Abstract We consider the problem of self-healing in peer-to-peer networks that are under repeated attack by an omniscient adversary. We assume that, over a sequence of rounds, an adversary either inserts a node with arbitrary connections or deletes an arbitrary node from the network. The network responds to each such change by quick “repairs,” which consist of adding or deleting a small number of edges. These repairs essentially preserve closeness of nodes after adversarial deletions, without increasing node degrees by too much, in the following sense. At any point in the algorithm, nodes \( v \) and \( w \) whose distance would have been \( \ell \) in the graph formed by considering only the adversarial insertions (not the adversarial deletions), will be at distance at most \( \ell \log n \) in the actual graph, where \( n \) is the total number of vertices seen so far. Similarly, at any point, a node \( v \) whose degree would have been \( d \) in the graph with adversarial insertions only, will have degree at most \( 3d \) in the actual graph.

Our distributed data structure, which we call the Forgiving Graph, has low latency and bandwidth requirements. The Forgiving Graph improves on the Forgiving Tree distributed data structure from Hayes et al. (2008) in the following ways: 1) it ensures low stretch over all pairs of nodes, while the Forgiving Tree only ensures low diameter increase; 2) it handles both node insertions and deletions, while the Forgiving Tree only handles deletions; 3) it requires only a very simple and minimal initialization phase, while the Forgiving Tree initially requires construction of a spanning tree of the network.

Keywords Self-healing · Networks · Peer-to-peer · Stretch · Trees · Dynamic · Responsive

1 Introduction

Many modern networks are reconfigurable, in the sense that the topology of the network can be changed by the nodes in the network. For example, peer-to-peer, wireless and mobile networks are reconfigurable. More generally, many social networks, such as a company’s organizational chart; infrastructure networks, such as an airline’s transportation network; and biological networks, such as the human brain, are also reconfigurable. Reconfigurable networks offer the promise of “self-healing” in the sense that when nodes in the network fail, the remaining nodes can reconfigure their links to overcome this failure. In this paper, we describe a distributed data structure for maintaining invariants in a reconfigurable network. We note that our approach is responsive in the sense that it responds to an attack by changing the network topology. Thus, it is orthogonal and complementary to traditional non-responsive techniques for ensuring network robustness.
Our model

We now describe our model of attack and network response. We assume that the network is initially a connected graph over \( n \) nodes. An adversary repeatedly attacks the network. This adversary knows the network topology and our algorithm, and it has the ability to delete arbitrary nodes from the network or insert a new node in the system which it can connect to any subset of the nodes currently in the system. However, we assume the adversary is constrained in that in any time step it can only delete or insert a single node. The detailed model is described in Sect. 2.

Our results

In this paper, we present a new, distributed data structure called the Forgiving Graph. For a peer-to-peer network that has both insertions and deletions, let \( G' \) be the graph consisting of the original nodes and inserted nodes without any changes due to deletions. Let \( n \) be the number of nodes in \( G' \). The Forgiving Graph ensures that: 1) the distance between any two nodes of the actual network never increases by more than \( \log n \) times their distance in \( G' \); and 2) the degree of any node never increases by more than 3 times its degree in \( G' \). This increase in distance and degree is within a constant factor of optimal, as we show in Sect. 7.2. Our algorithm is completely distributed and resource efficient. Specifically, after deletion, repair takes \( O(\log d \log n) \) time and requires sending \( O(d \log n) \) messages, each of size \( O(d \log n + \log^2 n) \) where \( d \) is the degree of the node that was deleted. The formal statement and proof of these results is in Sect. 7.

This paper builds significantly on results achieved in [10], which presented a responsive, distributed data structure called the Forgiving Tree for maintaining a reconfigurable network in the face of attack. Over a complete run of Forgiving Tree: 1) The diameter of the network can never exceed its original diameter by more than a multiplicative factor of \( O(\log \Delta) \) where \( \Delta \) is the maximum degree in the graph; and 2) the total increase in the degree of any node can never be more than 3.

The improvements of the Forgiving Graph over the Forgiving Tree are threefold. First, the Forgiving Graph maintains low stretch i.e. it ensures that the distance between any pair of nodes \( v \) and \( w \) is close to what their distance would be even if there were no node deletions. Second, the Forgiving Graph handles both adversarial insertions and deletions, while the Forgiving Tree could only handle adversarial deletions (and no type of insertion). Finally, the Forgiving Graph does not require an initialization phase, while the Forgiving Tree required an initialization phase which involved sending \( O(n \log n) \) messages, where \( n \) was the number of nodes initially in the network, and had a latency equal to the initial diameter of the network. Additionally, the Forgiving Graph is divergent technically from the Forgiving Tree; it makes significant use of a novel distributed data structure that we call a Half-full Tree or “haft”.

Related work

There have been numerous papers that discuss strategies for adding additional capacity or rerouting in anticipation of failures [2,5,6,12,18,21,23]. Results that are responsive in some sense include the following. Medard et al. [14] propose constructing redundant components to make backup routes possible when an edge or node is deleted. Anderson et al. [1] modify some existing nodes in a network to be RON (Resilient Overlay Network) nodes to detect failures and reroute accordingly. Some networks have enough redundancy built in so that separate parts of the network can function on their own in case of an attack [7]. In all these past results, the network topology is fixed. In contrast, our approach adds edges to the network as node failures occur. Further, our approach does not dictate routing paths or specifically require redundant components to be placed in the network initially. Our model of attack and repair builds on earlier work in [3,20].

There has also been recent research in the physics community on preventing cascading failures. In the model used for these results, each vertex in the network starts with a fixed capacity. When a vertex is deleted, some of its “load” (typically defined as the number of shortest paths that go through the vertex) is diverted to the remaining vertices. The remaining vertices, in turn, can fail if the extra load exceeds their capacities. Motter, Lai, Holme, and Kim have shown empirically that even a single node deletion can cause a constant fraction of the nodes to fail in a power-law network due to cascading failures [11,17]. Motter and Lai propose a strategy for addressing this problem by intentional removal of certain nodes in the network after a failure begins [16]. Hayashi and Miyazaki [8] propose another strategy, called emergent rewirings, that adds edges to the network after a failure begins to prevent the failure from cascading. Both of these approaches are shown to work well empirically on many networks. However, unfortunately, they perform very poorly under adversarial attack.

An extended abstract of the current paper appeared in PODC 2009 as [9].

Table of contents

The rest of this paper is organized as follows. In Sect. 2, we present our formal problem statement. In Sect. 3, we give a high level overview of the Forgiving Graph algorithm. In Sect. 4, we describe half-full trees (hafts, for short), which are a critical building block in the Forgiving Graph. Section 5 gives details of how the Forgiving Graph can be implemented in a completely distributed fashion, while Sect. 6 describes
the correlation between the forgiving graph and the underlying network. In Sect. 7, we give the statements and proofs of our main theorems. Finally, we conclude and give areas for future work in Sect. 8.

2 Node insert, delete and network repair model

We now describe the details of our node insert, delete and network repair model. Let \( G = G_0 \) be an arbitrary graph on \( n \) nodes, which represents processors in a distributed network. In each step, the adversary either deletes or adds a node. After each deletion, the algorithm gets to add some new edges to the graph, as well as deleting old ones. At each insertion, the processors follow a protocol to update their information. The algorithm’s goal is to ensure that the distance between any pair of nodes does not increase by too much. At the same time, the algorithm wants to minimize the resources spent on this task, especially keeping node degree small.

Initially, each processor only knows its neighbors, and neighbors of neighbors in \( G_0 \), and is unaware of the structure of the rest of \( G_0 \). After each deletion or insertion, only the neighbors of the deleted or inserted vertex are informed that the deletion or insertion has occurred. After this, processors are allowed to communicate by sending a limited number of messages to their direct neighbors. We assume that these messages are always sent and received successfully. The only synchronicity assumption we make is that no other vertex is deleted or inserted until the end of this round of computation and communication has concluded. To make this assumption more reasonable, the per-node communication cost should be very small in \( n \) (e.g. at most logarithmic).

We also allow a certain amount of pre-processing to be done before the first attack occurs. This may, for instance, be used by the processors to gather some topological information about \( G_0 \), or perhaps to coordinate a strategy. Another success metric is the amount of computation and communication needed during this preprocessing round. Our full model is described in Fig. 1.

For our success metrics, at any time \( T \), we compare the actual graph \( G_T \) to the graph composed of the original graph i.e. the graph at \( G_0 \) and only the adversarial insertions (inserted nodes and their incident edges). This is the graph which would have been present if the adversary was not doing any deletions and (thus) no self-healing algorithm was active. This is the natural graph for comparing results. Notice that if there were no insertions happening in our model, we could have compared \( G_T \) to \( G_0 \) but since insertions are happening, \( G_T \) may not even have the same nodes as \( G_0 \) rendering a node-based comparison impossible. Figure 2 shows an example of \( G'_T \) and a corresponding \( G_T \). The figure also shows, in \( G'_T \), the nodes and edges inserted and deleted, and

Each node of \( G_0 \) is a processor.
Each processor starts with a list of its neighbors in \( G_0 \).
Pre-processing: Processors may exchange messages with their neighbors.

\[ \text{for } t := 1 \text{ to } T \text{ do} \]

Adversary deletes a node \( v_i \) from \( G_{t-1} \) or inserts a node \( v_i \) into \( G_{t-1} \) forming \( H_t \).
if node \( v_i \) is inserted then
  \( v_i \) and its new neighbors may update their information and exchange messages with their neighbors.
if node \( v_i \) is deleted then
  All neighbors of \( v_i \) are informed of the deletion.

Recovery phase:
Nodes of \( H_t \) may communicate (asynchronously, in parallel) with their immediate neighbors. These messages are never lost or corrupted, and may contain the names of other vertices.
During this phase, each node may add edges joining it to any other nodes as desired. Nodes may also drop edges from previous rounds if no longer required.
At the end of this phase, we call the graph \( G_t \).

Success metrics: Minimize the following “complexity” measures:
Consider the graph \( G' \) which is the graph consisting solely of the original nodes and insertions without regard to deletions and healings. Graph \( G' \) is \( G \) at timestep \( t \) (i.e. after the \( t \)th insertion or deletion).

1. Degree increase.
\( \max_{v \in G} \frac{\deg(v, G_T)}{\deg(v, G'_T)} \)

2. Network stretch.
\( \max_{x, y \in G} \frac{\text{dist}(x, y, G)}{\text{dist}(x, y, G'_T)} \)
where, for a graph \( G \) and nodes \( x \) and \( y \) in \( G \), \( \text{dist}(x, y, G) \) is the length of the shortest path between \( x \) and \( y \) in \( G \).

3. Communication per node. The maximum number of bits sent by a single node in a single recovery round.

4. Recovery time. The maximum total time for a recovery round, assuming it takes a message no more than 1 time unit to traverse any edge and we have unlimited local computational power at each node.

Fig. 1 The node insert, delete and network repair model—distributed view in \( G_T \), the edges inserted by the healing algorithm, in different colors, as the network evolved over time. Figure 3 shows how the two graphs compare with regards to degree of a particular node \( v \), and Fig. 4 shows how the healing algorithm effects the distance between two nodes, \( u \) and \( v \). Our algorithm guarantees our invariants on the 'complexity' measures at every time step that the algorithms is in execution (Fig. 5).
2.1 Neighbor of neighbor information

As in [10], we assume that nodes maintain neighbor-of-neighbor information. This is necessary for any self-healing algorithm, since otherwise, deletion of a single node can render remaining nodes mutually unreachable.

There are many ways to maintain neighbor of neighbor information, see for example [13, 19]. Maintaining this information impacts resource costs in two ways. First, maintaining neighbor of neighbor information requires additional storage at each node: in a network where all nodes have degree $d$, each node will need to store $O(d^2)$ information about links. Second, maintaining this information requires additional communication. In particular, (1) when a new node enters the network, it must send its list of neighbors to its neighbors; and (2) whenever a node changes its neighbor list, due to the entrance or exit of another node, it must send this information to its neighbors.

In [19], a technique is outlined for piggy-backing this additional communication on the messages that are already sent in the network. In particular, if $u$ and $v$ are neighbors, $u$ periodically checks that $v$ is alive by sending ping messages to it. It is possible for $u$ to include in these ping messages a hash of $u$’s current list of neighbors. If $v$ detects a difference between its own view of $u$’s neighbors and what $u$ sent, it is possible to use a fast and communication efficient protocol for reconciling two sets (see e.g. [15]).

In [19], Naor and Wieder write: “We conclude that implementing Neighbor of Neighbor has very little cost both in communication complexity and in internal running time. It is almost a free tweak that may be implemented on top of the previous constructions.” Because the cost of maintaining neighbor of neighbor information seems incremental, and because the information can be used for other purposes, such as routing, we do not explicitly include this maintenance cost in the analysis of our algorithm.

3 The Forgiving Graph algorithm

Here, we give a high level description of our algorithm. An adversary can effect the network in one of two ways: inserting a new node in the network or deleting an existing node from the network. Node insertion is straightforward and is dependent on the specific policies of the network. When an insertion happens, our incoming node and its neighbors update the data structures that are used by our algorithm.

Each time a node $v$ is deleted, we can think of it as being replaced by a Reconstruction Tree, which is defined as follows:

**Reconstruction tree:** A tree like structure added by the healing algorithm on adverserial deletion of a single node and its edges. The reconstruction tree uses existing nodes (we call them *real nodes*) from the network and also may have virtual nodes i.e. nodes which are simulated by the real nodes in that reconstruction tree.
Reconstruction trees were first used by the authors in ForgivingTree [9]. Notice that we have defined Reconstruction Trees like a template and the exact structure is determined according to the desired properties of the algorithm e.g. in the ForgivingTree [9], a kind of binary balanced tree is used. In this paper, our reconstruction tree is a haft (defined in Sect. 4). For this paper, we will refer to the Reconstruction Tree formed on deletion of a node v as RT(v), and in general as RT. Only real nodes form the leaves of this tree and all the internal nodes are virtual nodes simulated by those leaf nodes. We will redefine RT(v) more formally in Sect. 5.

We will show that each virtual node has a degree of at most 3 (Theorem 1 part 1). A single real node itself is a trivial RT with one node. RT(v) is formed by merging all the neighboring RTs of v using the strip and merge operations from Sect. 4. Thus, following a deletion, we may have a graph with both real and virtual nodes. After a long sequence of such insertions and deletions, this graph is a patchwork mix of virtual nodes and real nodes. Let us call this graph FG (short for ForgivingGraph). As for the other graphs, FG T is the graph FG at time T.

Also, because the hafts are balanced binary trees, the deletion of a node v can, at worst, cause the distances between its neighbors to increase from 2 to $2 \lceil \log d \rceil$ by traveling through its RT, where d is the degree of v in G' (the graph consisting solely of the original nodes and insertions without regard to deletions and healings). However, since this deletion may cause many RTs to merge and the new RT formed may involve all the nodes in the graph, the distances between any pair of actual surviving nodes may increase by no more than a $\lceil \log n \rceil$ factor.

Since our algorithm is only allowed to add edges and not nodes, we cannot really add these virtual nodes to the network. We get around this by assigning each virtual node to an actual node, and adding new edges between actual nodes in order to allow “simulation” of each virtual node. More precisely, our actual graph is the homomorphic image of the graph described above, under a graph homomorphism which fixes the actual nodes in the graph and maps each virtual node to a distinct actual node which is “simulating” it. Figure 16 shows this homomorphism where the graph FG is mapped to the graph G. We discuss this homomorphism and its relationship to our results in more detail in Sect. 7.

We will show (Theorem 1 part 1) that because each actual node simulates at most one virtual node for each of its deleted neighbors, and virtual nodes have degree at most 3, this ensures that the maximum degree increase of our algorithm is at most 3 times the node’s degree in G'.

4 Half-full trees (“HAFTS”)

In this section, we define half-full trees (or hafts, for short), and describe their most important properties for our present application. These trees are similar to the “staircase trees” described by Vaucher [22]. However, our presentation will be self-contained. Hafts also evoke the flavor of other well known data structures such as binomial trees and binomial heaps. A comparison is given at the end of this section.

Half-full tree: A half-full tree, or haft, is a rooted binary tree in which every non-leaf node v has the following properties:

- v has exactly two children.
- The left child of v is the root of a complete binary subtree that contains at least half of v’s descendants.

Primary root: A primary root is a node in a haft such that:

- It is the root of a complete subtree.
- Its parent, if it has one, is not the root of a complete subtree.

Spine: A spine node is the parent of a primary root. Equivalently, it is a node in a haft which is not the root of a complete subtree. The spine of a haft is the set of all spine nodes. We observe that the spine, if non-empty, consists of the vertices of a path, with the root of the haft as one endpoint.

Figure 6a shows several examples of hafts. We now give a simple structural lemma which completely characterizes any haft as a function of the number of its leaves. This will be useful later when we wish to perform merging operations on the hafts used by our algorithm.

**Lemma 1** (Binary representation of Hafts) Let $\ell$ be a positive integer. Then there is a unique haft T having $\ell$ leaves. Moreover, let h be the number of ones in the binary representation of $\ell$, and suppose $x_1 > x_2 > \cdots > x_h$ are the indices of these ones, so that

$$\ell = \sum_{i=1}^{h} 2^{x_i}.$$

Then either

- $h = 1$, and T is a complete tree of depth $x_1$, or
- $h \geq 2$, and T consists of $h - 1$ spine nodes $s_1, \ldots, s_{h - 1}$, together with h complete binary trees $T_1, \ldots, T_h$, where
  - $s_1$ is the root of $T$,
  - each $T_i$ has depth $x_i$,
  - each $s_i$ has the root of $T_i$ as its left child
  - for $1 \leq i \leq h - 2$, $s_i$ has $s_{i+1}$ as its right child
  - $s_{h-1}$ has the root of $T_h$ as its right child
Corollary 1 Let $T$ be a haft having $\ell$ leaves. Then the depth of $T$ equals $\lceil \log \ell \rceil$.

Proof (Proof of Lemma 1) We will prove the detailed structure of $T$, from which the uniqueness follows directly.

First, consider the case $h = 1$ (i.e., $\ell$ is a power of 2). If $\ell = 1$, there is nothing to prove. Assume $\ell > 1$. Now the left subtree of $T$ is complete, and hence has number of leaves equal to a power of two. Since at least half of the leaves are on the left subtree, this power of two is at least $\ell/2$. Since the root of $T$ has two children, not all of the leaves are on the left subtree, and hence there are exactly $\ell/2$ leaves on the left subtree, and thus also $\ell/2$ leaves on the right subtree. Since it is immediate from the definition that any subtree of a haft is also a haft, it follows by induction on $\ell$ (being a power of two) that the right subtree is also a complete subtree. Thus, $T$ is complete.

Now, suppose $h \geq 2$. Let us denote the root of $T$ by $s_1$. Because $\ell$ is not a power of 2, $s_1$ must be a spine node. Since the left subtree, $T_1$, is complete and contains between $\ell/2$ and $\ell$ leaves, it must have depth $x_1$. Since the right subtree is a haft having number of leaves equal to

$$\ell - 2^{x_1} = \sum_{i=2}^{h} 2^{s_i}$$

it follows by induction on $\ell$ (being any positive integer) that it has the claimed structure. Thus, $T$ is also as claimed. \qed

4.1 Operations on hafts

We define the following operations on hafts:

1. Strip: Suppose $T$ is a haft with $h$ ones in its binary representation. The Strip operation removes $h - 1$ nodes from $T$ returning a forest of $h$ complete trees.
2. Merge: The Merge operation joins hafts together using additional isolated single nodes, to create a single new haft.

We now describe these operations in more detail:

4.1.1 Strip

By Lemma 1, if we remove the spine from a haft, we are left with a forest of $h$ complete binary trees, where $h$ is the number of leaves of $T$. The operation Strip($T$) returns this forest.

The Strip operation works as follows: If $T$ is a complete tree, then return $T$ itself. Note that the root of the $T$ is the only primary root in this case. If $T$ is not a complete tree, then $F$ is obtained as follows. Starting from the root of $T$, traverse the direct path towards the rightmost leaf of $T$. Remove a node if it is not a primary root. Stop when a primary root or a leaf node (which is a primary root too) is discovered. In Fig. 6b the Strip operation removes the nodes indicated by the square boxes (Fig. 7).

We now prove why the Strip operation works.

Lemma 2 The Strip operation returns the subtrees rooted at all primary roots in the input haft.

Proof By the definitions of haft and primary root, if a vertex is not the root of a complete subtree, its left child is guaranteed to be a primary root. Thus, either the root of the haft is a primary root or its left child is. If the left child is a primary root, there can be no other primary root in the left subtree, so we return the tree rooted at that child. Recursively applying the same test to the right child, we get all the primary roots. \qed
4.1.2 Merge

By Lemma 1, every haft is completely characterized by its number of leaves. Merging hafts is analogous to binary addition of these numbers. The new binary number obtained is the number of leaves in the haft produced by the Merge operation. This is illustrated in Fig. 8.

The first step of the Merge operation is to apply the Strip operation on the input trees. This gives a forest of complete trees. These complete trees can be recombined with the help of extra nodes to obtain a new haft. Let $\text{Size}(X)$ be the number of nodes in a tree $X$. Consider two complete trees $T_1$ and $T_2$ ($\text{Size}(T_1) > \text{Size}(T_2)$), with roots $r_1$ and $r_2$ respectively, and an extra node $v$. To merge these trees, make $r_1$ the left child and $r_2$ the right child of $v$ by adding edges between them. The merged tree is always a haft. Thus, the merge operation $\text{Merge}(\text{haft}_1, \text{haft}_2, \ldots)$ is as follows:

1. Apply Strip to all the hafts to get a forest of complete trees.
2. Let $T_1, T_2, \ldots, T_k$ be the $k$ complete trees sorted in ascending order of their size. Traverse the list in ascending order; let $T_i$ and $T_{i+1}$ be the first two adjacent trees of the same size and $v$ be a single isolated vertex, join $T_i$ and $T_{i+1}$ by making $v$ the parent of the root of $T_i$ and the root of $T_{i+1}$, to give a new tree. Reinsert this tree in the correct place in the sorted list. Continue traversal of the list from the position of the last merge, joining pairs of trees of equal sizes. At the end of this traversal, we are left with a sorted list of complete trees, all of different sizes.
3. Let $T_1, T_2, \ldots, T_l$ be the sorted list of complete trees obtained after the previous step. Traverse the list in ascending order, joining adjacent trees using single isolated vertices. Let $w$ be a single isolated vertex. Join $T_1$ and $T_2$ by making the root of $T_2$ the left child and the root of $T_1$ the right child of $w$, respectively. This gives a new haft. Join this haft and $T_3$ by using another available isolated vertex, making the larger tree ($T_3$) its left child. Continue this process till there is a single haft.

4.2 Hafts versus binomial heaps

Half-full trees are similar to binomial heaps [4] in the sense that both are mergable structures and their representation and merge have correspondence to binary numbers and binary addition. However, there are clear differences. Binomial heaps satisfy the heap property, which hafts (at least in our application) need not. During merging, two binomial heaps or trees are joined directly by connecting their roots whereas in the merge for hafts, an additional node is needed as a new root for the merging of two hafts. Also, binomial trees/heaps are not binary trees whereas hafts, by definition, are binary trees. For comparison purposes, let us call the number of leaves of a haft as its order since the shape of each haft is determined by its number of leaves. Then, a haft with order $k$ has $2k - 1$ nodes whereas a binomial tree of order $k$ has $2^k$ nodes. A binomial tree has a root node whose children are roots of binomial trees of smaller orders sorted by their order, whereas in a haft the spine (as defined before) has, as children, complete trees (which are a special case of hafts) of smaller orders also arranged in a sorted order.

4.3 Hafts versus complete binary trees

A complete binary tree is a haft where the number of leaves are a power of 2. However, their use for us is limited since we can have structures with any number of leaves. In comparison with a simple complete binary tree, a haft offers the advantage of efficient decentralized merging and splitting. In particular, a haft is constructed in such a way that it can be efficiently decomposed into at most $\log n$ full binary trees. Moreover, given a collection of full binary trees, we can efficiently merge all of them into a single haft.

5 FG: distributed implementation

In this section, we describe the details of how the Forgiving Graph can be implemented in a completely distributed fashion.

As mentioned earlier, deletion of a node $v$ leads to it being replaced by a Reconstruction Tree (RT$(v)$, for short) in $G$. The RT is a haft having “virtual” nodes as internal nodes and real neighbors of $v$ as the leaf nodes. Since we are now familiar with hafts, we formally define RT and RT$(v)$ here:

RT: In the ForgivingGraph, RT is a haft built after the deletion of a node (and its incident edges). The leaves of the haft are real nodes and all the internal nodes are virtual nodes simulated by those leaf nodes.

RT$(v)$: RT$(v)$ is the RT built on deletion of a node $v$. RT$(v)$ is composed of the neighbors of $v$ and of nodes from RTs in which $v$ was a member.
The virtual nodes are called helper nodes. Recall that the graph \( G' \) is the graph consisting of solely the original nodes and insertions (Table 1).

Figure 9 shows a small series of deletions and repairs by the ForgivingGraph algorithm. Notice that after healing on the third deletion some nodes are occurring as leaf nodes multiple times (Fig. 9f). Here, edge information is useful for differentiating between these nodes. A node takes part in a RT only if one of its neighbors got deleted. It can only have \( k \) edges into a RT if \( k \) of its neighbors have already been deleted, for some integer \( k \). Each edge from a real node into a RT corresponds to a deleted neighbor. We can imagine this edge never got deleted and just that its other endpoint got replaced by a helper node. Thus, if there was an edge between nodes \( x \) and \( y \), and node \( y \) got deleted, we can keep this edge labelled as \((x, y)\). Alternatively, the edge is labelled with it’s name in \( G' \), which will always be \((x, y)\) since \( G' \) has no deletions. For convenience, when a node occurs as a leaf node multiple times in a RT, we will often consider each occurrence as a separate node and describe it as such. Figure 10 shows this alternate representation. Notice that it is easy to see the haft structure in this representation and we stay in the realm of trees. From now on, when we refer to a leaf node of a RT, we will mean a real node augmented with the edge information. Thus, when we state that there is at most one helper node corresponding to a leaf node of a RT, this is equivalent to saying that there is at most one helper node in a RT corresponding to an edge in the graph \( G' \).

The actual processor on which we are executing the algorithm must keep track of its real nodes, edges and helper nodes. In Table 1, we list the information each processor \( v \) requires for each of its edges in \( G' \) in order to execute the ForgivingGraph algorithm. For node \( v \), the end point of the edge is stored in the field \( v\.endpoint \). For an edge \((v, x)\), if \( x \) is a real node (i.e. not a helper node) then the field \( v\.endpoint \) is simply the node \( x \). When one of the nodes of the edge gets deleted, a helper node from the new RT may take the place of the previous node. We will still refer to this edge as \((v, x)\) i.e. by its name in \( G' \) but update the fields endpoint and RTparent. Moreover, the processor may now simulate a helper node corresponding to this edge. Since each edge is uniquely identified, the real nodes and helper nodes corresponding to that edge can also be uniquely identified. This identification is used by the processors to pass messages along the correct paths. The Forgiving Graph algorithm is given in pseudocode form in Algorithm 5.2 along with the required subroutines.

At a high level, when a node is deleted, the algorithm for repair is as shown in Algorithm 5.1. The repair proceeds in two parts. The first part is a quick \( O(1) \) phase in which the neighbors of the deleted node connect themselves in the form of a binary tree (Algorithm 5.4, Fig. 11). Consider the effect of the deletion of \( v \) on one of the RTs of which \( v \) is a leaf. Removal of this leaf and of the helper node corresponding to that leaf (if any) splits this RT into connected components. We select particular nodes which were neighbors of the deleted nodes from each of these components. Let \( Nset \) be the collection of all these nodes together with any undeleted neighbors of \( v \) in FG. We shall call a component taking part in the merge process (irrespective of whether it is a haft or not) as a RTfragment, to distinguish it from the...
Fig. 9 Effect of 3 deletions on a graph. The RT for each deleted node consists of the helper nodes, plus the neighbors of the deleted node which form the leaves of the tree. In this example, the deleted nodes form an independent set, so the structure of the RTs does not depend on the deletion order.

**a** The original graph. Node v attacked.

**b** Healed graph. The new nodes inside ellipse are helper nodes.

**c** Node y attacked.

**d** Healed Graph. Notice two RTs with common leaf nodes.

**e** Node w attacked: notice w is a common leaf of both RTs.

**f** Healed Graph. The RTs have merged. Some of the leaf nodes (x, y) are identical (so the picture no longer shows the RT resembling a haft. However, refer Fig. 10)

Fig. 10 Equivalent representations of a RT. **a** From Fig. 9f. Nodes x and u have two edges each going into the haft corresponding to two of their deleted neighbors. **b** Nodes x and u repeated as leaf nodes of RTs with edges corresponding to their deleted neighbors. This shows the haft structure of the RT

**Part 1:**
Deletion of node v splits RTs into RTfragments
1: Anchors (designated nodes) from each RTfragment combine to form a binary tree $BT_v$ [Refer to Algorithm 5.4: DELETE(v)]

**Part 2:**
[Refer to Algorithm 5.5: BOTTOMUPRTMERGE()]
1: Each anchor will initiate 2-phase primary roots discovery [Refer to Algorithm 5.6: TWOPHASEFINDPRoots]
   i) Identify primary roots in their RTfragments (may have false positives)
   ii) Exchange primary roots lists and broadcast correct list through $BT_v$
2: Merge leaves of $BT_v$ with parent node, in parallel, to get $BT_v'$ (anchors may change)
3: Anchors do root discovery (no false positives now), and merge leaves with parent [Refer to Algorithm 5.7: SINGLEPHASEFINDPRoots]
4: Repeat previous step till a single RT left

Algorithm 5.1: ForgivingGraph actions after node deletion (high level pseudocode)

The anchors send probe messages to discover the primary roots which head these complete trees (Algorithms 5.6 and 5.7). This is similar to the Strip operation described in Sect. 4.1.1. The nodes maintain information about their height and number of their children in their RT or RTfragment. Thus, they are able to identify themselves as primary roots.
At the end of the primary roots discovery procedure, the Anchors have the complete and correct list of primary roots involved in the merge. Moreover, each primary root is registered with exactly one anchor.

The complete trees are then merged pairwise in a bottom-up fashion till only a single haft remains. This is illustrated in Fig. 11. At each round, every leaf RT in $BT_v$ will merge with its parent RT. This can be done in parallel, so that the number of rounds of merges will be equivalent to the height of the tree. For two trees to merge, as shown in the Merge operation (Sect. 4.1.2), an additional node is needed that will become the parent of these two trees. This node must be simulated by a real node that is not already simulating a helper node in the tree. Since the number of internal nodes in a tree is one less than the leaf nodes, there is exactly one such leaf node for each tree. The roots of these two trees have the identity of this node for their tree. This node is called a Representative (of the root node). For merging, we use an algorithm that we call the representative mechanism. The formal definition of a representative and details of the representative mechanism are given in Sect. 5.3. Each node keeps the identity of its representative stored in the field Representative (Table 1). The mechanism allows us the following:

**Lemma 4** When RTs merge (Algorithm 5.5), every primary root needs to know the identity of the available helper node that will be its parent after the merge; The representative mechanism allows the node to know this identity.

Now, we briefly describe merging using representatives. When two trees (Note that a tree may even be a single node) are merged (Algorithms 5.10 and 5.11), the representative of the root of the bigger tree (or of one of the trees, if they have the same size) instantiates a new helper node, and makes the two roots its children. To make the new structure a haft, the root of the bigger tree shall become the left child of the new helper node. The new helper node will now inherit as its representative the representative of the root of its right subtree, since this is the node in the merged tree that does not have a helper node. An example of merging using this algorithm is shown in Fig. 12.

At the end of each round, we have a new set of leaf RTs. Each new leaf is now a merged haft of the previous leaves and their parent. We need a new anchor for this haft. We can continue having the anchor of the parent RT as the anchor. However, this node may be one of the extra nodes marked for removal. In this case, the anchor designates one of the nodes that was a primary root in its RT as the new anchor, passes on its links and removes itself. The newly formed leaf hafts may have primary roots which are different from those of the previous ones. The new anchor will send probe messages and gather the relevant information and inform the new primary roots of their role. As mentioned earlier, now the root discoveries can be done in a single phase.
since the undiscovered deletion problem cannot arise now and the nodes will be able to correctly identify themselves as primary roots. This process will continue till we are left with a single RT. Thus, we get the following (using Lemmas 3 and 4):

Lemma 5 On deletion of a node \( v \), the algorithm (Algorithm 5.1, Algorithm 5.4) selfheals the deletion by distributively constructing RT(\( v \)).

Algorithm 5.2: FORGIVING GRAPH: The main function

Algorithm 5.3: INIT(\( v \)): initialization of the node \( v \)

Algorithm 5.4: DELETEFix(\( v \)): Self-healing on deletion of a node

5.2 Two-phase primary roots discovery

The protocol to discover primary roots needs to ensure that all the primary roots involved in the merging are correctly identified and listed with at most one anchor. The later part is ensured by each node forwarding the probe messages of only one anchor. That is, a node will forward only the messages from the first anchor received and send a ‘blocked’ reply back to any other anchor that sends it a probe message. The primary roots are able to correctly identify themselves using Algorithm 5.8: TESTPRIMARYROOT, since they know their height in the RTfragment and also the number of their descendants, allowing them to calculate if they head a complete tree. Since every subtree of a complete tree is another complete tree, the node checks with it’s parent, and if the parent is not heading a complete tree, the node knows it is a primary root. Note that if the probe message comes from an anchor that is in the node’s subtree, the node knows that it is not heading a complete subtree anymore. It does not even need to know its exact number of descendants since the node will be a spine node and hence removed after the present merge. The only situation in which a node can misidentify itself as a primary root is when there is an undiscovered deletion in its subtree. This can only happen when it is contacted by an anchor from outside its subtree and the node is not aware of the deletion in its subtree. Since the communication is asynchronous, this is possible even though the node may have a shorter path to the leaves in its subtree. This is illustrated in Fig. 13 where the probe message from anchor \( a2 \) reaches node \( v \) earlier than that from \( a1 \). Moreover, the node cannot wait for a message from its descendants since it is possible that no deletion has happened in its subtree and it is genuinely a primary root.
The protocol will proceed as follows. When a node $v$ receives a probe request through its children, it will respond as usual, knowing that it is not a primary root and will further forward the probe. If it receives a probe from its parent (say, originating from anchor node $a_2$ in the figure), it will check using its information if it may be heading a complete tree and if it thinks it does, it will identify itself as a primary root, respond back via its parent and not further the probe to its children. Node $a_1$ will add $v$ to its list of primary roots. At some later point, $v$ will receive a probe message from $a_2$ and realise that it had wrongly identified itself. This leads to the possibility that there may be undiscovered primary roots in the other subtree of $v$ (the subtree not having $a_2$). Thus, $v$ will further the probe to that subtree. When $v$ will respond back to $a_2$, it will add itself to the misidentified list. Thus, $a_2$ will now have a list with identified primary roots and misidentified primary root $v$. When all the anchors have had their probes answered, they will have such a list. Moreover, the union of the lists of identified primary roots make a new RT tree. When $v$ may have been reused and unmarked by MakeRT, the hafts mediated by anchors $p$, $\ell$ and $r$.
5.3 Representative mechanism

In this section, we discuss representatives and their use in merging in more detail. Formally, we define a representative as follows:

Representative: In the Forgiving Graph FG, given a node \( y \), the representative of \( y \) is a real node, decided as follows:

- If \( y \) is a real node, then \( y \) itself.
- If \( y \) is a helper node, then the unique leaf node that is a descendant of \( y \) and does not have a helper node in the subtree headed by \( y \).

Recollect that one of our objectives is to maintain an invariant that a real node simulate at most one helper node. Moreover, this has to happen in the dynamic environment of nodes getting deleted, inserted, RTs breaking and merging. The representative mechanism allows us to do this in an efficient manner, as we will show. Intuitively, a representative is a real node who we know is not simulating a helper node yet, and so is available for providing a helper node. Each node in the Forgiving Graph has a representative. Formally, for a node \( y \), if \( y \) is a real node, \( y \) is its own representative. This makes sense since \( y \) is the root of a RT (a single node RT) and not simulating a real node. If node \( y \) is a helper node its representative is the unique leaf node that is \( y \)'s descendant in \( y \)'s subtree that is not simulating a helper node. Notice that there is exactly one such leaf node in any subtree since the number of internal nodes are one less than the number of the leaf nodes, and as a consequence of our invariant, all other leaf nodes are simulating exactly one helper node each in that subtree. Due to the way our merge operations operate, each helper node gets assigned a representative when the helper node is created and moreover it never changes its representative during its lifetime. This is a very useful property as we will see later.

First, we will discuss how representatives are used to merge hafts. The simplest example is shown in Fig. 14: two real nodes (a real node is a singleton haft) merge using their representatives. To recollect, when two hafts merge, a new helper node is needed to become the parent of both. We choose this node to be simulated by the representative of the root of the bigger haft. If the hafts are of the same size, either can be selected. The chosen representative is informed: it instantiates a new helper node and makes the two roots its children. To make the new structure a haft, the root of the bigger tree shall become the left child of this new helper node. The new helper node now needs a representative of its own. The obvious choice is the representative of its right child, since that leaf node still has not supplied a helper node. This is consistent with the definition of a representative (this can be verified for the small
example of Fig. 14). This is the conceptual picture. In the distributed implementation, as described earlier, this communication takes place through the anchors which exchange information among the merging anchors. This information consists of the identity of the primary roots, their height and representative information. Each anchor is then able to run the merge algorithm in its memory, and it directly contacts the nodes with which it has to make edges. If this is a new node it is also provided with the identity of its representative.

What happens when a deletion happens and a RT splits into smaller complete trees? To merge back, we need to find the representatives of the roots of these trees. Should we traverse the subtree of these roots to find the representative? Obviously, this is expensive. Fortunately, the representative mechanism renders this unnecessary. To recall, merging happens using primary roots, which are the roots of complete trees. After a split, we are only left with complete trees. Obviously, complete trees have not had a deletion in their subtree, thus, none of the nodes in these trees need to change their representatives. Since only the nodes of the complete trees will be merging (via their roots) we need only worry about their representatives. This is shown in Fig. 15. As shown in the picture, we can imagine that the representatives of the primary roots are in an 'active' state i.e. they will be used for the upcoming merge, whereas representatives of all internal nodes are in a 'dormant' state meaning though they are not required at the present stage, they may be utilized in the future.

6 Real graph from the forgiving graph

It is easy to see that the Forgiving Graph, FG, maps to the real graph G in a straightforward way: map all the helper nodes to the real nodes simulating them. Figure 16 shows an example. More formally, G is a homomorphic image of FG. Consider two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. In this context, a homomorphism may be defined as follows: A homomorphism is a function $f : V_1 \rightarrow V_2$ such that if undirected edge $[v, w]$ is in $E_1$ (the edge set of $G_1$) this implies that the edge $[f(v), f(w)]$ is in $E_2$. Moreover, we say that $G_2$ is the homomorphic image of $G_1$ under $f$ if the edges of $G_2$ are exactly the images of the edges of $G_1$ under the homomorphism. We know that, in FG, there can be multiple real and helper nodes corresponding to a processor in the network that performs all the functions required of those nodes. Each node is identified by its processor and some additional information. For node $v$ in FG, let $Processor(v)$ be the name of that processor. Also, in the graph G, there is only one node per processor and consider this node to be labelled with the name of that processor. Then, our homomorphism $H : V(FG) \rightarrow V(G)$ is simply $H(v) = Processor(v)$.

Let us make the following observations about homomorphisms which will be useful to us in proving our results (Sect. 7).

Observation 1 For any graph homomorphism $F : G_1 \rightarrow G_2$, for all nodes $u$, $v$ in $V$, $dist_{G_2}(F(u), F(v)) \leq dist_{G_1}(u, v)$ where $dist_G(x, y)$ is the distance between two nodes $x$ and $y$ in a graph $G$.

Observation 2 If the graph $G_2$ is the homomorphic image of graph $G_1$ under a graph homomorphism $F : G_1 \rightarrow G_2$, then for all nodes $v'$ in $G_2$, $deg_{G_2}(v') \leq \sum_{v \in F^{-1}(v')} deg_{G_1}(v)$, where $deg_G(x)$ is the degree of the node $x$ in a graph $G$.
7 Results and proofs

In this section, we present the main results for our algorithm (Sect. 7.1), and a lower bound for our self-healing problem (Sect. 7.2).

7.1 Upper bounds

As earlier, let $G$ be the graph of the network, $FG$ the Forgiving Graph, and $G'$ the graph consisting solely of the original nodes and insertions without regard to deletions and healings. Let $G_T$, $FG_T$ and $G'_T$ be these graphs at time $T$.

Lemma 6 Given the edge $(v, x)$ in $G'_T$,

1. There can be at most one helper node in $FG_T$ corresponding to $(v, x)$.
2. During the Repair after a node deletion, there can be at most two helper nodes corresponding to the edge $(v, x)$. Moreover, one of these could also be an anchor in $BT_v$.

Proof There is only one ‘real’ node in $FG_T$ corresponding to an edge in $G'_T$ (Fig. 9). Let us refer to this node as simply $v$. Moreover, $v$ can only be a leaf node of a RT, and a helper node can only be an internal node.

We prove part 1 by contradiction. Suppose there are two helper nodes in $FG_T$ corresponding to the real node $v$. Let us call these nodes $v'$ and $v''$. The following cases arise:

1. $v'$ and $v''$ belong to different RTs:
   This case is depicted in Fig. 17. We assume that both $v'$ and $v''$ exist but that they are in different RTs. By the representative mechanism, a helper node is created only if the real node that simulates it is the representative of a node (e.g. in line 5.11 in Algorithm 5.11). By definition, the representative of a node is a unique leaf node in the subtree headed by that node in its RT. If both $v'$ and $v''$ exist and belong to different RTs, this implies that node $v$ exists as a leaf node in two different RTs. This is a contradiction.

2. $v'$ and $v''$ belong to the same RT:

   Without loss of generality, assume that $v''.height \geq v'.height$. The following cases arise:

   (a) $v'$ is a node not in the subtree headed by $v''$:
   This case is shown in Fig. 18. We assume that both $v$ and $v''$ exist, and that they are in the same RT but in different subtrees i.e. $v''$ is not an ancestor of $v'$. The proof is similar to that of case 1. The representative mechanism and definition of a representative implies that node $v$ was a representative in two non-intersecting subtrees in the same RT. This implies that node $v$ occurs as a leaf twice in that RT. This is not possible.

   (b) $v'$ is a node in the subtree headed by $v''$:
   This case is shown in Fig. 19. We assume that both $v$ and $v''$ exist, and that they are in the same RT and moreover $v''$ is not an ancestor of $v'$. Note that by the representative mechanism, when two nodes are to be joined, the representative of one of them provides the single node that will be their parent. This new node inherits the other (unused) representative as its representative. The tree gets built up bottom up with available representatives propagating upwards. Thus, node $v'$ will be created before node $v''$. By definition of a representative, neither $v'$ nor any of its ancestors can now have $v$ as a representative since $v$ is now already simulating a helper node. Thus, $v''$ was created without any of its children having $v$ as a representative. However, this is not possible.

Now, we prove part 2. As stated earlier, at each stage of the merge procedure, RTfragments in $BT_v$ will merge with their parent. Suppose that $v'$ is a helper node simulated by
real node $v$, and $v'$ is not part of any complete subtree in such a RTfragment. This means that $v'$ will be marked red and removed when this stage of merge is completed (Refer Fig. 11). Let node $y$ be the root of the complete subtree (i.e. a primary root in that RTfragment) that has $v$ as a leaf node. Node $v'$ is an ancestor of node $y$ since $v'$ cannot be $y$’s descendant. By definition, $y$.Representative $=$ $v$, since $v$ will be the unique leaf node in $y$’s subtree not simulating a helper node in that subtree. When the trees are being merged, $v$ may be asked to create another helper node. Thus, $v$ may have two helper nodes. Also, each RTfragment has exactly one anchor node. This anchor may be $v'$ or another node. Thus, in the repair phase, a real node may simulate at most two helper nodes, and one of these helper nodes may be an anchor. However, node $v'$ will be removed as soon as this stage is completed, and if $v'$ was an anchor, a new anchor is chosen from the existing nodes. Since at the end of the merge, $BT_v$ collapses to leave one RT, the extra helper nodes and the edges from the anchor nodes are not present in $FG_T$, thus, not contradicting part 1. □

**Lemma 7** After each deletion, the repair phase requires the sending of at most $O(d \log n)$ messages, each of length $O(d \log n + \log^2 n)$. Moreover, this can be done in parallel by the neighbors of the deleted node, in time polylog($d, n$).

**Proof** There are mainly two types of messages exchanged by the algorithm. They are the probe messages sent by the FINDPRROOTS() (Algorithms 5.6 and 5.7) within a RT and the messages containing the information about anchors, about the primary roots exchanged by the anchors in $BT_v$, and among the primary roots themselves (Algorithm 5.9: COMPUTEHAFT()). The major messages used by ForgivingGraph are listed in Table 2.

Let $size(BT_v)$ be the number of RTs of $BT_v$. Since a helper node can split a RT into maximum 3 parts, and there can be at most $d$ helper nodes, where $d$ is the degree of the deleted node $v$, $size(BT_v) \leq 3d$. Now, let us calculate the number of messages:

- **Probe messages (Algorithms 5.6 and 5.7):** A probe message is generated by an anchor of a RT. This is similar to the $Strip$ operation (Sect. 4.1.1). The path that the probe message follows is the direct path from the originating node to the rightmost node of the RT. At most 2 messages can be generated for every node on the way. Each node waits for a reply to its message. If it had a neighbor as a primary root, it will hear back from it with the root’s identity. If it had an anchor as a neighbor, it will get an ‘end of path’ message. This node will then reply back to the message it had received from its neighbor on the path from the requesting anchor. Thus, each message generated by the request from the anchor will get a reply back with identities of one or multiple primary roots or end of path messages. By the property of hafts, each node on this path will have a primary root as a neighbor, thus, the longest path a message can take is equal to the diameter of the tree, which is the longest path in the tree.

Let $n$ be the number of nodes and $probemsgs$ be the probe messages sent in a single RT. The length of the longest path is $2 \log n$. Thus,

$$|probemsgs| \leq 2 \cdot 2 \cdot 2 \log n$$

$$\leq 8 \log n$$

- **Exchange of primary roots lists:** When Phase-II of the primary roots discovery (Algorithm 5.5) (BOTTOMUPRTMERGE()) is executed, the nodes convey their misidentified nodes list (mislist) with at most $2(d - 1)$ messages going at most once bottom-up and top-down the tree (Let us call this $Mismsgs$). Also, this phase is invoked only once per deletion. After that, at each merge step, the leaves in $BT_v$ merge with their parents. Let $rtlistmsgs$ be the messages exchanged for every such merge. The anchors of the leaves of $BT_v$ send their primary roots lists to the parent, which in turn can send both its list and the sibling’s list to the child. Thus, $|rtlistmsgs| = 4$. In addition, every anchor will send this list to the primary roots in its RT, generating at most another $\log n$ messages (Let us call this $Atokmsgs$).

**Change of Anchor messages:** After a merge happens, the new RT may have a different Anchor, and this Anchor will be contacted by the previous Anchor with the requisite information (Algorithm 5.9). Let us call such a message $AChangeMsg$. Such a message is only generated once for every merge, for the anchor of the newly constructed RT.

As stated earlier, in the $BT_v$, leaves merge with their parents. The number of such merges before we are left with a single RT is $\lceil size(BT_v) / 2 - 1 \rceil$. Also, at most 3 RTs are involved in each merge. Let $totmessages$ be the total number of messages exchanged. Hence,
The Forgiving Graph

totmessages = [size(BT_v)/2 - 1](3(|probemsgs| + |AroRmsgs|) + |rlistmsgs| + |AChangeMsg()| + |mismsgs|) ≤ (3d/2 - 1)(27 log n + 5) + 2(d - 1) ∈ O(d log n)

In BT_v, leaves and their parents merge. This can be done in parallel such that each time the level of BT_v reduces by one. Within each RT, the time taken for message passing is still bounded by O(log n) assuming constant time to pass a message along an edge. Since there are at most ⌈log(d)⌉ levels, the time taken for passing the messages is O(d log n) i.e. polylog(d, n).

During Phase-II of two phase root discovery, each anchor may discover at most one misidentified primary root, which will be an ancestor in its RTfragment. In the worst case, this list can grow up to a size of d such roots by accumulation up to the root of the BT_v. Thus, the maximum size of this message (with node IDs) can be O(d log n). During actual merging, the biggest message exchanged may have information about the primary roots of up to two RTs. This may be the message sent by a parent RT in BT_v to its children RT. Since there can be at most O(log n) primary roots, the size of messages containing their ID is O(log^{2} n).

We now prove our main result. Recall that G_T is the graph produced after T steps of our algorithm, while G'_T is the graph resulting from the insertions only, with no deletions or repairs.

**Theorem 1** The Algorithm ForgivingGraph has the following properties:

1. **Degree increase**: For any node v in V(G_T), after any number of time steps, T, the degree of v in G_T is at most 3 times the degree of v in G'_T.
2. **Stretch**: For any nodes x, y in V(G_T), after any number of time steps, T, the distance between x and y in G_T is at most log(n) times the distance in G'_T.
3. **Cost**: After each deletion, the repair phase requires the sending of at most O(d log n) messages, each of length O(d log n + log^2 n). Moreover, this can be done in parallel by the neighbors of the deleted node, in time polylog(d, n).

**Proof** Part 1 follow directly by construction of our algorithm. Note that for a real node v in FG_T, any degree increase for v is imposed by the edges of its helper node to hparent(v) and hchildren(v). From Lemma 6 part 1, we know that, in FG_T, node v can play the role of at most one helper node for any of its neighbors in G'_T at any time (i.e. equal to the degree of v in G'_T). The number of children of a helper node are never more than 2, because the reconstruction trees are binary trees. Thus the total degree of v in FG_T is at most 3 times its degree in G'_T. From observation 1 and noting that G_T is a homomorphic image of FG_T, we can see that the degree of v in G_T is at most 3 times its degree in G'_T.

We next show Part 2. We show that the stretch of the Forgiving Graph FG_T is O(D log n), where n is the number of nodes in G_T. The distance between any two nodes x and y cannot increase by more than the factor of the longest path in the largest RT on the path between x and y. Since the number of nodes in FG_T is O(n), this factor is log n at the maximum. Since there is a homomorphism from the graph FG_T to G_T, the result follows directly from observation 2.

The proof of Part 3 follows from Lemma 7. Note that besides the communication of the messages discussed, the other operations can be done in constant time in our algorithm.

7.2 Lower bound

We now present a lower bound that shows that the degree increase and stretch of the Forgiving Graph is within a constant factor of optimal.

**Theorem 2** Let n be a positive integer, α ≥ 3 and β = (logα(n - 1) - 1). Then there exists a graph on n vertices and a vertex deletion such that any way of repairing this deletion under our model must either increase the degree of some node by more than a factor of α, or it must increase the distance between some pair of nodes by at least a factor of β.

**Proof** Let G be a star on n vertices, where x is the root node, and x has an edge with each of the other nodes in the graph. The other nodes (besides x) have a degree of only 1. Let G' be the graph created after the adversary deletes the node x. Consider a breadth first search tree, T, rooted at some arbitrary node y in G'. We know that the self-healing algorithm can increase the degree of each node by at most a factor of α, thus every node in T besides y can have at most α - 1 children. Let h be the height of T. Then we know that 1 + α ∑ h−1 i=0 (α − 1)i ≥ n - 1. This implies that (α)^h+1 ≥ n - 1 for α ≥ 3, or h + 1 ≥ logα(n - 1). Let z be a leaf node in T of largest depth. Then, the distance between y and z in G' is h and the distance between y and z in G is 2. Thus, β ≥ h/2, and 2β ≥ logα(n - 1) - 1, or β ≥ 1/2(logα(n - 1) - 1). This is illustrated in Fig. 20.

8 Conclusion

We have presented a distributed data structure that withstands repeated adversarial node deletions by adding a small number of new edges after each deletion. Our data structure is efficient and ensures two key properties, even in the face of both
Deletion of the central node $v$ of a star leads to an increase in the stretch. Here, the healing algorithm can increase the degree of any node by at most a factor of $\alpha$.

adversarial deletions and adversarial insertions. First, the distance between any pair of nodes never increases by more than a log $n$ multiplicative factor than what the distance would be without the adversarial deletions. Second, the degree of any node never increases by more than a 3 multiplicative factor.

Several open problems remain including the following. Can we design algorithms for less flexible networks such as sensor networks? For example, what if the only edges we can add are those that span a small distance in the original network? Can we extend the concept of self-healing to other objects besides graphs? For example, can we design algorithms to rewire a circuit so that it maintains its functionality even when multiple gates fail?

References

1. Andersen, D., Balakrishnan, H., Kaashoek, F., Morris, R.: Resilient overlay networks. SIGOPS Oper. Syst. Rev. 35(5), 131–145 (2001)
2. Awerbuch, B., Patt-Shamir, B., Peleg, D., Saks, M.: Adapting to asynchronous dynamic networks (extended abstract). In: TOC ’92: Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing, pp. 557–570. ACM, New York (1992)
3. Boman, I., Saia, J., Abdallah, C.T., Schamiloglu, E.: Brief announcement: self-healing algorithms for reconfigurable networks. In: Symposium on Stabilization, Safety, and Security of Distributed Systems(SSS) (2006)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2 edn. McGraw-Hill, New York (2001)
5. Doverspike, R.D., Wilson, B.: Comparison of capacity efficiency of DCS network restoration routing techniques. J. Netw. Syst. Manag. 2(2), 95–123 (1994)
6. Frisano, T.: Optimal spare capacity design for various protection switching methods in ATM networks. In: IEEE International Conference on Communications (ICC 97 Montreal), ‘Towards the Knowledge Millennium’, 1997, vol. 1, pp. 293–298 (1997)
7. Goel, S., Belardo, S., Iwan, L.: A resilient network that can operate under duress: to support communication between government agencies during crisis situations. In: Proceedings of the 37th Hawaii International Conference on System Sciences, 0-7695-2056-1/04:1–11 (2004)
8. Hayashi, Y., Miyazaki, T.: Emergent rewirings for cascades on correlated networks. cond-mat/0503615 (2005)
9. Hayes, T.P., Saia, J., Trehan, A.: The Forging Graph: a distributed data structure for low stretch under adversarial attack. In: PODC ’09: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 121–130. ACM, New York (2009)
10. Hayes, T., Rustagi, N., Saia, J., Trehan, A.: The forgiving tree: a self-healing distributed data structure. In: PODC ’08: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, pp. 203–212. ACM, New York (2008)
11. Holme, P., Kim, B.J.: Vertex overload breakdown in evolving networks. Phy. Rev. E 65, 066109 (2002)
12. Iraschko, R.R., MacGregor, M.H., Grover, W.D.: Optimal capacity placement for path restoration in STM or ATM mesh-survivable networks. IEEE/ACM Trans. Netw. 6(3), 325–336 (1998)
13. Manku, G.S., Naor, M., Wieder, U.: Know thy neighbor’s neighbor: the power of lookahead in randomized p2p networks. In: Proceedings of the 36th ACM Symposium on Theory of Computing (STOC) (2004)
14. Medard, M., Finn, S.G., Barry, R.A.: Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs. IEEE/ACM Trans. Netw. 7(5), 641–652 (1999)
15. Minsky, Y., Trachtenberg, A.: Practical set reconciliation. Technical Report, Boston University, Boston (2002–2003)
16. Motter, A.E.: Cascade control and defense in complex networks. Phys. Rev. Lett. 93, 098701 (2004)
17. Motter, A.E., Lai, Y.-C: Cascade-based attacks on complex networks. Phys. Rev. E 66, 065102 (2002)
18. Murakami, K., Kim, H.S.: Comparative study on restoration schemes of survivable ATM networks. In: INFOCOM (1), pp. 345–352 (1997)
19. Naor, M., Wieder, U.: Know thy neighbor’s neighbor: better routing for skip-graphs and small worlds. In: Proceedings of IPTPS, 2004, pp. 269–277 (2004)
20. Saia, J., Trehan, A.: Picking up the pieces: Self-healing in reconfigurable networks. In: IEEE International Parallel & Distributed Processing Symposium (2008)
21. van Caenegem, B., Wauters, N., Demeester, P.: Spare capacity assignment for different restoration strategies in mesh survivable networks. In: IEEE International Conference on Communications (ICC 97 Montreal), ‘Towards the Knowledge Millennium’, vol. 1, pp. 288–292 (1997)
22. Vaucher, J.G.: Building optimal binary search trees from sorted values in o(n) time. In: Essays in Memory of Ole-Johan Dahl, pp. 376–388 (2004)
23. Xiong, Y., Mason, L.G.: Restoration strategies and spare capacity requirements in self-healing ATM networks. IEEE/ACM Trans. Netw. 7(1), 98–110 (1999)