

# On Distributed Verification

Amos Korman and Shay Kutten

Faculty of IE&M, Technion, Haifa 32000, Israel

**Abstract.** This paper describes the invited talk given at the 8th International Conference on Distributed Computing and Networking (ICDCN 2006), at the Indian Institute of Technology Guwahati, India. This talk was intended to give a partial survey and to motivate further studies of distributed verification. To serve the purpose of motivating, we allow ourselves to speculate freely on the potential impact of such research.

In the context of sequential computing, it is common to assume that the task of verifying a property of an object may be much easier than computing it (consider, for example, solving an NP-Complete problem versus verifying a witness). Extrapolating from the impact the separation of these two notions (computing and verifying) had in the context of sequential computing, the separation may prove to have a profound impact on the field of distributed computing as well. In addition, in the context of distributed computing, the motivation for the separation seems even stronger than in the centralized sequential case.

In this paper we explain some motivations for specific definitions, survey some very related notions and their motivations in the literature, survey some examples for problems and solutions, and mention some additional general results such as general algorithmic methods and general lower bounds. Since this paper is mostly intended to “give a taste” rather than be a comprehensive survey, we apologize to authors of additional related papers that we did not mention or detailed.

## 1 Introduction

This paper addresses the problem of locally verifying global properties. This task complements the task of locally computing global functions. Since many functions cannot be computed locally [29, 42, 41], local verification may prove more useful than local computing - one can compute globally and verify locally.

In terms of sequential time, there exists evidence that verification is sometimes easier than computation. For example, verifying that a given color assignment on a given graph is a legal 3 coloring is believed to consume much less time than computing a 3 coloring [36]. As another example, given a weighted graph together with a tree that spans it, it is required to decide whether this tree is an MST of the graph. This *MST verification problem* was introduced by Tarjan in the sequential model. A linear time algorithm for computing an MST is known only in certain cases, or by a randomized algorithm [35, 37]. On the other hand, the sequential verification algorithm of [34] is (edge) linear.

In the context of distributed tasks, other measures of complexity are used, e.g., communication complexity. Still, one can ask a similar natural question. Given

a *distributed representation* of a solution for a problem (for example, each node holds a pointer to one of its incident edges), we are required to verify the legality of the represented solution (in the example, to verify that collection of pointed edges forms an MST). Does the verification consume fewer communication bits than the computation of the solution (e.g., the MST)?

Since faults are much more likely to occur in a distributed setting than in a sequential one, the motivation for verification in a distributed setting seems to be even stronger than in a sequential one. A common application of local distributed verification is in the context of self stabilization. See, for example, the *local detection* [31], or the *local checking* [9], or the *silent stabilization* [32]. Self stabilization deals with algorithms that must cope with faults that are rather severe, though of a type that does occur in reality [27, 28]. The faults may cause the states of different nodes to be inconsistent with each other. For example, the collection of pointed edges may not be a tree, or may not be an MST. Self stabilizing algorithm thus often use distributed verification repeatedly. If the verification fails, a (much heavier) global MST recomputation algorithm is invoked. An efficient verification algorithm thus saves repeatedly in communication. We discuss the use application of distributed verification to self stabilization in more length in Section 4.

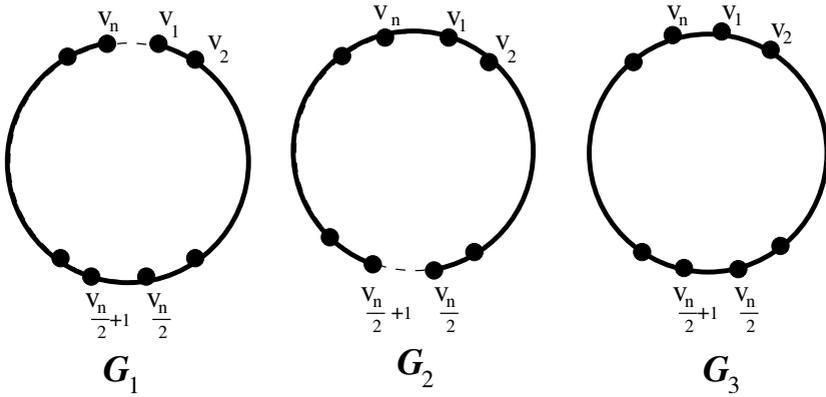
In the *simple model* for local verification, all nodes are awakened simultaneously and start a computation. In a *t-local verification algorithm*, it is required that the represented solution is illegal iff after at most  $t$  time rounds, at least one processor outputs 0 (the rest may output 1). Since we want the locality parameter  $t$  to be independent of the network, it would be desired to have  $t$  be a constant.

Note, that for a constant  $t$  (even for  $t = 1$ ), many representations can be trivially verified. For example, in the legal-coloring verification task, each node just checks that each of its neighbors has a different color than its own. As another example, in a distributed representation of a Minimal Independent Set (MIS), each node holds a flag indicating whether it belongs to the MIS or not. Clearly, such an MIS representation can be verified in one time round.

In a distributed representation of a subgraph of  $G$ , each node may point at some of its incident edges. The set of pointed edges forms a subgraph of  $G$ . In the *spanning tree* (respectively, *MST*) verification problem, it is required to check whether this subgraph is a spanning tree (resp., MST) of  $G$  or not. The following simple claim indicates that in a too simple model for local verification, the verifications of some basic representations require  $\Omega(n)$  time rounds. (We do not describe the simple model explicitly).

**Claim 1.** *In the simple model for local verification, both the spanning tree and the MST verification problems require  $\Omega(n)$  time rounds.*

**Sketch of Proof:** We show the result for the spanning tree case. Let  $G = \{v_1, v_2, \dots, v_n\}$  be a circle. For simplicity of presentation, we assume  $n$  is even. Consider three distributed representations of  $G$  as depicted in Figure 1. In the first representation,  $G_1$ , for each  $1 \leq i \leq n - 1$ , node  $v_i$  holds a pointer to



**Fig. 1.** The three representations of subgraphs in  $G$ . The thick edges correspond to the pointed edges and the dashed edges correspond to the non-pointed edges.

edge  $(v_i, v_{i+1})$ . Therefore, the pointed edges in  $G_1$  are all the edges except for  $(v_n, v_1)$ . In the second representation,  $G_2$ , for each  $1 \leq i \leq n/2 - 1$  and each  $n/2 + 1 \leq i \leq n$ , node  $v_i$  holds a pointer to the edge  $(v_i, v_{i+1}) \pmod{n + 1}$ . Therefore, the pointed edges are all the edges except for  $(v_{n/2}, v_{n/2+1})$ . Note that in both  $G_1$  and  $G_2$ , the pointed edges form a spanning tree. In  $G_3$ , for each  $1 \leq i \leq n$ , node  $v_i$  holds a pointer to the edge  $(v_i, v_{i+1})$ . Therefore, the set of pointed edges consists of all edges in the circle.

First note that since the pointed edges in  $G_1$  and  $G_2$  form a spanning tree, no node in either  $G_1$  or  $G_2$  outputs 0. Assume by contradiction that the spanning tree verification can be accomplished in  $t$  time rounds, where  $t < n/4$ . In this case, a node can only gather information about the nodes at distance at most  $t$  from it. Therefore, for every  $1 \leq i \leq n/4$  and every  $3n/4 \leq i \leq n$ , the output of  $v_i$  in  $G_3$  is the same as the output of  $v_i$  in  $G_2$ . Similarly, for each  $n/2 \leq i \leq 3n/4$ , the output of  $v_i$  in  $G_3$  is the same as the output of  $v_i$  in  $G_1$ . It follows that the output of each vertex in  $G_3$  is not 0, contradicting the fact that the pointed edges in  $G_3$  do not form a tree.  $\square$

In order to deal with verification tasks such as verifying spanning trees, the concept of *proof labeling schemes* was introduced in [40]. The formal definitions are given in Section 2. Informally, it is assumed that the *state* of every node has already been computed by some algorithm (in the above example, the state may consist of a pointer to an incident edge). The configuration (formed as the collection of states of all nodes) is supposed to satisfy some predicate (e.g., “the pointed edges form an MST of the underlying graph”). To enable local verification, labels are added to the nodes in preprocessing stage. To perform the verification, a node computes some *local* predicate, considering only its own state, as well as the above mentioned labels of its neighbors but not their states (!). The global configuration predicate is implied by the conjunction of the local predicates in the following manner. If the configuration is legal then each node

outputs 1 (i.e., “I do not detect a problem”). However, if the configuration is not legal, then for *every* possible way of labeling the vertices, at least one node should detect a problem, i.e. output 0. This, in a way, means that if the configuration is not legal, the adversary cannot fool the verifier by changing the labels. The restriction of one time round can obviously be generalized to  $t$  time rounds (hopefully  $t$  being constant). However, all the results that have been previously established in the area of proof labeling schemes hold for the case  $t = 1$ .

Note, that there is some resemblance between the definition of proof labeling schemes and the notion of NP. Informally, the collection of assigned labels in the preprocessing stage can be considered a witness. If the configuration is legal then there exists a witness (labeling assignment) such that the legality of the configuration can be verified in one time round. Otherwise, if the configuration is not legal then there does not exist such a witness, i.e., for any labeling assignment, in one time round, at least one node should detect a problem.

We note that the number of bits in a label is the number of information bits a node needs to convey to its neighbors in the verification. Ideally, this number is as small as possible, even smaller than the state of the vertex. We evaluate a proof labeling scheme by its *label size*, i.e., the maximum number of bits assigned to a node of the graph in the preprocessing stage.

## 2 Model ([40])

We consider distributed systems that are represented by connected graphs. The vertices of the graph  $G = \langle V, E \rangle$  correspond to the nodes in the system, and we use the words “vertex” and “node” interchangeably. The edges of  $G$  correspond to the links, and we use the words “link” and “edge” interchangeably. Denote  $n = |V|$ . Every node  $v$  has internal ports, each corresponding to one of the edges attached to  $v$ . The ports are numbered from 1 to  $\text{deg}(v)$  (the degree of  $v$ ) by an internal numbering known only to node  $v$ . If  $G$  is undirected, then for every vertex  $v$  let  $N(v)$  denote the set of edges adjacent to  $v$ . If  $G$  is directed, then for any vertex  $v$  let  $N(v)$  denote the set of edges incoming to  $v$ . In either case, for every vertex  $v$  let  $n(v) = |N(v)|$ . Unless mentioned otherwise, all graphs considered are undirected. For two vertices  $u$  and  $v$  in  $G$ , let  $d_G(u, v)$  denote the unweighted distance between  $u$  and  $v$ .

Given a vertex  $v$ , let  $s_v$  denote the state of  $v$  and let  $v_s = (v, s_v)$ . A *configuration graph* corresponding to a graph  $G = \langle V, E \rangle$  is a graph  $G_s = \langle V_s, E_s \rangle$ , where  $V_s = \{v_s \mid v \in V\}$  and  $(v_s, u_s) \in E_s$  iff  $(v, u) \in E$ . A *family of configuration graphs*  $\mathcal{F}_s$  corresponding to graph family  $\mathcal{F}$  consists of configuration graphs  $G_s \in \mathcal{F}_s$  for each  $G \in \mathcal{F}$ . Let  $\mathcal{F}_S$  be the largest possible such family when every state  $s$  is taken from a given set  $S$ . Unless mentioned otherwise, let  $S$  denote the set of integers. We sometimes refer to each state  $s_v$  of a configuration graph as having two fields:  $s_v = (\text{id}(v), s'(v))$ . Field  $\text{id}(v)$  is  $v$ 's *identity* and is encoded using  $O(\log n)$  bits. When the context is clear we may refer to  $s'(v)$  as the state of  $v$  (instead of to  $s(v)$ ). A configuration graph  $G_s$  is *id-based* if for every pair of vertices  $v$  and  $u$  it is given that  $\text{id}(u) \neq \text{id}(v)$ . A graph whose

identities are arbitrary (including possibly the case where all identities are the same) is termed *anonymous*. An id-based (respectively, anonymous) family is a family of id-based (respectively, anonymous) graphs. Let  $\mathcal{F}^{all}$  be the collection of all directed strongly-connected and all undirected connected graphs with  $O(n)$  vertices. Let  $\mathcal{F}^{undirected}$  be the collection of all undirected connected graphs with  $O(n)$  vertices. When it is clear from the context, we use the term “graph” instead of “configuration graph”, “id-based graph” or “anonymous graph”. We may also use the notation  $v$  instead of  $v_s$ . Given a family of configuration graphs  $\mathcal{F}_s$ , let  $\mathcal{F}_s(W)$  denote the family of all graphs in  $\mathcal{F}_s$  such that, when considered as weighted, the (integral) weight of each edge is bounded from above by  $W$ .

Many of the results in [40] deal with a distributed representation of subgraphs. Such a representation is encoded in the collection of the nodes’ states. There can be many such representations. For simplicity, we focus on the case that an edge is included in the subgraph if it is explicitly pointed at by the state of an endpoint. That is, given a configuration graph  $G_s$ , the subgraph (respectively, directed subgraph) induced by the states of  $G_s$ , denoted  $H(G_s)$  (respectively,  $D(G_s)$ ), is defined as follows. For every vertex  $v \in G$ , if  $s_v$  includes an encoding of one of  $v$ ’s ports pointing to a vertex  $u$ , then the edge (respectively, directed edge)  $(v, u)$  is an edge in the subgraph. These are the only edges in the subgraph.

Consider a graph  $G$ . A distributed problem  $Prob$  is the task of selecting a state  $s_v$  for each vertex  $v$ , such that  $G_s$  satisfies a given predicate  $f_{Prob}$ . This induces the problem  $Prob$  on a graph family  $\mathcal{F}$  in the natural way. We say that  $f_{Prob}$  is the *characteristic function* of  $Prob$  over  $\mathcal{F}$ .

This model tries to capture adding labels to configuration graphs in order to maintain a (locally checkable) distributed proof that the given configuration graph satisfies a given predicate  $f_{Prob}$ . Informally, a proof labeling scheme includes a *marker* algorithm  $M$  that generates a label for every node, and a *decoder* algorithm that compares labels of neighboring nodes. If a configuration graph satisfies  $f_{Prob}$ , then the decoder at every two neighboring nodes declares their labels (produced by marker  $M$ ) “consistent” with each other. However, if the configuration graph does *not* satisfy  $f_{Prob}$ , then for *any possible* marker, the decoder must declare “inconsistencies” between some neighboring nodes in the labels produced by the marker. It is not required that the marker be distributed. However, the decoder is distributed and *local*, i.e., every node can check only the labels of its neighbors (and its own label and state).

More formally, A *marker* algorithm  $L$  is an algorithm that given a graph  $G_s \in \mathcal{F}_s$ , assigns a label  $L(v_s)$  to each vertex  $v_s \in G_s$ . For a marker algorithm  $L$  and a vertex  $v_s \in G_s$ , let  $N'_L(v)$  be a set of  $n(v)$  fields, one per neighbor. Each field  $e = (v, u)$  in  $N'_L(v)$ , corresponding to edge  $e \in N(v)$ , contains the following. (1) The port number of  $e$  in  $v$ ; (2) the weight of  $e$  (if  $G$  is unweighted we regard each edge as having weight 1); (3)  $L(u)$ .

Let  $N_L(v) = \langle (s_v, L(v)), N'_L(v) \rangle$ . Informally,  $N'_L(v)$  contains the labels given to all of  $v$ ’s neighbors along with the port number and the weights of the edges connecting  $v$  to them.  $N_L(v)$  contains also  $v$ ’s state and label. A *decoder* algorithm  $\mathcal{D}$  is an algorithm which is applied separately at each vertex  $v \in G$ .

When  $\mathcal{D}$  is applied at vertex  $v$ , its input is  $N_L(v)$  and its output,  $\mathcal{D}(v, L)$ , is boolean.

A *proof labeling scheme*  $\pi = \langle \mathcal{M}, \mathcal{D} \rangle$  for some family  $\mathcal{F}_s$  and some characteristic function  $f$  is composed of a *marker* algorithm  $\mathcal{M}$  and a *decoder* algorithm  $\mathcal{D}$ , such that the following two properties hold:

1. For every  $G_s \in \mathcal{F}_s$ , if  $f(G_s) = 1$  then  $\mathcal{D}(v, \mathcal{M}) = 1$  for every vertex  $v \in G$ .
2. For every  $G_s \in \mathcal{F}_s$ , if  $f(G_s) = 0$  then for every marker algorithm  $L$  there exists a vertex  $v \in G$  so that  $\mathcal{D}(v, L) = 0$ .

We note that all the proof labeling schemes constructed so far use a polytime decoder algorithm. The *size* of a proof labeling scheme  $\pi = \langle \mathcal{M}, \mathcal{D} \rangle$  is the maximum number of bits in the label  $\mathcal{M}(v_s)$  over all  $v_s \in G_s$  and all  $G_s \in \mathcal{F}_s$ . For a family  $\mathcal{F}_s$  and a function  $f$ , we say that the *proof size* of  $\mathcal{F}_s$  and  $f$  is the smallest size of any proof labeling scheme for  $\mathcal{F}_s$  and  $f$ .

### 3 Basic Examples

To illustrate the definitions, we now present a basic proof labeling scheme [40] concerning agreement among all vertices. Note that  $v$ 's neighbors cannot 'see' the state of  $v$  but they can see  $v$ 's label. This is different than what is assumed e.g. in [31]. We note that the following lemma also demonstrates a connection between the notion of proof labeling scheme and that of communication complexity [43].

**Lemma 2.** [40] *The proof size of  $\mathcal{F}_S^{all}$  and  $f_{Agreement}$  is  $\Theta(m)$ .*

*Proof.* We first describe a trivial proof labeling scheme  $\pi = \langle \mathcal{M}, \mathcal{D} \rangle$  of the desired size  $m$ . Given  $G_s$  such that  $f_{Agreement}(G_s) = 1$ , for every vertex  $v$ , let  $\mathcal{M}(v) = s_v$ . I.e., we just copy the state of node  $v$  into its label. Then,  $\mathcal{D}(v, L)$  simply verifies that  $L(v) = s_v$  and that  $L(u) = L(v)$  for every neighbor  $u$  of node  $v$ . It is clear that  $\pi$  is a correct proof labeling scheme for  $\mathcal{F}_S^{all}$  and  $f_{Agreement}$  of size  $m$ . We now show that the above bound is tight up to a multiplicative constant factor even assuming that  $\mathcal{F}_S^{all}$  is id-based. Consider the connected graph  $G$  with two vertices  $v$  and  $u$ . Assume, by way of contradiction, that there is a proof labeling scheme  $\pi = \langle \mathcal{M}, \mathcal{D} \rangle$  for  $\mathcal{F}_S^{all}$  and  $f_{Agreement}$  of size less than  $m/2$ . For  $i \in S$ , let  $G_s^i$  be  $G$  modified so that both  $u$  and  $v$  have state  $s(u) = s(v) = i$ . Obviously,  $f_{Agreement}(G_s^i) = 1$  for every  $i$ . For a vertex  $x$ , let  $\mathcal{M}^i(x)$  be the label given to  $x$  by marker  $\mathcal{M}$  applied on  $G_s^i$ . Let  $L^i = (\mathcal{M}^i(v), \mathcal{M}^i(u))$ . Since the number of bits in  $L^i$  is assumed to be less than  $m$ , there exist  $i, j \in S$  such that  $i < j$  and  $L^i = L^j$ . Let  $G_s$  be  $G$  modified so that  $s_u = i$  and  $s_v = j$ . Let  $L$  be the marker algorithm for  $G_s$  in which  $L(u) = \mathcal{M}^i(u)$  and  $L(v) = \mathcal{M}^j(v)$ . Then for each vertex  $x$ ,  $\mathcal{D}(x, L) = 1$ , contradicting the fact that  $f(G_s) = 0$ .  $\square$

Note, that the corresponding computation task, that of assigning every node the same state, requires only states of size 1.

By the above lemma, it is clear that for any  $m$  there exists a family  $\mathcal{F}_s$  and a function  $f$  with proof size  $\Theta(m)$ . A somewhat stronger claim is presented in [40], namely, that a similar result exists also for *graph problems* (that is, problems where the input is only the graph topology).

**Corollary 3.** *For every function  $1 \leq m < n^2$ , there exists a graph problem on an id-based family with proof size  $\Theta(m)$ .*

Let us now show a family with a smaller proof size. The following example concerns the representation of various spanning trees in the system. The upper bound employs structures and ideas used in many papers including [3, 5, 31, 14, 4, 40]. The lower bound is taken from [40]. A lower bound in the different model of silent stabilization for one of the tasks below was presented in [32]. Consider five different problems, obtained by assigning states to the nodes of  $G$  so that  $H(G_s)$  (respectively,  $D(G_s)$ ) is a (respectively, directed) (1) forest; (2) spanning forest; (3) tree; (4) spanning tree; (5) BFS tree of  $G$  (for some root vertex  $r$ ). Let  $f_{No-cycles}$  (respectively,  $f'_{No-cycles}$ ) be the characteristic function of either one of the five problems above.

**Lemma 4.** [40] *The proof size of  $\mathcal{F}_S^{all}$  and  $f_{No-cycles}$  (respectively,  $f'_{No-cycles}$ ) is  $\Theta(\log n)$ .*

*Proof.* For proving the upper bound, construct the proof labeling scheme  $\pi_{span} = \langle \mathcal{M}_{span}, \mathcal{D}_{span} \rangle$  for  $\mathcal{F}_S$  and  $f$  being “ $H(G_s)$  is a spanning tree”. The other cases are constructed in a similar manner. Given  $G_s$  so that  $f(G_s) = 1$ , the marker algorithm  $\mathcal{M}_{span}$  operates as follows. If  $H = H(G_s)$  is a spanning tree, then it has  $n - 1$  edges. Therefore, either there is only one vertex  $r$  in  $G_s$  whose state is not an encoding of one of its port numbers or there exist exactly two vertices whose states point at each other. In the second case let  $r$  be the vertex with the smaller identity among the two and in both cases  $r$  is considered as the root. Note that the state of each non-root vertex points at its parent in the rooted tree  $(H, r)$ . Let  $\mathcal{M}_{span}(v) = \langle id(r), d_H(v, r) \rangle$ . For a vertex  $v_s$  and a marker algorithm  $L$ , the first field  $L(v)$  is denoted by  $L_1(v)$  and the second by  $L_2(v)$ . The decoder  $\mathcal{D}_{span}(v, L) = 1$  iff all the following easy to verify events occur.

1. For every neighbor  $u$  of  $v$ ,  $L_1(u) = L_1(v) \in S$ . I.e., all vertices agree on the identity of the root.
2. If  $id(v) = L_1(v)$  then  $L_2(v) = 0$ .
3. If  $id(v)$  is not  $L_1(v)$  then  $s_v$  is an encoding of a port number of  $v$  leading to a vertex  $u$  such that  $L_2(v) = 1 + L_2(u)$ .
4. If  $L_2(v) = 0$  then either  $s_v$  is not an encoding of a port of  $v$  or an encoding of a port of  $v$  leading to vertex  $u$  and  $L_2(u) = 1$ .

Obviously, the size of  $\pi_{span}$  is  $O(\log n)$  so we only need to prove that the scheme is correct. Given  $G_s$  so that  $f(G_s) = 1$ . We show that  $\mathcal{D}_{span}(v, L) = 1$  for all  $u, v \in V$ . The first fields of  $\mathcal{M}_{span}(u)$  and  $\mathcal{M}_{span}(v)$  are the same since they are both the identity of the root  $r$ . If  $v \neq r$  then  $s_v$  is the identity of  $v$ 's parent in the tree  $H$ , therefore  $dist_H(v, r) = 1 + dist_H(s_v, r)$ . Also, (2) above holds for  $r$ . Hence,  $\mathcal{D}(v, \mathcal{M}_{span}) = 1$  for each vertex  $v \in G$ .

If, for some marker algorithm  $L$ ,  $\mathcal{D}(v, L) = 1$  for every vertex  $v$ , then by (1), all vertices must agree in the first field of their label. Denote this value  $x$ . Since the identities of the vertices are disjoint, there can be at most one vertex  $r$

satisfying  $id(r) = x$ . Also, by (3), such a vertex must exist. By (3), for every vertex  $u$  such that  $id(u) \neq x$  corresponds a directed edge leading to some vertex  $w$  and  $L(u) - 1 = L(w)$ . Therefore all directed paths must reach the special vertex  $r$  (satisfying  $id(r) = x$ ). Therefore the edges corresponding to all vertices but  $r$ , form a spanning tree  $T$  and the only case to be inspected is whether the edge that correspond to  $r$  (if this edge exists), belongs to this tree. This is verified by (4). The upper bound for the case of a spanning tree follows.

In the case were  $f$  (respectively,  $f'$ ) is a “(respectively, directed) BFS tree”, the decoder  $\mathcal{D}(v, L)$  also checks that  $|L_2(u) - L_2(v)| \leq 1$  for each (respectively, directed) neighbor  $u$  of vertex  $v$ .

**Remark:** a similar approach applies also to BFS trees on weighted id-based graphs except that the size of the scheme changes to  $O(\log n + \log W)$ . Note that in the above schemes if the decoder satisfies  $\mathcal{D}(v, L) = 1$  for every  $v$  then  $L_2(v) = d_G(v, r)$ . Therefore, using this scheme we can also prove that each vertex holds its distance to the root.

Let us next prove the lower bound (the proof is essentially the same for all five problems). Let  $P$  be the horizontal path of  $n$  vertices. For the sake of analysis only, enumerate the vertices of  $P$  from left to right, i.e.,  $P = (1, 2, \dots, n)$ . For  $i < n$ , let  $s_i$  be the port number of the edge leading from vertex  $i$  to  $i + 1$ . Obviously,  $f(P_s) = 1$  and  $f'(P_s) = 1$ . Assume, by way of contradiction, that there exists a proof labeling scheme  $\pi = \langle \mathcal{M}, \mathcal{D} \rangle$  for  $\mathcal{F}_s$  and either  $f$  or  $f'$  which is of size less than  $\log(n/2) - 2$ . Let  $L(i)$  be the label given by  $\mathcal{M}$  to vertex  $i$  in the above path  $P_s$ . Since the number of bits in each  $L(i)$  is less than  $\log(n/2) - 2$ , there exist two pairs of vertices  $(i, i + 1)$  and  $(j, j + 1)$  where  $1 < i$  and  $i + 1 < j < n - 1$  so that  $L(i) = L(j) = L'$  and  $L(i + 1) = L(j + 1) = L''$ . We now build the following ring  $R$  consisting of  $j - i$  vertices whose identities are clockwise ordered from  $i$  to  $j - 1$ . For  $i \leq k < j - 1$  let  $s_k$  be the port number of vertex  $k$  leading from  $k$  to  $k + 1$  and let  $s_{j-1}$  be the port leading from  $j - 1$  to  $i$ . Let us give  $R_s$  the same labeling  $L$  as  $\mathcal{M}$  gives  $P_s$ , i.e., each vertex  $i \leq k < j$  in  $R_s$  is labeled  $L(k)$ . By the correctness of  $\pi$  on  $P_s$  we get that for each vertex  $v \in R_s$ ,  $\mathcal{D}(v, L) = 1$ . This is a contradiction to the fact that  $f(R_s) = 0$  and  $f'(R_s) = 0$ .

Note that the proof applies to all the cases in the lemma, including the case that a (not necessarily spanning) subgraph does not have a cycle.  $\square$

Note that the above lemma implies a lower bound of  $\Omega(\log n)$  for proof labeling schemes for the Minimum Spanning Tree problem (MST). A proof in the spirit of the proof of lemma 2 was then used in [40] to increase this lower bound to  $\Omega(\log n + \log W)$  where  $W$  is the maximum weight of an edge in the graph. This lower bound was later increased in [38] to  $\Omega((\log n \log W))$ . The proof of the lower bound in [38] is quite involved. It uses a new combinatorial structure termed  $(h, \mu)$ -hypertrees that is a combination between  $(h, \mu)$ -trees and a hypercube. That is, an  $(h, \mu)$ -hypertree is constructed by connecting (via a weighted path) every node in one  $(h - 1, \mu)$ -hypertree to the corresponding node in another  $(h - 1, \mu)$ -hypertree. This doubling of the hypertree is partially responsible for the logarithmic behavior of the lower bound. The intuition behind this construction

is that (1) the proof needed a structure with many cycles; and (2) the proof needed to make many nodes neighbors, since proof labeling schemes deal only with neighboring nodes. In the construction,  $h$  is the height of the hypertree and  $\mu$  is the weight of some edges that are crucial for the MST. That proof follows the general structure of [39] in the sense that labels for some  $(h-1, \mu^2)$ -hypertree  $H'$  are computed using the labels for some  $(h, \mu)$ -hypertree  $H$ . However, the specifics are more complex and require some new tricks. For example, the verifier described in the construction for the lower bound, at any node  $v$ , has to *guess* labels for some other nodes.

Two general approaches to constructing proof labeling schemes are presented in [40]. One is a modular construction of a scheme from modules that are other schemes. The other is a simulation of the execution of a distributed algorithm that computes the function to be verified. The second method bears some similarity to the idea of the roll back compiler of [9], that is described briefly in Section 4. This method is used in [40] together with ad hoc improvements to derive an upper bound of  $O(\log^2 n + \log n \log W)$  for the MST problem. This was improved later in [38] to match their improved lower bound.

Additional upper and lower bounds given in [40] for a number of graph problems, including many basic building block problems. Other results therein demonstrated the role and the cost of identities in this model. It was also shown that every predicate has a proof labeling scheme in id-based families of graphs.

## 4 Self Stabilization: An Application of Distributed Verification

In this section we mention the notion of self stabilizing algorithms. It turns out that distributed verification, in addition to its theoretical interest, can be very useful for the design of such algorithms.

The notion of self stabilization was suggested by Dijkstra in 1974 ([10], see also [12]). Dijkstra's paper later won the ACM-PODC influential paper award, that shortly after that became the Dijkstra Prize in Distributed Computing awarded by the ACM (the Association for Computing Machinery) at the Annual Symposium on the Principles of Distributed Computing (PODC). Starting in 2007, this prize will be given by the ACM and EATCS (the European Association for Theoretical Computer Science). It took some years until the importance of that paper became evident, as highlighted first by Lamport [17]. However, since then, a lot of attention has been invested in self stabilization, and this sub-area now even has its own conference (SSS).

In the above mentioned paper, Dijkstra studied the example of a token ring. This is a network of processors arranged in a circle, where each processor can "see" the whole state of one processor that immediately precedes it (e.g. in a clockwise order). The state of the processor (and of the preceding one) may imply that the processor "has a token", or that it "does not have a token". It is required that exactly one of the processors in the whole ring is in the state of "having a token" at any given time. A second requirement was that each node

“passes the token” eventually to the processor succeeding it on the ring. When this action was taken, the passing processor no longer had a token, while the successor started to have one. Thus, the token circulates the ring.

This example was based on a commercial network where if two processors “had a token” their actions could have collided, while if no processor had a token the network could deadlock. Hence, if either more or less than one processor has a token, the network is in an illegal global state (configuration). The designers of the commercial network assumed that it could sometimes reach an illegal state because of either an incorrect initialization, or some equipment error, or bug, etc. (It was proven by [27] that in actual network, even simple and rather common faults may drive protocols into an arbitrary global state.) Hence, the commercial products had a mechanism to recover from an illegal state. This mechanism was based on a timing assumption- one processor serving as a leader (a “station”) waited for a certain time (“timeout”) to receive the token from each predecessor. If the token is not received, then it is assumed lost, and the leader generates a new token. A similar method is used to destroy a redundant token.

In some sense, the commercial solution involved a *global* verification. That is, the length of the timeout had to be large enough so that the token could visit every processor in the ring. Moreover, the decision about the size of the timeout had to take into account the durations the various processors needed to hold the token. For example, if some processors were slower than others, the decision about the timeout had to take this into consideration.

Dijkstra replaced the global timeout by a local action- each processor considered its own state and the state of its predecessor only, and acted. He showed that the network converged into a correct global state in spite of this distributed control. It is worth mentioning that Dijkstra’s solution nevertheless involved a global computation. For example, assume that the network was in a legal state, and some adversary changes the state of one processor. In this case, it is possible to return to a correct global state by changing the state of one processor. However, Dijkstra’s solution involves changes in the states of all the processors, as well as time that is long enough for all of them to be involved in the computation. (Moreover, a causal chain of events [18] of length  $\Omega(n)$ , where  $n$  is the number of the processors, may result.)

A part of the elegance in Dijkstra’s algorithms was that they never really detected an illegal state. Instead, when the network was put in an illegal state, it “somehow” converged towards a legal state, and then stayed in the set of legal states. This was also a characteristic of many later algorithms. While elegant, this approach makes the design of algorithms difficult. Katz and Perry [16] suggested a method of partitioning the design of self stabilizing algorithms:

1. Design an algorithm- the *base algorithm*, that is not necessarily self stabilizing (this implies a definition of the legal global states).
2. Detect, in a self stabilizing manner, that the above algorithm reached an illegal state.
3. In the case that an illegal global state is detected, restart the execution of the algorithm from some global legal state.

In fact, they presented a method to perform the detection, given a leader node. The detection (distributed verification) was performed in a rather centralized manner. That is, their algorithm collected all the state information from all the nodes to the leader node. The leader then checked whether the collection of the states was a legal global state. (Collecting local states such that they form a consistent global state is not a trivial task even in a non- self stabilizing network, since in an asynchronous network local states are collected in different times, and may thus not be parts of the same global state [11].)

In terms of complexity, note, first, that the time complexity of the verification task above was linear in the number of nodes. Clearly, the communication cost for the above approach may be large.

The paradigm of *local detection* was developed independently in [31]. This can be viewed as replacing the second step above. The idea was to replace the definition of a correct global state by a collection of definitions for correct local states. Somewhat more formally, assume that the correctness of a global state is defined as a *global* predicate  $P$ , that is,  $P$  is defined over all the variables in all the states of the nodes in the network. Let us say that a predicate is *local* if it is defined only over the state of a single node  $v$  together with the states of all of  $v$ 's neighbors. Now assume that the conjunction of local predicates implies  $P$ . If none of the local predicates is violated, then  $P$  holds.

The above allows to replace the detection step of the Katz and Perry's algorithm by a local detection. Each node collects the states of its neighbors and computes its local predicate repeatedly. If the local predicate is violated at any node, this node starts the recovery phase. The recovery may involve a computation that may not be local. However, the recovery may never be needed, while the attempt to detect an illegal state is performed infinitely often. Hence, it is much more important to have an efficient verification.

A self stabilizing algorithm for a spanning tree construction was presented in [31] for several purposes. First, it demonstrated the local detection by detecting potential cycles in the "tree" using the distance variables (see Section 3). Second, it demonstrated that the local detection could be used also for a dynamically changing state, as opposed to a state that contains already the desired spanning tree and thus is not supposed to change. Specifically, in the algorithm of [31], a node who wished to join a tree sent a message all the way to the tree root to ask for a permission. This message was forwarded by the nodes over the tree. In a self stabilizing environment, it is possible that the node never actually sent that request, even though it "remembers" in its state that it did. Hence, had the algorithm at the node just waited for an answer, a deadlock may have resulted, since such an answer may never arrive (e.g. if the request has never actually been sent). This non- local predicate- "a request message from node  $v$  is on its way to the root" is replaced in the algorithm of [31] by a set of local predicates at the nodes on the route of the request message. If the request message is not there, then some node on its assumed route would detect that illegal state.

Another motivation for the tree algorithm in [31] was to enable a self stabilizing *reset* instead of the third step of [16] (the recovery step). A distributed

reset protocol restarts the base algorithm from a predetermined initial state. It was observed by Awerbuch, Patt-Shamir, Varghese, and Dolev [8] that it is not trivial to show that a general self-stabilizing reset algorithm together with local detection can perform the transformation of any algorithm to a self-stabilizing one correctly. However, it is rather easy to show that a self-stabilizing reset that uses a spanning tree suffices. Several other self-stabilizing tree algorithms were suggested independently. They defer in some of their properties (e.g., one assumed a leader, another used an upper bound on the number of nodes), but they too suggested, at least implicitly, the distributed verification of cycle freedom described in Section 3, see the work by Dolev, Israeli, and Moran, and by Arora and Gouda [13, 4].

The notion of *local checking* was presented in [9]. It bears similarities to the notion of local detection. Instead of a local predicate involving a node and all its neighbors, the local predicates in [9] are defined over the two endpoints of one edge. This has a potential of simplifying algorithms using these predicates. In [7], Awerbuch, Patt-Shamir, and Varghese extended the methodology of local detection and global correction to local detection and local correction. The methodology is applied in [7] to develop self-stabilizing interactive distributed protocols, such as, end-to-end communication and network reset.

As described above, the verification step using the method of [16] consumes  $\Omega(n)$  time, while the verification using e.g. the approach of [31] takes  $O(1)$  time. Methods suggested in [20, 15, 6, 21, 19] to detect cycles sacrificed some time efficiency in order to reduce the total sizes of variables used in the local predicates compared to that of [31]. This suggests the existence of a size-time trade-off. On the other hand, it is not clear whether the total communication cost for these methods is inherently smaller. Indeed, these algorithms communicate a smaller number of bits, but those are communicated to larger distances.

A specific subset of problems allows for a specific kind of self-stabilization called *silent stabilization*. These are studied in Dolev, Gouda, and Schneider in [32]. Informally, when silent stabilization is obtained, the only activity a processor can be involved in is collecting the state information of its neighbors that appear in its local predicates, and computing its local predicates. In a sense, this too is a form of a local detection- if the desired property of the network does not hold (that is, if the network is in an illegal state) this should be detected at least by one node that will take additional actions to correct the state. This captures *input output* relations- for example, this can be useful for protocols that compute a spanning tree. When the tree is correct, no additional activity is required except for the checking. On the other hand, this does not capture an interactive problem, e.g. that of a token ring.

Some of the latter can be captured by the *Roll-back compiler* introduced by Awerbuch and Varghese in [9]. It can be applied to any deterministic protocol (however, if this protocol is not for an input-output problem, then the space used by the compiler may not be bounded). Each node maintains its own log of its events and states, and sends the log often to all its neighbors. Thus, every node can check every transition it made in the past, to see whether its view of

this transition is consistent with the view of its neighbors. These are the local predicates. The global predicate  $P_{history}$  is that “the current global state is a result of a legal run of the base algorithm”. *Local checking* is also used in [9] for designing self stabilizing algorithms directly for several tasks such as shortest paths, topology update, leader election, and computing the maximum flow.

Beauquier, Delaet, Dolev, and Tixeuil (in [33]) assumed that only the part of the state meant to be visible to the outside can be read by other nodes. (The output is the part that appears in the specification of the task to be performed.) It was shown in [33] that this assumption may imply the need for a very large memory usage (e.g. for verifying a spanning tree).

Multiple self stabilizing algorithms have since used the idea of first detecting that the global state is illegal, and then correcting it. This makes a large body of work a potential application of distributed verification. We do not have the space here to survey them all. A rather comprehensive survey (but not up to date) of self stabilization by Herman and Johnen can be found in [1].

We note the following major difference between the model of proof labeling schemes and the ones used by past self stabilization algorithms. In the latter models, the design of the computation stage was intertwined with that of the verification stage, and the designers sought to design a computation process that will be easy for verification, and vice versa. This approach may lead to a low cost local verification. However, this approach might also have the disadvantage of making the design process less modular. In proof labeling schemes, it is assumed that the distributed representation of the structure or function at hand is already given, and the computed labels are required to verify this specific representation. This allows for more modular algorithm design and frees the algorithm designer to consider other goals when designing the distributed representation. The approach of proof labeling schemes may sometimes be useful also in verifying properties on existing structures, even when the original design of those structures was done without verification in mind.

To illustrate this difference, let us point out to one of the results in proof labeling schemes, which states that local checking sometimes requires labels that are longer even than the states (such as the states used in previous local checking methods). This occurs in the natural setting where vertices are required to have distinct states. For example, this can happen in an algorithm that hashes unique identities of nodes into shorter unique states. In the case where the underlying graph is an  $n$ -vertex path, the size of vertex labels that are required in order to verify that all the states are unique is  $\Omega(n)$ . This is longer than the state, which is  $O(\log n)$ . On the other hand, were we allowed to compute the states (rather than prove the given hashing), labels of size zero would have sufficed in the case of unique identities: just have the state equal the identity. (Since the identities are assumed in this example to be unique, the states “computed” in that way are unique too.) We note that in many other cases, “small” proof labeling schemes exist even under the stronger requirements that the state to prove was developed independently, and now it is required to develop the scheme.

## References

1. A Comprehensive Bibliography on Self-Stabilization. A Working Paper in the CJTCS, <http://cjtc.cs.uchicago.edu/>.
2. Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing Journal*, 7:27–34, 1993.
3. S. Aggarwal. Time optimal self-stabilizing spanning tree algorithms. M.Sc Thesis, MIT, May 1994.
4. A. Arora and M. Gouda. Distributed reset. In *Proc. of the 10-th FSTTCS: Springer-Verlag LNCS 472*, pp. 316–331, September 1990.
5. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self stabilizing synchronization. In *Proc. 25th STOC*, pp. 652–661, May 1993.
6. B. Awerbuch and R. Ostrovsky. Memory efficient and self stabilizing network reset. In *PODC*, August 1994.
7. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proc. of the 32nd IEEE FOCS*, pp. 268–277, October 1991.
8. B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self stabilizing by local checking and global reset. in the Proc. of WDAG 94, Springer-Verlag LNCS, pp. 226–239, October 1994.
9. B. Awerbuch, , and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *32nd IEEE FOCS*, pp. 258–267, October 1991.
10. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *CACM*, 17:643–644, November 1974.
11. K. Mani Chandy, Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3(1): 63-75 (1985).
12. E.W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5-6, 1986.
13. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing Journal*, 7, 1994.
14. S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.* 8(4): 424-440 (1997).
15. G. Itkis, and L Levin. Fast and Lean Self-Stabilizing Asynchronous Protocols. In *Proc. of the 35th IEEE FOCS*, pp. 226-239, October 1994.
16. S. Katz and K. J. Perry. Self-stabilizing extensions. *Distributed Computing*, 7, 1994.
17. L. Lamport. Solved problems, unsolved problems and nonproblems in concurrency. Proceedings of the 3rd PODC, pp. 1-11 August 1984.
18. Leslie Lamport: Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21(7): 558-565 (1978).
19. A. Mayer, R. Ostrovsky, and M. Yung. Self-stabilizing algorithms for synchronous unidirectional rings. In *Proc. 7th SODA*, Jan. 1996.
20. A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung. Self-stabilizing symmetry breaking in constant space. In *Proc. 24th STOC*, pages 667-678, May 1992.
21. G. Parlati and M. Yung. Non-exploratory self stabilization for constant-space symmetry-breaking In *Proc. 2d ESA '94*, pages 183–201. LNCS 855 Springer Verlag.
22. M. Naor and L. Stockmeyer. What can be computed locally. In *Proc. 25th STOC*, pp. 185–193. ACM, May 1993.

23. Schieber and Snir. Calling names on nameless networks. *Information and Computation (formerly Information and Control)*, 113, 1994. also in: *Proc. of PODC 1989*, pp. 319–328, August 1989.
24. A. Segall. Distributed network protocols. *IEEE Trans. on Information Theory*, IT-29(1):23–35, January 1983.
25. J. Spinelli and R. G. Gallager. Broadcast topology information in computer networks. *IEEE Trans. on Comm.*, 1989.
26. G. Varghese. *Dealing with Failure in Distributed Systems*. PhD thesis, MIT, 1992.
27. G.M. Jayaram and Varghese. Crash failures can drive protocols to arbitrary states. *PODC 1996*, pp. 247-256.
28. M. Jayaram, G. Varghese. The Complexity of Crash Failures. *PODC 1997*, pp. 179-188.
29. M. Naor and L. Stockmeyer. What can be computed locally? *Proc. 25th STOC*, pp. 184–193, 1993.
30. R.G. Gallager, P.A. Humblet, P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *TOPLAS* 5 (1983) 66-77.
31. Y. Afek, S. Kutten, and M. Yung. The Local Detection Paradigm and Its Application to Self-Stabilization. *Theor. Comput. Sci.* 186(1-2): 199-229 (1997).
32. S. Dolev, M. Gouda, and M. Schneider. Requirements for silent stabilization. *Acta Informatica*, 36(6): 447-462, 1999.
33. Beauquier, J., Delaet, S., Dolev, S., and Tixeuil, S., “Transient Fault Detectors”. *Proc. of the 12th DISC*, Springer-Verlag LNCS:1499, pp. 62-74, 1998.
34. B. Dixon, M. Rauch, and R. E. Tarjan. Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time. *SIAM Journal on Computing*, Vol. 21, No 6, pp. 1184-1192, December 1992.
35. M.L. Fredman and D.E. Willard. Trans-Dichotomous algorithms for minimum spanning trees and shortest paths. *Proc. 31st IEEE FOCS*, Los Alamitos, CA, 1990, pp. 719-725.
36. M. Garey and D. Johnson. *Computers and Intractability*. W.H. Freeman and Company, New York, 1979.
37. D. R. Karger, P.N. Klein, and R.E. Tarjan. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *JACM* Vol. 42, No. 2, pp. 3210328, 1955.
38. Amos Korman and Shay Kutten. “Distributed Verification of Minimum Spanning Trees”. in *Proc. 25th PODC 2006*, July 23-26 2006, Denver, Colorado, USA.
39. M. Katz, N.A. Katz, A. Korman, and D. Peleg. Labeling schemes for flow and connectivity. In *19th SODA*, Jan. 2002.
40. Amos Korman, Shay Kutten, and David Peleg. “Proof Labeling Schemes”. *Proceedings of the 24th PODC 2005*, Las Vegas, NV, USA, July 2005.
41. Nathan Linial. Distributive Graph Algorithms-Global Solutions from Local Data. *FOCS 1987*: 331-335.
42. Fabian Kuhn, Thomas Moscibroda, Roger Wattenhofer. What cannot be computed locally! *PODC 2004*: 300-309.
43. Andrew C. Yao. Some Complexity Questions Related to Distributed Computing. *STOC 1979*, 209-213.