: If $|u - v| \leq k$, $k \geq 1$, there exists an increasing path between u and v in the current graph.

Inductive Step: We will show that if $|u-v| = k+1$, then there exists an increasing path P between u and v in the current graph.

If v is a current left neighbor of node u, $P = vu$ and we are done. Thus assume that node u only remembers node v. Since v is not a current left neighbor of node u, either u reconnected node v (i.e., u replaced edge $\{u, v\}$ by another edge in step 3 of the LM algorithm) or v reconnected node u, at some prior time. Without loss of generality, assume that u reconnected node v (the proof is analogous if v reconnected u), by replacing edge $\{u, v\}$ by an edge $\{v, w\}$, where w was (and may still be) a current neighbor of u and $v < w < u$. Hence $|v - w|$ and $|w-u|$ are at most k (and u remembers both v and w currently). By the induction hypothesis, there must exist increasing paths P_1 and P_2 in the current graph between v and w and between w and u, respectively. Hence P_1P_2 is an increasing path between u and v in the current graph and the induction hypothesis follows. □

Claim 3.2. *The graph remains connected after any number of rounds of the LM algorithm.*

Proof. From Claim 3.1, we know that if a node u remembers a node v, there exists an increasing path between u and v in the current graph (either v was a left neighbor of u at some prior time, or u was a left neighbor of v, and the theorem above applies). Nodes will always remember their initial neighbors. So, from Claim 3.1, there exists a current neighbor path between nodes and their initial neighbors. So, graph will always remain connected. □

Claim 3.2 shows that the graph will remain connected. Thus, Claims 2.1, 2.2, and Claim 2.3 all hold for the LM algorithm (taking into account only the current neighbors of a node, and *not* all the neighbors the node remembers). Hence, the upper bound in Theorem 2.2 also applies to the LM algorithm.

□

4 Linearization with Shortcut Neighbors

In this section we present the linearization with shortcut neighbors (LSN) technique. This technique can be seen as a variation of the LM technique. The main ideas behind the LSN algorithms are that (i) a node v only remembers at most $2(\log n + 1)$ carefully selected shortcut neighbors, which function as representatives of exponentially increasing length intervals out of node v, and (ii) at each round, a node exchanges information about its shortcut neighbors with all of its current or shortcut neighbors at that time. Without loss of generality, we assume that each node has an arbitrary label belonging to the interval $[0, 1)$. Each node v divides $[v, 1)$ into the intervals $I_i^+(v) = [v + 2^{i-1}/n, v + 2^i/n)$, for all $1 \leq i \leq k$ such that $v + 2^k/n < 1$ and $v + 2^{k+1}/n > 1$; and the intervals $I_0^+(v) = [v, v + 1/n)$ and $I_{k+1}^+(v) = [v + 2^k/n,\ 1)$. Each node v divides $[0, v]$ into the intervals $I_i^-(v) = [v - 2^i/n,\ v - 2^{i-1}/n)$, for all $1 \leq i \leq j$ such that $v - 2^j/n > 0$ and $v - 2^{j+1}/n < 0$; and the intervals $I_0^-(v) = [v - 1/n, v]$ and $I_{j+1}^+(v) = [0, v-2^j/n)$. For each of the intervals I_i^+ and I_i^-, node v will remember one shortcut neighbor that belongs to the interval, if such a node exists. Hence v will remember at most $2(\log n + 1)$ shortcut nodes. Those shortcut neighbors that v remembers will aid v in placing its current neighbors within the sorted list, in the same way as the LM algorithm.

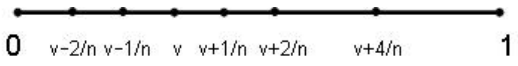


Figure 5: Intervals for node v

Algorithm 3 Linearization with Shortcut Neighbors (LSN)

Each node u executes the following:

1: **for** each round **do**
2: Send your shortcut neighbors to your original and shortcut neighbors
3: Receive shortcut neighbors of your neighbors.
4: Check the shortcut neighbors u just received from its neighbors and the current neighbors u has. If there exists a node v that belongs to an interval $I_i^+(u)$ or $I_i^-(u)$ which was not covered by any of the prior neighbors of node u (i.e., there is no shortcut neighbor for this interval at node u), add v to u's shortcut neighbors list (if there are two or more such nodes, select one of them arbitrarily).
5: Reconnect the current left neighbors of u, as described in Step 3 of the LM algorithm (node u only remembers its current neighbors and shortcut neighbors, at any time).
6: Reconnect the current right neighbors, as described in Step 4 of the LM algorithm.
7: **end for**

The intuition behind the LSN algorithm is our hope that by using geometrically decreasing intervals near each node and collecting neighbors for these, the network topology converges quickly against something close to a hypercubic network. A hypercubic network structure would have the advantage that nearest neighbors can be found much quicker (basically in logarithmic time via binary search) than in arbitrary networks. In fact, it is not difficult to show that linearization on top of a hypercubic network would result in a sorted line within $O(\log n)$ communication rounds. Algorithm 3 describes the LSN algorithm in detail.

The LSN algorithm also performs poorly in the worst-case, since, like the PL and LM algorithm, it is based on the concept of a virtual space (an extremely unevenly distribution of node labels would hurt the algorithm). However, it is not clear what the *exact* worst-case complexity of the LSN algorithm is. The upper bound given by Theorem 2.2 still applies to the LSN algorithm. However, since nodes now exchange their neighbor information with each other, the example used in the proof of Theorem 2.1 no longer gives a lower bound of $n - 2$.

5 Experimental Results

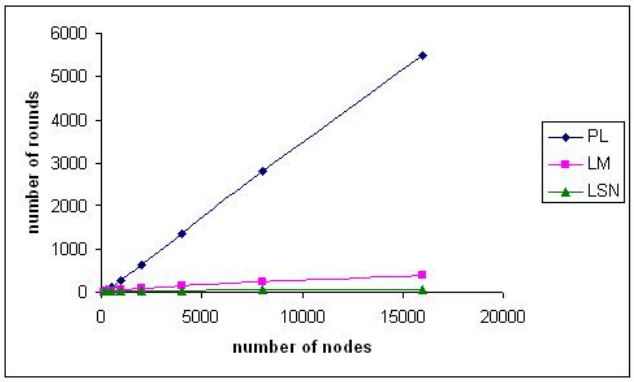
5.1 Regular Graphs All three algorithms, PL, LM, and LSN, are implemented with Java. The runtimes of the algorithms are first compared for connected random graphs such that the expected degree of each node is 10. The random graphs are generated such that the probability of existence of each edge is $10/(n-1)$, where n is number of nodes in the graph. If the generated graph is not connected, a new graph is generated. We run experiments for graphs with up to 16000 nodes. Each experiment is repeated 1000 times. We calculate the average number of required rounds.

	500	1000	2000	4000	8000	16000
PL	122	275.7	634.6	1347	2835	5485.2
LM	26.3	45.41	79.47	135.2	227.7	394.83
LSN	18.5	25.97	35.28	43.93	53.4	65.1

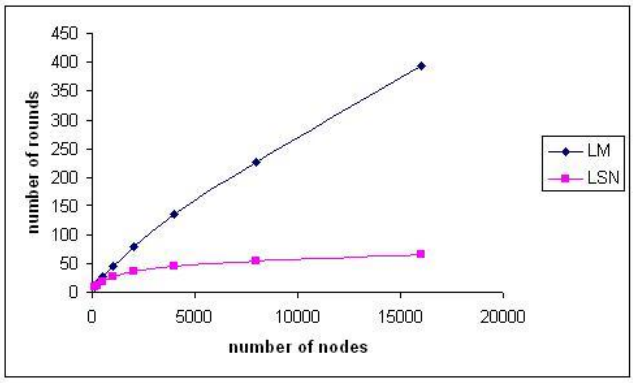
Table 1: Average number of rounds for PL, LM and LSN (for each node avg number of neighbors is 10)

The experimental results show that the LSN algorithm improves over the LM algorithm and both LM and LSN algorithms seem to have a polylogarithmic runtime (Table 1 and Figure 6). The former observation is somewhat surprising when taking into account that the nodes in LM may collect much more edges over time than LSN. However, the fact that LSN uses a structured approach of collecting edges still seems to give it an advantage

Next, for the LM and LSN algorithms the average



(a)



(b)

Figure 6: Average number of rounds for PL, LM and LSN

maximum node degree is computed experimentally (Figure 7 and Table 2). As expected, the LM algorithm has a greater average maximum degree than the LSN algorithm. Still, the experimental results indicate that the maximum node degree might be polylogarithmic for both algorithms.

	50	100	200	400	1000	2000
LM	25.9	35	46.1	58.38	76.8	92.1
LSN	12.9	17.1	22.1	27.15	33.39	37.74

Table 2: Average maximum node degree for LM and LSN (for each node avg number of neighbors is 10)

5.2 Power Law Graphs In recent work, it is claimed that the Internet and certain unstructured P2P network topologies satisfy the power law [11, 12, 13, 14].

DEFINITION 5.1. (POWER LAW) *The number of nodes that have degree k is proportional to $1/k^{\alpha}$, where α is a constant number and $\alpha > 1$.*

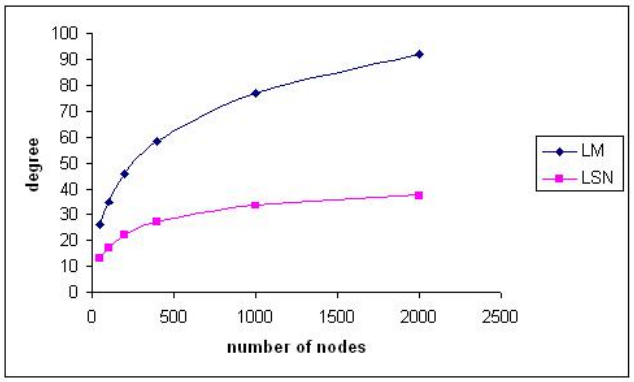


Figure 7: Average maximum node degree for LM and LSN (for each node avg number of neighbors is 10)

We generated random power law graphs according to the Chung-Lu model [15], with $\alpha = 2$ and ran some experiments. To generate these graphs, we first generated the degrees of the nodes according to the power law. Then we considered every node pair $\{i, j\}$ and selected an edge between them with probability $\frac{d_i * d_j}{2m}$ where d_i is the desired degree of node i and m is number of edges in the graph (Chung-Lu model [15]). Thus, we have a graph with node i having expected degree d_i.

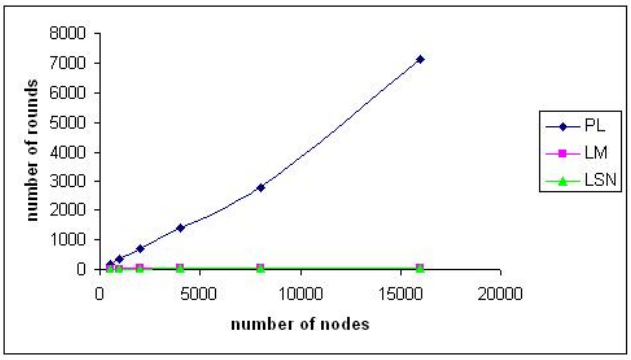
	500	1000	2000	4000	8000	16000
PL	170	339	699.2	1420	2786	7142
LM	13.9	21.62	30.08	39.74	54.15	62.67
LSN	12.1	15.21	18.12	22.97	28.08	38.167

Table 3: Average number of rounds for PL, LM and LSN (graphs satisfy power law with $\alpha = 2$)

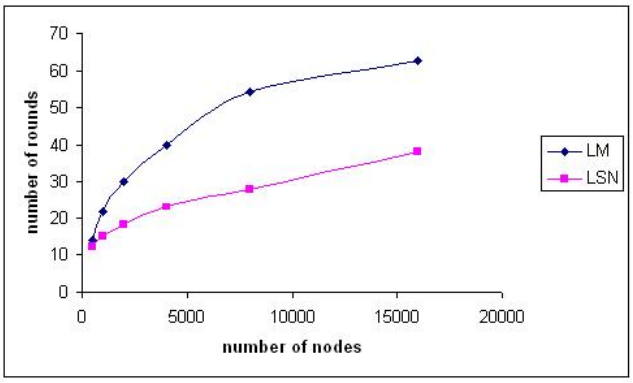
As for regular random graphs, the experimental results indicate that the LM and LSN algorithms seem both to have an expected polylogarithmic runtime for random power law graphs (Table 3 and Figure 8).

5.3 Bounded Locality We generated random graphs with *bounded locality* B, i.e. each edge in the graph crosses at most B nodes. Recall that an edge $\{u, v\}$ crosses a node w if $u < w < v$. We generated random graphs with 1000 nodes and expected node degree of 10. We ran experiments for different values of B, by letting $B = 125, 250, 500, 1000$.

The experimental results also indicate that the expected running time of the PL algorithm increases linearly and that the expected running time of both the LM and LSN algorithms increase polylogarithmicly with B (Table 4, Figure 9).



(a)



(b)

Figure 8: Average number of rounds for PL, LM and LSN (graphs satisfy power law with $\alpha = 2$)

6 Conclusions

We presented distributed local self-stabilizing algorithms for converting an arbitrary graph into a sorted list. In the pure linearization (PL) algorithm, each node repeatedly converts its neighbors into a sorted list. We show that the worst-case running time of the PL algorithm is linear. Hence, we further looked at proper extensions of the PL algorithm, namely, the linearization with memory (LM) and linearization with shortcut neighbors (LSN) algorithms. Both algorithms are based on the idea of remembering old neighbors. Experimental results on random graphs, random power law graphs and random graphs with bounded locality show that the LM and LSN algorithms significantly improve the PL algorithm and that these algorithms seem to have a polylogarithmic time complexity in expectation. Moreover, the LSN algorithm has better average runtime and maximum node degree than the LM algorithm. Ultimately, we would like to provide a rigourous argument proving the expected polylogarithmic behaviour of the LSN and LM algorithms. However, this has proven to be a very challenging task and is left for future work.

	125	250	500	1000
PL	45.18	88.25	181.1	275.7
LM	13.82	22	34.13	45.41
LSN	11.04	15.36	21.55	25.97

Table 4: Average number of rounds for PL, LM and LSN with bounded locality

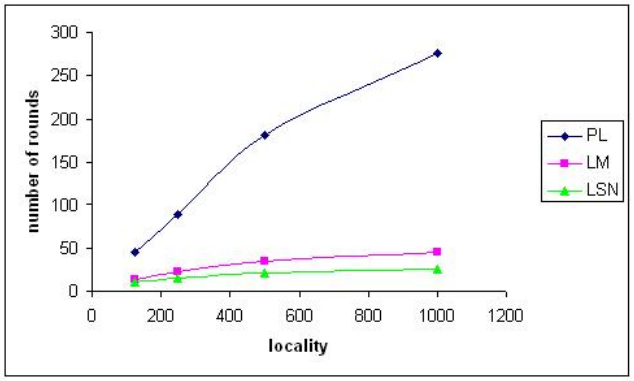


Figure 9: Average number of rounds for PL, LM and LSN with bounded locality

References

[1] D. Angluin, J. Aspnes, J. Chen, Y. Wu and Y. Yin, *Fast construction of overlay networks*, Proc. of the 17^{th} ACM Symp. on Parallel Algorithms and Architectures (SPAA), 2005, pp. 145–154.

[2] J. Aspnes and G. Shah, *Skip graphs*, Proc. of the 14^{th} ACM Symp. on Discrete Algorithms (SODA), 2003, pp. 384–393.

[3] B. Awerbuch and C. Scheideler, *Group Spreading: A protocol for provably secure distributed name service*, Proc. of the 31^{th} International Colloquium on Automata, Languages and Programming (ICALP), 2004, pp. 183–195.

[4] J. Brzezinski and M. Szychowiak, *Self-Stabilization in Distributed Systems - a Short Survey*, Foundations of Computing and Decision Sciences, 25 (1), 2000.

[5] C. Cramer and T. Fuhrmann, *Self-Stabilizing Ring Networks on Connected Graphs*, Technical Report 2005-5, System Architecture Group, University of Karlsruhe, 2005.

[6] E.W. Dijkstra, *Self-stabilization in spite of distributed control*, Communications of the ACM, 17 (1974), pp. 643–644.

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

[8] T. Herman, *Self-Stabilization Bibliography: Access Guide*, University of Iowa, 2002.

[9] A. Shaker and D.S. Reeves, *Self-stabilizing structured ring topology P2P systems*, 5^{th} IEEE Int. Conference on Peer-to-Peer Computing, 2005, pp. 39–46.

[10] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M.F. Kaashoek, F. Dabek and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, Technical Report, MIT, 2002.

[11] Michalis Faloutsos, Petros Faloutsos and Christos Faloutsos, *On power-law relationships of the Internet topology*, SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication, 1999, pp. 251–262.

[12] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins and Janet Wiener, *Graph structure in the web*, 9^{th} Int. WWW Conference, 2000.

[13] Albert-Laszlo Barabasi and Reka Albert, *Emergence of Scaling in Random Networks*, Science, (286) 1999, pp. 509–512.

[14] Matei Ripeanu, Ian Foster and Adriana Iamnitchi, *Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design*, IEEE Internet Computing Journal special issue on peer-to-peer networking, vol. 6(1) 2002.

[15] F. Chung, L. Lu. Connected components in random graphs with given degree sequences. *Annals of Combinatorics* 6 (2002) 125-145.