# On Local Fixing

Michael König and Roger Wattenhofer

Computer Engineering and Networks Laboratory,
ETH Zurich, 8092 Zurich, Switzerland
{mikoenig, wattenhofer}@ethz.ch
Fax: +41 44 63 21035

**Abstract.** In this paper we look at the difficulty of fixing solutions of classic network problems. We study local changes in graphs (edge resp. node insertion resp. deletion), and network problems (e.g. maximal independent set, minimum vertex cover, spanning trees, shortest paths). A change/problem combination is *locally fixable* if an existing solution of a problem can be fixed in constant time in case of a local change in the graph. We analyze a variety of well-studied classic network problems with different characteristics.

**Keywords:** *Local Fixing, Fault Tolerance, Graph Problems, Complexity Classes and Maximal Independent Set.*

## 1 Introduction

Every driver knows about the buying vs. fixing dilemma: Is it worth it to repair the old car, or should one instead rather buy a new model? This dilemma also exists in the context of distributed computing: If a solution to a problem breaks because of a small topology or input change, is it cheaper to fix the solution, or should one rather compute a new solution from scratch? Clearly the answer to this general question depends on many parameters, such as the studied problem, or how broken a solution is, or the measure of cost for fixing and computing.

For the weighted matching problem, Lotker, Patt-Shamir, and Rosen proved that fixing [21] is indeed strictly cheaper than computing [14]. Even more surprisingly, there are also examples where computing is cheaper than fixing. Kutten and Peleg show that fixing a maximal independent set (MIS) is $NP$-complete in a footnote in [17], whereas computing is known to take at most polylogarithmic time [22]. These two examples motivated our quest towards a better understanding of the distributed complexity of fixing vs. computing.

In this paper we freeze two of the many parameters of the problem space. First, we are only interested in whether graph changes can be fixed locally (in constant time). Second, we assume that a solution is pretty much intact, i.e., the broken pieces are small, and well-separated in space or time. The topology changes we are looking at in particular are deletions and insertions of single nodes and edges, as they would happen in a moderately dynamic network. For node changes we further differentiate between nodes with one or more edges.

We believe that this array of changes is a suitable model for typical failures in real networks, where a single node might crash or a single edge could become disconnected. In our analysis we only cover one such change in the entire network, however, it is possible that several such changes happen, as long these changes are either well-separated in space (such that they do not influence each other) or time (such that there is enough time to fix one change before the next happens). We examine a diversity of well-studied classic network problems with different characteristics.

Our main findings are as follows: (i) Many problems that feature a constant or polylogarithmic distributed computing complexity can be fixed locally. However, there is no general rule, as there are exceptions. (ii) Global problems are generally not locally fixable. However, adding or removing leaves (nodes with a single edge) often seems to pose no difficulty. Again, there is no general rule, as there are exceptions. (iii) In addition, we show relations between different types of changes.

In summary, even though fixing is often cheaper than computing, in a mathematical sense the two are orthogonal. An overview of our concrete findings is given in Table 1.

| | Computation | | Local Fixing | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | lower bound | upper bound | $+e$ | $-e$ | $w \to w'$ | $+v_1$ | $-v_1$ | $+v_*$ | $-v_*$ |
| $\Gamma_1$-Count | $\Omega(1)$ | $O(1)$ | ✔ | ✔ | — | ✔ | ✔ | ✔ | ✔ |
| $o(n)$-MDS | $\Omega(1)$ | $O(1)$ [16] | ✗ | ✗ | — | ✗ | ✗ | ✗ | ✗ |
| MIS | $\Omega(\sqrt{\log n})$ [14] | $O(\log n)$ [22] | ✔[17] | ✔[17] | — | ✔[17] | ✔[17] | ✔ | ✔ |
| $O(1)$-MWM | $\Omega(\sqrt{\log n})$ [14] | $O(\log n)$ [21] | ✔ | ✔ | ✔ | ✔ | ✔ | ✔[21] | ✔[21] |
| MM | $\Omega(\sqrt{\log n})$ [14] | $O(\log n)$ [12] | ✔ | ✔ | — | ✔ | ✔ | ✔ | ✔ |
| 2-MVC | $\Omega(\sqrt{\log n})$ [14] | $O(\log n)$ [12] | ✔ | ✔ | — | ✔ | ✔ | ✔ | ✔ |
| $\Gamma_{\log n}$-Count | $\Omega(\log n)$ | $O(\log n)$ | ✗ | ✗ | — | ✗ | ✗ | ✗ | ✗ |
| ST | $\Omega(D)$ | $O(D)$ | ✔ | ✗ | — | ✔ | ✔ | ✔ | ✗ |
| MST | $\Omega(D)$ | $O(D)$ | ✗ | ✗ | ✗ | ✔ | ✔ | ✗ | ✗ |
| SPT | $\Omega(D)$ | $O(D)$ | ✗ | ✗ | ✗ | ✔ | ✔ | ✗ | ✗ |
| Flow | $\Omega(D)$ | $O(D)$ | ✗ | ✗ | ✗ | ✔ | ✔ | ✗ | ✗ |
| Leader | $\Omega(D)$ | $O(D)$ | ✔ | ✔ | — | ✔ | ✔ | ✔ | ✔ |
| Count | $\Omega(D)$ | $O(D)$ | ✔ | ✔ | — | ✗ | ✗ | ✗ | ✗ |

**Table 1.** Overview of our results. On the left side we present the known lower and upper bounds to compute a solution for a given problem; these bounds are in the local model, where message size is not bounded. The problems are subdivided by their distributed complexity classes (local, polylogarithmic and global). On the right hand side we list the cost of fixing each problem/change combination (the shorthands for the changes are explained in Section 2.3). A "✔" entry means that the combination can be fixed locally (in constant time), a "✗" entry means that it is not possible to fix the combination locally, and "—" entries only appear in rows where the problem instance does not have edge weights, i.e., where weight changes are not defined. Note that there is a "✔" *and* a "✗" in every column, in all distributed complexity classes.

## 2 Model

### 2.1 Distributed Computing

We are given a network modeled as a graph $G = (V, E)$, in which the nodes must base their computations and decisions on the knowledge about their local neighborhoods. More precisely, a distributed algorithm needs time $t$ if each node $v \in V$ can decide based on its $t$-hop neighborhood $\Gamma_t(v)$. Nodes decide individually on their outputs without communication. Hence, the output of each node $v$ is a function of $\Gamma_t(v)$.

This *neighborhood model*, first introduced by Linial [19], is related to the classic *message passing* model of distributed computing. In the message passing model, the distributed system is modeled as a communication network, again described by an undirected graph $G = (V, E)$. Each vertex $v \in V$ represents a node (host, device, processor, ...) of the network, and an edge $(u, v) \in E$ is a bidirectional communication channel that connects two nodes.

Initially, nodes have no knowledge about the network graph; they only know their own identifier and potential additional inputs. All nodes wake up simultaneously and computation proceeds in synchronous *rounds*. In each round, every node can send one message to each of its neighbors. A node may send different messages to different neighbors in the same round. Additionally, every node is allowed to perform local computations based on information obtained in messages of previous rounds. Communication is reliable, i.e., every message that is sent during a communication round is correctly received by the end of the round. A message passing algorithm has *time complexity* $t$ if all nodes compute their output in $t$ communication rounds.

If messages may be large, it is well known that the message passing model is equivalent to the neighborhood model, i.e., nodes can compute their output based on their $t$-hop neighborhood if and only if they can compute their output in $t$ rounds of synchronous communication in the message passing model. This common $t$ is known as the distributed time complexity.

Similarly, we can define the time $t$ to fix a change to be either the size of the neighborhood $\Gamma_t(v)$ of a node $v$ that is involved in the fix, or as the number of communication rounds $t$ in a message passing algorithm to fix the change.

Various distributed complexity classes are known for $t$. The most important classes are

– *local* algorithms, where the time $t$ is a constant independent of any parameter of the network, i.e., $t \in \Theta(1)$,
– *polylog* algorithms where the time $t$ is polylogarithmic in the number of nodes $n$, i.e., $t \in \Theta(\text{polylog } n)$, and
– *global* algorithms which need $\Theta(D)$ time, where $D$ is the diameter of the network.

Depending on the application, the boundary between local and polylog [23,27] or the boundary between polylog and global [19] are considered more important. In this paper we deal with all three classes. Regarding the fixing time, we are only interested in strictly local algorithms, i.e., a change must be fixed in constant time, in the $O(1)$-neighborhood. Regarding the computing time, we look at both the polylog and the global class in order to get a broader sense of the fixing vs. computing issue.

## 2.2 Network Problems

The different network problems we discuss are, grouped by complexity class:

- *local*
  - $o(n)$-Minimum Dominating Set
  - Counting the 1-neighborhood
- *polylog*
  - Maximal Independent Set
  - Maximal Matching
  - $O(1)$-Maximum Weighted Matching
  - 2-Minimum Vertex Cover
  - Counting the $\log n$-neighborhood
- *global*
  - Spanning Trees
  - Minimum Spanning Trees
  - Shortest Paths Tree
  - Maximum Flow
  - Leader Election
  - Counting the whole graph

For space reasons we omit the full problem definitions here and ask the interested reader to consult the full version.

## 2.3 Examined Graph Changes

We considered the following graph changes when examining the possibility of local fixing:

- *Edge insertion ($+e$):* adding a previously absent edge to the graph without changing the nodes of the graph.
- *Edge deletion ($-e$):* removing a previously present edge from the graph without changing the nodes of the graph.
- *Edge weight change ($w \to w'$):* changing the weight of an already present edge in the graph without changing the nodes of the graph.
- *1-edge vertex insertion ($+v_1$):* adding a vertex to the graph plus a single edge connecting the new vertex to an existing one.
- *1-edge vertex deletion ($-v_1$):* removing a vertex which is only adjacent to one edge together with its edge from the graph.

– *Vertex insertion ($+v_*$):* adding a vertex to the graph plus any amount of edges connecting the new vertex to existing ones.
– *Vertex deletion ($-v_*$):* removing any vertex and all edges adjacent to it from the graph at once.

For weighted graphs inserted edges may have any positive weights assigned to them. The insertion and deletion of nodes without any edges is trivial for all the problems in question. We assume that after a change occurs all nodes directly adjacent to the change are notified of the exact kind of change that occurred.

Further, we allow treating a node "crash" (i.e., a sudden removal from the communication graph) as if the node gracefully "signed off" (organizing any necessary restructuring of the system prior to the node's departure). For this we let every node whose sudden removal would be critical create a "last will" and deploy it at its immediate neighbors. The last will contains the results a proper sign-off procedure would have had. To compute the last will, the sign-off procedure is simulated beforehand, which we require to be local (i.e., conclude within $O(1)$ rounds). Note that every time a state change in the graph could cause the results of a sign-off to change the respective last will must be computed and distributed anew. However, also note that this procedure does not affect the time complexity of computing or fixing a problem, as we require the computation of the last will to only take $O(1)$ rounds. We require last wills for some local fixability results in Sections 4.4 and 4.10.

**Definition 1 ($P^C$ Notation).** *We write $P^C$ to denote the problem of fixing a solution of the graph problem $P$ after a graph change $C$. For instance, $MIS^{+e}$ denotes the problem of fixing a maximal independent set after an edge insertion.*

## 3 Related Work

Distributed network algorithms have been studied ardently for almost 30 years. One of the most basic problems is the maximal independent set (MIS) problem. It was shown that the distributed computation of an MIS can be done in $O(\log n)$ time [2,22]. Closely related to the MIS problem is the maximal matching problem, as a maximal matching can essentially be computed by computing an MIS on the edges, and as such both algorithms are similar [12]. Since the vertices adjacent to a maximal matching are a 2-approximation for vertex cover, also 2-MVC can be solved in $O(\log n)$ time.

The study of distributed weighted matching is more recent, the first constant approximation in polylogarithmic time was shown less than a decade ago [28]. Later, [21] discovered that some of the steps of the algorithm of [28] can be executed in parallel, improving the distributed time complexity to $O(\log n)$. It was shown by [20] that one can even achieve a $(1+\varepsilon)$-approximation in the same time, using a different method.

Kuhn et al. showed that a polylogarithmic approximation for MVC cannot be solved in less than polylogarithmic time [13,14]. Using reductions, one can immediately prove an $\Omega(\sqrt{\log n})$ lower bound for our problems with polylogarithmic distributed complexity. This lower bound was strengthening the earlier log-star lower bound by Linial [19], showing that all these problems (and some more) are indeed in the polylogarithmic distributed complexity class.

Our tree-based problems are in the global distributed complexity class, as one must send information across the whole network, and as such $\Omega(D)$ is a time lower bound. If message size is not bounded, just gathering all the information at all the nodes, and then computing the solution locally solves all problems in asymptotically optimal $O(D)$ time. Using a simple flooding process, one can compute a spanning tree in $O(D)$ time using small messages only. In the synchronous model, this spanning tree will be a shortest path tree. For the MST problem, it is not possible to get a solution in $O(D)$ time using short messages only [25,9,26]. For flow and other global problems, there are results which also suggest a distributed complexity polynomial in $n$ [26,10]. Our overview table contains the results in the unbounded message size model, also known as the local model.

The subject of our paper is not so much the complexity of distributed *computing*, but rather the complexity of distributed *fixing*. Clearly, faults have played a major role in distributed computing since an early time. In fact, one may argue that distributed fixing was in fact studied even earlier, as early as in the 1970s when Dijkstra introduced the concept of *self-stabilization* [6,7]. In contrast to our work, a self-stabilizing algorithm must survive many failures, not just one, and as such it seems to be a difficult challenge. However, as shown 20 years ago [4,1,5], efficient self-stabilization often boils down to distributed *computation*. As such, surprisingly, computation and self-stabilization are more closely related than computation and fixing. See [8,18] for an overview. More recently, "self-healing" algorithms have gained attention [24,11].

*Dynamic networks* are another area related to our work, in which the graph topology is permanently changing, either because of changing environmental conditions (edge changes in wireless networks), mobility (edge changes because of moving nodes in mobile networks), algorithmic dynamics (edge changes due to algorithmic decisions in overlay networks), or churn (nodes constantly joining or leaving as in peer-to-peer systems). In dynamic networks no node is capable of maintaining up-to-date global information on the network. Instead, nodes have to perform their intended (global or polylogarithmic) task based on locally available information only, i.e., all computation in these systems is inherently local. In the last decade there was a tremendous rise in interest in dynamic networks, see [15] for an overview. This line of work is also more ambitious than ours in the sense that large fractions of the network can change concurrently. On the other hand, we restrict ourselves to constant time solutions.

Regarding fixing vs. computing, a most inspiring prior work is by Kutten and Peleg [17]. For the MIS problem, if $P \neq NP$, they show that fixing can be much harder than computing. For this, they consider a model, in which each node is in one of three states: ('1') in the MIS, ('0') not in the MIS, or ('?') forgot whether or not in the MIS. They then study how long it takes to compute the missing node states. 3SAT can be reduced to this problem in a straightforward way. We briefly describe the construction here, because Kutten and Peleg only mention it in a footnote, and did not bother to describe it in detail. Every clause of a 3SAT instance is represented by a node in state '0'. Every variable is represented by two connected nodes (one for true, one for false), both in state '?'. For each clause, there are 3 edges between the clause node and the variable nodes of the variables in the clause. We conclude that fixing an MIS in their model is $NP$-complete.

In a more relaxed model they consider fixing an MIS where every node knows whether it is in the MIS but may be in a conflicting state, i.e., be in the MIS while having a neighbor in the MIS or not being in the MIS while having no neighbors in the MIS. They present a transformation for MIS algorithms yielding a $O(\log x)$ randomized and a $2^{O(\sqrt{\log x})}$ deterministic fixing algorithm, where $x$ is the number of nodes in conflicting states. Our model has a certain overlap with this model: In case a topology change in our model only puts a constant number of nodes into a conflicting state, their method also offers a local fix.

Another milestone is Chapter 4 of the previously mentioned paper by Lotker, Patt-Shamir, and Rosen [21], where they prove that their technique can be adapted to dynamic graphs. In fact, not only do they introduce our notion of topology changes, but they also show that a single node insertion or deletion with any amount of adjacent edges in a maximum weighted matching solution can indeed be fixed in constant time, keeping a constant approximation ratio. Since this beats the lower bound regarding the computational complexity for this problem, it is a nice example that fixing can be strictly easier than computing.

## 4 Results

An overview of our results can be found in Table 1. For space reasons we will omit some of the lemmas and proofs here and ask the interested reader to consult the full version, which contains proofs for all of the listed results.

In the following we will make use of the two graph classes defined below.

**Definition 2 (Paths, Rings).** *A* path graph *with $n$ vertices is given by $G = (V, E)$:*

$$V = \{0, \ldots, n-1\},$$
$$E = \bigcup_{i=1}^{n-1} (i-1, i) \ .$$

*A* ring graph *additionally has the edge $(0, n-1)$.*

### 4.1 Graph Change Relationships

The different graph changes we are studying are related. The following lemmas summarize some implications that can be made.

**Lemma 1.** *For any graph problem $P$:*

- *If we can fix $P^{+v_*}$ locally, we can also fix $P^{+v_1}$ locally.*
- *If we can fix $P^{-v_*}$ locally, we can also fix $P^{-v_1}$ locally.*

**Lemma 2.** *For any weighted graph problem $P$, if we can fix both $P^{+e}$ and $P^{-e}$ locally, we can also fix $P^{w \to w'}$ locally.*

**Lemma 3.** *For any graph problem $P$, if we can fix both $P^{+v_*}$ and $P^{-v_*}$ locally, we can also fix $P^{+e}$, $P^{-e}$ and $P^{w \to w'}$ locally.*

## 4.2 Vertex Counting

In this section we will discuss the problem of each node knowing the number of nodes in its $r$-neighborhood for different values of $r$.

While very straightforward, $\Gamma_1$-Count is a typical example of a problem which can be computed and also fixed in constant time:

**Lemma 4.** *$\Gamma_1$-Count$^{+e}$, $\Gamma_1$-Count$^{-e}$, $\Gamma_1$-Count$^{+v_1}$, $\Gamma_1$-Count$^{+v_*}$, $\Gamma_1$-Count$^{-v_1}$ and $\Gamma_1$-Count$^{-v_*}$ are local.*

*Proof.* After any change all directly adjacent nodes can simply recompute their count values. This requires $O(1)$ rounds. Any node not adjacent to a change will still have a valid count. The lemma follows.

For $r \in \omega(1)$, i.e., non-constant $r$, counts can generally not be fixed in constant time anymore:

**Lemma 5.** *For any $r \in \omega(1)$: $\Gamma_r$-Count$^{+v_1}$, $\Gamma_r$-Count$^{+v_*}$, $\Gamma_r$-Count$^{-v_1}$ and $\Gamma_r$-Count$^{-v_*}$ are not local.*

*Proof.* Adding or removing a node with any (positive) amount of edges anywhere requires updating the node counts in all nodes up to $r$ hops away. This requires $r \notin O(1)$ rounds. The lemma follows.

**Lemma 6.** *For any $r \in \omega(1)$: $\Gamma_r$-Count$^{+e}$ and $\Gamma_r$-Count$^{-e}$ are not local if $r < D$; $\Gamma_r$-Count$^{+e}$ and $\Gamma_r$-Count$^{-e}$ are local for $r \geq D$.*

*Proof.* Consider a path graph. Removing edge $(0, 1)$ or adding edge $(0, n-1)$ requires updating the node counts in all nodes with indices 0 through $r-1$. This requires $r - 1 \notin O(1)$ rounds. The first part of the lemma follows.

If $r \geq D$ every node is counting all nodes in the graph, since we are not considering graph changes which disconnect the graph. Adding and removing edges would not change any node counts in that case. The second part of the lemma follows.

### 4.3 Minimum Dominating Set

In this section we will discuss the problem of approximating minimum dominating sets. This problem does not allow for any local fixing and was chosen to give an example for this particular phenomenon. We are considering only non-trivial approximations, i.e., within $o(n)$ of the minimum dominating set.

Note that although we can compute an $o(n)$-$MDS$ from scratch in constant time [16], fixing one within $O(1)$ hops of a graph change is an entirely different problem!

**Lemma 7.** *$o(n)$-$MDS^{+v_1}$ is not local.*

*Proof.* First, we will show, that no algorithm can solve $k$-$MDS^{+v_1}$ in any constant number of steps $c$ ("locally"), for any $k$ with $1 \leq k \leq \frac{n+1}{c} - 2$. Let us define:

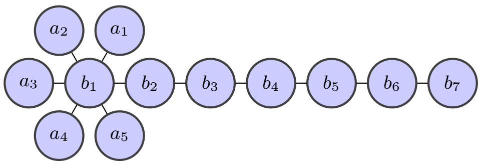

**Fig. 1.** Example graph with x=5 and y=7.

$$\begin{aligned}
x &= \lfloor (k-1)(c+1) \rfloor, \\
y &= 3c+1, \\
G &= (V, E), \\
V &= (a_1, a_2, \ldots, a_x, b_1, b_2, \ldots, b_y), \\
E &= \{(a_i, b_1) \mid 1 \leq i \leq x\} \cup \{(b_i, b_{i+1}) \mid 1 \leq i < y\}, \\
U &= \{a_i \mid 1 \leq i \leq x\} \cup \{b_2\} \cup \{b_{3i} \mid 1 \leq i \leq c\}, \\
U^* &= \{b_{3i+1} \mid 0 \leq i \leq c\} \ .
\end{aligned}$$

See fig. 1 for an example of G. Note that $U$ is a $k$-MDS and $U^*$ is a 1-MDS with respect to $G$. Adding a new vertex $v$ and a new edge $(b_y, v)$ to $G$ will now invalidate $U$ as a dominating set, while $U^*$ still is a 1-MDS. While it is possible to create a new dominating set $U'$ from $U$ by fixing it locally, i.e., only within $c$ hops of vertex $b_y$, at least one additional vertex will have to be added: $|U'| \geq |U|+1$. Since $U^*$ stayed the same, this entails an increased approximation factor $k'$ for $U'$:

$$k' = \frac{|U'|}{|U^*|} = \frac{x+c+2}{c+1} = \frac{\lfloor (k-1)(c+1) \rfloor + c + 2}{c+1} > \frac{(k-1)(c+1)+c+1}{c+1} = k$$

Since $k' > k$, $k$-$MDS^{+v_1}$ is not local for $1 \leq k \leq \frac{n+1}{c} - 2$, and hence $o(n)$-$MDS^{+v_1}$ is not local.

The proofs for the non-locality of $o(n)$-$MDS^{-v_1}$, $o(n)$-$MDS^{+e}$ and $o(n)$-$MDS^{-e}$ are analogous.

## 4.4 Maximal Independent Set

In this section we will discuss fixing maximal independent sets. Kutten and Peleg [17] already showed that $MIS^{+e}$, $MIS^{-e}$, $MIS^{+v_1}$ and $MIS^{-v_1}$ can be fixed in constant time by running a transformed MIS algorithm. We will nevertheless still provide a set of simple proofs for those graph changes. Additionally, we will show that $MIS^{+v_*}$ and $MIS^{-v_*}$ are locally fixable as well.

We say a vertex is *covered* if it is part of the MIS or has a neighbor in the MIS. Note that all vertices in a graph being covered is a sufficient condition for an MIS to be maximal.

We assume that every MIS node knows its 2-hop-neighborhood and is made aware of changes to it (this can be achieved by flooding a message for 2 hops each time a change occurs). This is necessary to allow each MIS node to compute a last will (see Section 2.3), which contains which of its neighbors should enter the MIS in case of a "crash" to ensure retaining a valid MIS.

To compute its last will an MIS node computes the subset of its direct neighbors which are only covered by itself and then computes an MIS on the subgraph of only these neighbors and the edges between them. The nodes of the subgraph's MIS are then chosen to become MIS nodes of the actual graph should the node the last will is for fail. Note that this computation does not require any further messages to be exchanged. Hence, updating the last wills only adds $O(1)$ time to the fixing procedures for each graph change.

Below we will detail the actions which need to be taken in the cases of edge addition and removal of a node with any number of edges. The actions to be taken for the other graph changes are trivial and can be found the in the full version.

**Lemma 8.** *$MIS^{+e}$ is local.*

*Proof.* The MIS can be fixed by doing the following: when an edge $e = (v, u)$ is added and both $v \in MIS$ and $u \in MIS$, pick one of $v$ and $u$ (for instance, whichever has the lower identifier), remove it from the MIS and add those nodes to the MIS which are designated in its last will.

If $v \notin MIS$ or $u \notin MIS$ the MIS remains valid: both nodes directly affected by the change are still covered by either being in the MIS themselves or having a neighbor in the MIS (we know this because we had a valid MIS prior to the edge insertion), and independence is still warranted since not both nodes are in the MIS.

**Lemma 9.** *$MIS^{-v_*}$ is local.*

*Proof.* The case where the removed node is not part of the MIS is trivial – the remaining MIS on the remaining nodes is still valid. In the following we will consider the other case.

Without the node performing a "sign-off" (i.e., participating in the fixing before actually leaving) or an adequate preparation (such as a last will) it is not possible to salvage the MIS in constant time. To see this just imagine an arbitrarily complex graph where one node is connected to every other node. If an MIS is formed by that node alone, its unprepared removal would require computing a new MIS on the whole remaining graph which is known to take at least $\Omega(\sqrt{\log n})$ time.

Luckily, we stated that every MIS node deposits a last will at each of its neighbors stating which nodes should enter the MIS. This way every node can decide in constant time whether it should join the MIS.

## 4.5 Maximal Matching

Maximal matchings can be fixed locally as well. Two individual graph changes are discussed below.

We say an edge $a$ *blocks* another edge $b$ with respect to a matching $M$ if the edges share a vertex and $a \in M$ and $b \notin M$. A matching being maximal is equivalent to every edge either being part of the matching or being blocked.

**Lemma 10.** *$MM^{-e}$ is local.*

*Proof.* An edge being removed potentially allows for two edges to be added in turn: one at each of the vertices of the edge. Both vertices can identify and choose an unblocked edge adjacent to them to join the matching in constant time, which restores maximality.

**Lemma 11.** *$MM^{+v_*}$ is local.*

*Proof.* Of the new edges at most one can become part of the matching, because they all share a vertex. No existing edge can become part of the matching through this change or the matching would not have been maximal before the change. Therefore, by picking any of the new edges which are not blocked (if there are any) and adding the picked edge to the matching, we can obtain a valid maximal matching again.

## 4.6 Spanning Trees

In this section we will discuss spanning trees which do not necessarily have minimum weight. We will not consider graph changes which cause the graph to become disconnected.

**Lemma 12.** *$ST^{-e}$ and $ST^{-v_*}$ are not local.*

*Proof.* Consider a spanning tree on a ring graph: it consists of all the graph's edges except for one at some vertex $i$. Removing vertex $(i + \lfloor \frac{n}{2} \rfloor) \bmod n$, or an edge adjacent to it, requires the edge at vertex $i$ to be added to the spanning tree. For this to happen messages must be sent across up to $\lfloor \frac{n}{2} \rfloor \in \Omega(n)$ links. Hence, $ST^{-e}$ and $ST^{-v_*}$ cannot be fixed locally.

## 4.7 Minimum Spanning Trees

In this section we will discuss minimum spanning trees. We will not consider graph changes which cause the graph to become disconnected.

**Lemma 13.** *$MST^{-e}$ and $MST^{-v_*}$ are not local.*

The proof for Lemma 13 follows that of Lemma 12.

**Lemma 14.** *$MST^{w \to w'}$ is not local.*

*Proof.* Consider a minimum spanning tree on a ring graph where every edge has weight 1: it consists of all the graph's edges except for one at some vertex $i$. Increasing the weight of an edge adjacent to vertex $(i + \lfloor \frac{n}{2} \rfloor) \bmod n$ by any amount requires the edge at vertex $i$ to be added to the minimum spanning tree. For this to happen messages must be sent across up to $\lfloor \frac{n}{2} \rfloor \in \Omega(n)$ links. Hence, $MST^{w \to w'}$ cannot be fixed locally.

**Lemma 15.** *$MST^{+e}$ and $MST^{+v_*}$ are not local.*

*Proof.* Consider a minimum spanning tree on a path graph where every edge has weight 1 except for the edge between vertices $\lfloor \frac{n}{2} \rfloor$ and $\lfloor \frac{n}{2} \rfloor + 1$ which has weight 2: it consists of all the graph's edges. Adding an edge between vertices 0 and $n-1$ with weight 1, or adding a vertex with two edges of weight 1 to vertices 0 and $n-1$ of the original graph, requires the edge with weight 2 to be removed from the minimum spanning tree. For this to happen messages must be sent across up to $\lfloor \frac{n}{2} \rfloor \in \Omega(n)$ links. Hence, $MST^{+e}$ and $MST^{+v_*}$ cannot be fixed locally.

## 4.8 Shortest Paths Trees

In this section we will discuss shortest paths trees. We will not consider graph changes which cause the graph to become disconnected or which remove the root of the SPT.

**Lemma 16.** *$SPT^{-e}$ and $SPT^{-v_*}$ are not local.*

The proof for Lemma 16 follows that of Lemma 12. Which node the SPT is rooted in is irrelevant for this proof.

**Lemma 17.** *$SPT^{w \to w'}$ is not local.*

*Proof.* Consider a SPT rooted in node 0 on a ring graph where every edge has weight 1: it consists of all the graph's edges except for one adjacent to node $\lfloor \frac{n}{2} \rfloor$. Increasing the weight of the edge $(0, 1)$ to $n$ requires the missing edge to be inserted into the spanning tree replacing edge $(0, 1)$. For this to happen messages must be sent across up to $\lfloor \frac{n}{2} \rfloor \in \Omega(n)$ links. Hence, $SPT^{w \to w'}$ cannot be fixed locally.

### 4.9 Maximum Flow

In this section we will discuss maximum flows. We will not consider graph changes which cause source and sink to become to become parts of different graph components or which remove source or sink.

**Lemma 18.** *$Flow^{w \to w'}$, $Flow^{-e}$ and $Flow^{-v_*}$ are not local.*

*Proof.* Consider a ring graph where all edge weights are 1 and which has an additional vertex $v_{source}$ which is only attached to vertex $\lfloor \frac{n}{2} \rfloor$ over an edge with weight 1. Let $v_{source}$ be the flow's source and vertex 0 be the flow's sink. All maximum flows on this graph have a strength of 1 and are divided into two parts which travel over vertices $\{0, 1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ and over vertices $\{0, \lfloor \frac{n}{2} \rfloor, \ldots, n-1\}$ respectively.

Decreasing the weight of either edge $(0,1)$ or edge $(n-1, 0)$ below the strength of the part of the flow on that respective side will require the flow across all edges to be changed (save for the edge adjacent to $v_{source}$). The same may be caused by removing or removing the adjacent non-sink vertex of either edge $(0,1)$ or edge $(n-1, 0)$. Hence, $Flow^{w \to w'}$, $Flow^{-e}$ and $Flow^{-v_*}$ cannot be fixed locally.

**Lemma 19.** *$Flow^{+e}$ and $Flow^{+v_*}$ are not local.*

*Proof.* Consider a path graph where all edge weights are 1. Let vertex 0 be the source of the flow and let vertex $\lfloor \frac{n}{2} \rfloor$ be the sink of the flow. Any maximum flow only uses the edges $\{(a-1, a) \mid 0 < a \leq \lfloor \frac{n}{2} \rfloor\}$.

Adding an edge between vertices 0 and $n-1$, or adding a vertex with two edges of weight 1 to vertices 0 and $n-1$ of the original graph, requires any maximum flow on the resulting graph to use *all* edges. Hence, $Flow^{+e}$ and $Flow^{+v_*}$ cannot be fixed locally.

### 4.10 Leader Election

In this section we will discuss the problem of fixing a leader election. Note that we do not require any node but the leader itself to know who the leader is. The sole requirement is that there is exactly one leader at any time. We will not consider graph changes which cause the graph to become disconnected.

This problem is particularly interesting, because computing it initially takes $\Omega(D)$ rounds [3], while fixing requires little to no effort and can always be done in constant time.

**Lemma 20.** *$Leader^{+e}$, $Leader^{-e}$, $Leader^{+v_1}$, $Leader^{-v_1}$, $Leader^{+v_*}$ and $Leader^{-v_*}$ are local.*

*Proof.* In all cases where merely edges or non-leader nodes get added or removed, we do not need to change the leader node. This takes constant time.

To cover cases in which the leader node gets deleted, we will make use of the "last will" technique again (see Section 2.3). The leader node has at all times exactly one last will deployed at one of its neighbors, stating that that node should become the leader should the leader node be deleted. However, the node should not become a leader if merely the edge to the leader is deleted; in that case it should scrap the last will. Should the last will node be deleted or should its edge to the leader node be deleted, the leader node will issue a new last will. These operations ensure that there is always exactly one leader after any graph change and also take constant time.

## References

1. Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In *Proc. of Distributed Algorithms, 4th International Workshop, WDAG '90, Bari, Italy*, pages 15–28, September 1990.
2. Noga Alon, Laszlo Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.
3. Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 230–240. ACM, 1987.
4. Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Proc. of 29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 206–219. IEEE, 1988.
5. Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. of 32nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 258–267. IEEE, 1991.
6. Edsger W. Dijkstra. Self-stabilization in spite of distributed control. Manuscript EWD391, October 1973.
7. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
8. Shlomi Dolev. *Self-stabilization*. The MIT press, 2000.
9. Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proc. of the 36th ACM Symposium on Theory of Computing (STOC), Chicago, USA*, pages 331–340, 2004.
10. Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proc. of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1150–1162. SIAM, 2012.
11. Thomas P Hayes, Jared Saia, and Amitabh Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing*, 25(4):261–278, 2012.
12. Amos Israeli and Alon Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.
13. Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In *Proc. of the 23rd ACM Symposium on the Principles of Distributed Computing (PODC)*, pages 300–309. ACM, 2004.
14. Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *CoRR*, abs/1011.5470, 2010.
15. Fabian Kuhn and Rotem Oshman. Dynamic networks: Models and algorithms. *ACM SIGACT News*, 42(1):82–96, March 2011.

16. Fabian Kuhn and Roger Wattenhofer. Constant-Time Distributed Dominating Set Approximation. In *Springer Journal for Distributed Computing, Volume 17, Number 4*, May 2005.
17. Shay Kutten and David Peleg. Tight fault locality. In *Proc. of 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 704–713. IEEE, 1995.
18. Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: Self-stabilization on speed. In *Proc. of 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Lyon, France*, November 2009.
19. Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.
20. Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved distributed approximate matching. In *Proc. of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 129–136. ACM, 2008.
21. Zvi Lotker, Boaz Patt-Shamir, and Adi Rosen. Distributed approximate matching. In *Proc. of the 26th annual ACM symposium on Principles of distributed computing (PODC), Portland, Oregon, USA*, pages 167–174. ACM, 2007.
22. Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.
23. Moni Naor and Larry Stockmeyer. What can be computed locally? In *Proc. of the 25th annual ACM symposium on Theory of Computing (STOC), San Diego, California, USA*, pages 184–193. ACM, 1993.
24. Gopal Pandurangan and Amitabh Trehan. Xheal: localized self-healing using expanders. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 301–310. ACM, 2011.
25. David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM Journal on Computing*, 30(5):1427–1442, 2001.
26. A.D. Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed verification and hardness of distributed approximation. *Arxiv preprint arXiv:1011.3049*, 2010.
27. Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys*, 2011.
28. Mirjam Wattenhofer and Roger Wattenhofer. Distributed weighted matching. In *Proc. of the 18th Annual Conference on Distributed Computing (DISC), Amsterdam, Netherlands*, 2004.