Local Algorithms: Self-Stabilization on Speed

Christoph Lenzen[1], Jukka Suomela[2], and Roger Wattenhofer[1]

[1] Computer Engineering and Networks Laboratory TIK
ETH Zurich, 8092 Zurich, Switzerland
lenzen@tik.ee.ethz.ch, wattenhofer@tik.ee.ethz.ch
http://www.dcg.ethz.ch
[2] Helsinki Institute for Information Technology HIIT
P. O. Box 68, FI-00014 University of Helsinki, Finland
jukka.suomela@cs.helsinki.fi
http://www.hiit.fi

1 Introduction

Fault tolerance is one of the main concepts in distributed computing. It has been tackled from different angles, e.g. by building replicated systems that can survive crash failures of individual components, or even systems that can tolerate a minority of arbitrarily malicious ("Byzantine") participants.

Self-stabilization, a fault tolerance concept coined by the late Edsger W. Dijkstra in 1973 [1, 2], is of a different stamp. A self-stabilizing system must survive arbitrary failures, beyond Byzantine failures, including for instance a total wipe out of volatile memory at all nodes. In other words, the system must self-heal and converge to a correct state even if starting in an arbitrary state, provided that no further faults happen.

Local algorithms, on the other hand, have no apparent relation to fault tolerance. Instead, the basic question studied is whether one can build efficient network algorithms, where any node only knows about its immediate neighborhood. What problems can be solved in such a framework, and how efficiently? Local algorithms have first been studied about 10 years after Dijkstra proposed the notion of self-stabilization [3–6]; recently they experience an Indian summer because of new application domains, such as overlay or sensor networks [7].

It seems that the world of self-stabilization (which is asynchronous, long-lived, and full of malicious failures) has nothing in common with the world of local algorithms (which is synchronous, one-shot, and free of failures). However, as shown in the late 1980s, this perception is incorrect [8, 9]; indeed one can prove quite easily that the two areas are essentially equivalent. Intuitively, this is because (i) asynchronous systems can be made synchronous by using synchronizers [10], (ii) self-stabilization concentrates on the case after the last failure, when the system tries to become correct again, and (iii) one-shot algorithms can just be executed in an infinite loop.

One can show that upper and lower bounds in either area more or less transfer directly to the other area.[3] Unfortunately, it seems that this equivalence has been somewhat forgotten in the last decades. For instance, hardly ever does a paper from one area cite work from the other area. We take the opportunity of this invited paper to summarize the basics, to discuss the latest developments, and to point to possible open problems. We believe that the two areas can learn a great deal from each other!

2 Deterministic Algorithms

The connection between local algorithms and self-stabilizing algorithms is particularly straightforward in the case of deterministic algorithms: any deterministic local algorithm is also a deterministic self-stabilizing algorithm. Furthermore, any deterministic, synchronous local algorithm whose running time is T synchronous communication rounds provides a self-stabilizing algorithm that stabilizes in time T. In this section, we review the conversion in detail, first through an example and then in the general case.

2.1 An Example: Graph Coloring

Throughout this work we consider distributed systems that consist of computational devices and communication links between them. The distributed system is represented as a graph $\mathcal{G} = (V, E)$ with $n = |V|$ nodes: each node $v \in V$ is a device, and two nodes can communicate directly if they share an edge $\{u, v\} \in E$.

Although the connection between local algorithms and self-stabilizing algorithms is more general, in this text we focus on distributed algorithms that solve graph problems. We use the problem of finding a graph coloring as a running example. In this case we want to assign a color $c(v)$ to each node $v \in V$ such that no two adjacent nodes share the same color, i.e., the nodes of each color form an independent set. In general it is NP-hard to determine the minimum number of colors required to color a graph, so we settle for $(\Delta + 1)$-colorings, where Δ is the maximum node degree. Each node v must produce a *local output* from the set $\{0, 1, \ldots, \Delta\}$ such that for any pair of adjacent nodes the local outputs are different.

2.2 A Deterministic Local Algorithm for Graph Coloring

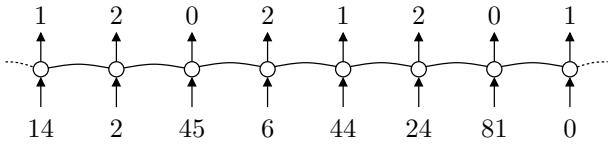
Perhaps the simplest model of distributed computing is a synchronous distributed algorithm. In a synchronous algorithm, all nodes in the network perform steps in parallel: during each *synchronous communication round*, all nodes

[3] So was local algorithms just old wine in new skins? Not really, because the two areas had quite a different focus. Whereas self-stabilization mostly dealt with correctness, local algorithms were all about complexity and efficiency. Today, this difference is disappearing, as also self-stabilization is more and more about efficiency.

in parallel (i) perform local computation, (ii) send out messages to their neighbors, (iii) wait for the messages to propagate along the edges, and (iv) read the incoming messages. Finally the nodes determine their output and terminate. A synchronous *local algorithm* is simply a distributed algorithm that completes in T synchronous communication rounds. Typically T is a constant [6, 11] or a slowly-growing function of n [3–5].

In T communication rounds, information is propagated for only T hops in the communication graph; hence the output of a node v can only depend on the structure of the graph $\mathcal{G}$ in the radius-T neighborhood of v. This is the very idea suggested by the term "local algorithm": nodes make decisions based on *local* information, yet the decisions must be globally consistent.

We start with a variant of a very fast and elegant algorithm, the well-known Cole–Vishkin algorithm [4], which 3-colors an n-cycle in $O(\log^* n)$ rounds. The function $\log^* n$ is defined as the number of times the logarithm has to be applied to n until the result is a constant. This function grows exceptionally slowly and is bounded by a small number for any reasonable size of n. In the Cole–Vishkin algorithm, the local input of a node is a unique identifier with $O(\log n)$ bits, and the local output of a node will be a color from the set $\{0, 1, 2\}$:



To keep things simple, we assume that the nodes know an upper bound on n, and the cycle has a consistent orientation such that each node has one successor and one predecessor:

The algorithm works as follows. Initially, the color of a node is equal to its unique identifier; the idea is to repeatedly decrease the number of colors required. In each round, each node v sends its current color to its successor w. The node w compares bitwise its own color to the received one to determine the least significant bit where they differ. It binary encodes the position and appends the differing bit, resulting in its new color in the form of a bit string. The new color of w cannot be identical to the new color of its predecessor v: either the indices of the bits v and w determined are not the same, meaning that the colors have a different prefix, or the computed indices referred to bits with different values, i.e., the new colors differ in their terminal bits.

The following example shows two iterations of the algorithm on a part $t \to u \to v \to w$ of a cycle:

$$\begin{aligned} t&: \ 1010110000 \to \ \ldots \ \to \ldots \\ u&: \ 0010110000 \to 10010 \to \ldots \\ v&: \ 1010010000 \to 01010 \to 111 \\ w&: \ 0110010000 \to 10001 \to 001. \end{aligned}$$

The initial colors, i.e., the nodes' unique identifiers, have $O(\log n)$ bits. After one step, the colors consist of $O(\log \log n)$ bits, namely a binary encoded position in a string of length $O(\log n)$ plus one bit. Applying this observation also to subsequent rounds, we see that after $O(\log^* n)$ rounds, the number of bits—and thus colors—has become constant. At this point, a simple constant-time algorithm can be used to reduce the number of colors to $\Delta + 1 = 3$: in each round, we remove the largest color.

In summary, we have an algorithm for 3-coloring an n-cycle in $O(\log^* n)$ rounds; furthermore, this running time is asymptotically optimal [5]. The approach can be generalized to bounded-degree graphs and rooted trees [12, 13]. Recently, the technique was utilized to find colorings in bounded-independence graphs in $O(\log^* n)$ rounds [14]; we will discuss recent work in more detail in Sect. 4.1.

2.3 A Self-Stabilizing Algorithm for Graph Coloring

The local algorithm presented in the previous section is not fault-tolerant in any way. We assumed that all nodes are activated simultaneously in a specific initial state, and the network does not change during the execution of the algorithm. The algorithm eventually stops, after which it does not react in any way to changes in the network. Furthermore, we assumed that all nodes perform computations in a synchronous manner, as if a global clock pulse was available.

Nevertheless, it is possible to convert this local algorithm into an efficient asynchronous *self-stabilizing* algorithm. A self-stabilizing algorithm, by definition, provides an extreme form of fault tolerance [2, 15, 16]: an adversary can choose an arbitrary initial configuration, and a self-stabilizing algorithm is still guaranteed to converge into a correct output.

For the sake of concreteness, we use the shared-memory model here: we assume that each communication link $\{u, v\} \in E$ consists of a pair of communication registers, one which is written by u and read by v, and one for passing information in the opposite direction. Typically support of atomic reads and writes is assumed.

In this model, a *configuration* of the system consists of the local outputs of the nodes, the contents of the local variables of the nodes, and the contents of the communication registers. In a *legitimate configuration* the system behaves as intended—in our example, a legitimate configuration simply refers to any configuration in which the local outputs of the nodes form a valid coloring. We refer to Dolev's book [16, §2] for more details on these definitions and on the model of self-stabilizing algorithms in general.

We now convert the variant of the Cole–Vishkin algorithm presented in Sect. 2.2 into an asynchronous self-stabilizing algorithm. Asynchronicity means here that there are no guarantees on how fast computations are done and information is exchanged. Rather, the algorithm must be resilient to a worst-case situation where a non-deterministic distributed daemon may schedule any computational step at any node next. The algorithm must reach a legitimate state

regardless of the decisions of the daemon. The time complexity of an asynchronous self-stabilizing algorithm is defined as the number of *asynchronous cycles* required to converge from an arbitrary state to a legitimate configuration; an asynchronous cycle is an execution during which each node at least once reads its input and incoming messages, and infers and writes its new output and outgoing messages.

The algorithm from Sect. 2.2 can be adapted to this model as follows. Let $T = O(\log^* n)$ be the running time of the Cole–Vishkin algorithm. For each edge in the cycle, we divide the associated communication register (in the described algorithm communication is unidirectional, hence a single register suffices) into T parts, each of which represents one round of the local algorithm. Let v be a node in the oriented cycle, with predecessor u and successor w. Now the state of the communication register on the edge $\{u, v\}$ corresponds to *all* messages that u sends to v during the execution of the Cole–Vishkin algorithm; similarly, the register on the edge $\{v, w\}$ corresponds to the messages sent by v to w.

The node v continuously reads its input (its unique identifier) and the values in the communication register on the edge $\{u, v\}$. The node v *simulates* its own actions during a complete execution of the Cole–Vishkin algorithm, *assuming* that these incoming messages are correct, and writes its own outgoing messages to the communication register on the edge $\{v, w\}$. The node also continuously re-writes its local output based on this simulation.

Naturally, in the beginning the output might be nonsense, as the initial memory state is arbitrary. After one asynchronous cycle, however, the nodes will have (re)written their identifiers into the parts of the registers responsible for the messages in round one of the Cole–Vishkin algorithm. In the next cycle, their neighbors will read them, compute the new colors, and write them into the parts for round two, and so on. After $T + 1$ asynchronous cycles, the initial state of the system has been erased and replaced by the values the local algorithm would compute in a single run, independently of the schedule the daemon chooses. Hence the same arguments as in the previous section prove that the output must be correct at all nodes. Moreover, no further state transitions occur, as the outcome of all steps of the computation depends only on the local inputs (unique identifiers) of the nodes.

We conclude that the algorithm stabilizes within $T + 1$ asynchronous cycles, where T is the running time of the local algorithm. Hence in the conversion from local to self-stabilizing algorithms, we can guarantee much more than mere *eventual* convergence into a legitimate configuration: we can show that the convergence is *fast*.

Note that the algorithm is also efficient in terms of the number of bits sent and the required memory. In total

$$\sum_{i=1}^{T} O(\log^{(i)} n) = O(\log n)$$

bits need to be exchanged along each edge, where $\log^{(i)} n$ denotes the i times iterated logarithm. Apart from the presented special case where edges are ori-

ented, this bit complexity is asymptotically optimal [17], a result also holding for randomized algorithms which are presented in Sect. 3. No additional memory beyond the communication registers is needed.

2.4 General Case

The example of Sect. 2.3 was fairly simple: in the original local algorithm, each node sends messages to only one neighbor. However, the general case is not much more complicated: there are two communication registers on each edge, and all registers are divided in T parts, one part for each communication round.

Figure 1 shows the basic idea behind the conversion: given any deterministic distributed algorithm $\mathcal{A}$ with running time T, we can construct an equivalent *circuit* that produces the same output as $\mathcal{A}$. The figure shows the conversion in the case where the communication graph $\mathcal{G}$ is a cycle, but the same idea can be applied to arbitrary graphs.

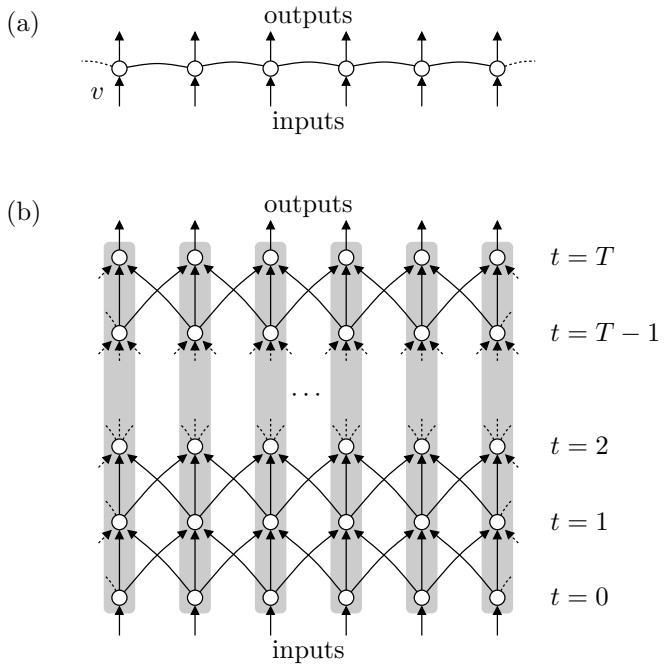


Fig. 1. (a) A distributed system that executes a local deterministic algorithm $\mathcal{A}$ with running time T. (b) A circuit that computes the same output.

Each node v in Fig. 1a is replaced by $T + 1$ virtual nodes $v_0, v_1, \ldots, v_T$ in Fig. 1b. The node v_0 represents the initial state of the node v in the algorithm $\mathcal{A}$, and the node v_i for $i = 1, 2, \ldots, T$ represents the state of the node v in

the algorithm $\mathcal{A}$ after the synchronous communication round i. A directed edge from v_{i-1} to u_i represents the message sent by v to u during the synchronous communication round i. Clearly the output of the circuit is equal to the output of the original algorithm $\mathcal{A}$.

So far we seem to have gained little: we have just an alternative representation of the original local algorithm $\mathcal{A}$. However, the key observation is that it is easy to *simulate* the computations of the circuit of Fig. 1b by a self-stabilizing algorithm. Furthermore, the simulation can be realized in virtually any model of distributed computing (assuming, of course, that the model allows us to implement any kind of reliable computation at all).

In essence, we simply replace each diagonal edge from v_{i-1} to u_i by a point-to-point communication channel from the node v to u. The node v continuously

1. re-reads its local input and all incoming messages,
2. simulates the behavior of $\mathcal{A}$ for each time step, assuming that the incoming messages are correct, and
3. re-writes its local output and all outgoing messages.

After $i+1$ asynchronous cycles, the outgoing signals of the virtual nodes v_i are correct, and after $T+1$ asynchronous cycles, the output of each node is correct, regardless of the initial configuration.

In the example of Sect. 2.3 we implemented point-to-point communication from u to v by using a communication register that was written by u and read by v. Equally well we could use the message-passing model and a self-stabilizing implementation of unit capacity data links; see, e.g., Awerbuch et al. [18].

Naturally, if T is large, say, $T = \Theta(n)$, then the conversion of Fig. 1 is of little use. However, in the case of local algorithms, typically $T \ll n$ and in some cases even $T = O(1)$. Hence this simple and easy-to-implement conversion yields an efficient self-stabilizing algorithm for most deterministic local algorithms. In particular, constant-time distributed algorithms are also self-stabilizing algorithms that stabilize in constant time. Furthermore, the memory requirement and message size is increased only by a factor of T: for example, if the original local algorithm transmits m-bit messages on each edge, the self-stabilizing algorithm sends (Tm)-bit messages.

2.5 The Simple Conversion in Literature

The observation that deterministic distributed algorithms can be easily converted into self-stabilizing algorithms is by no means new. The conversion of Fig. 1 is, in essence, equal to the "simulator" introduced by Awerbuch and Sipser [8] more than 20 years ago. Awerbuch and Sipser explicitly referred to the problem of simulating local algorithms, even though the field of local algorithms barely existed back then. While Awerbuch and Sipser did not focus on self-stabilizing algorithms, all key ingredients were already present. Their algorithm was triggered by a topology change in the network; equally well we can trigger the algorithm by periodically re-reading the inputs, and we obtain a self-stabilizing algorithm.

Awerbuch and Varghese [9] make the connection between synchronous distributed algorithms and self-stabilizing algorithms explicit. They use the term "rollback compiler" to refer to a simple conversion similar to that of Fig. 1. In their terminology, the states of the virtual nodes $v_0, v_1, \ldots, v_T$ together with the incoming messages constitute a *log* that contains the full execution history of the node v; hence the node can *verify* that the execution of the algorithm $\mathcal{A}$ is correct from its own perspective. If the execution is correct from the perspective of all nodes, then also the outputs are correct and the algorithm has stabilized. By keeping track of the execution history, we have made the output of the distributed algorithm locally checkable.

The simple conversion can also be interpreted as an application of local checking and correction that is introduced in Awerbuch et al. [18]. We can *locally check* the state of each directed edge in Fig. 1b. If a link (u_{i-1}, v_i) is in an inconsistent state, we can *locally correct* the state of v_i. By construction, each dependency chain in this system has length at most T, and hence the system will stabilize in time $T + 1$.

However, even though the simple conversion itself is well-known [16, §5.1], it seems that fairly little attention has been paid to it in the literature. The main focus has been on non-local problems such as spanning trees and leader election. Even in Awerbuch and Varghese's [9] work the main contribution is related to the conversion of non-local distributed algorithms whose running time T is larger than the diameter of the network.

A notable exception is Mayer et al. [19]. In this work—which is a follow-up of the seminal paper by Naor and Stockmeyer [6] that initiated the study of strictly local (constant-time) distributed algorithms—Mayer et al. specifically draw attention to the connection between local algorithms and fault-tolerance in dynamic systems. However, the field of local algorithms was still in its infancy in 1995, and positive examples of local algorithms were scarce.

We believe it is time to revisit the issue now, as we have numerous recent examples of local algorithms. In Sect. 4, we survey highlights from the field of local algorithms—both positive and negative results—and explain what implications they have from the perspective of self-stabilizing algorithms. However, we will first have a look at the much more complicated issue of randomized local algorithms.

3 Randomized Algorithms

So far we have restricted our attention to *deterministic* local algorithms. There is a considerable number of local algorithms that are *randomized*, i.e., each node has access to (uniformly) random bits. These can be useful to break symmetry or locally take decisions that probably perform well on a global scale, creating algorithms which are likely to be faster than their deterministic counterparts, to achieve better approximation guarantees, or to yield correct solutions despite short running times.

3.1 Basic Symmetry Breaking

Sometimes deterministic algorithms are even incapable of solving a particular task. Coloring an anonymous cycle, i.e., a cycle without a means to distinguish between nodes, is impossible without randomization. Due to total symmetry, when executing a deterministic algorithm, all nodes must take the same actions and eventually attain the same color.[4] On the other hand, running the Cole–Vishkin algorithm from Sect. 2 with $O(\log n)$ random bits as "identifier" at each node will result in a correct output *with high probability* (w.h.p.), i.e., for any choice of a constant $c > 0$ we can bound the probability of failure by $1/n^c$. Using random bit strings of length $(c + 2) \log n$, any pair of nodes will have distinct bit strings with probability $1/n^{c+2}$. Summing over all pairs of nodes, the probability of two nodes having the same string can be bounded by $n(n-1)/(2n^{c+2}) < 1/n^c$. Thus, with probability at least $1 - 1/n^c$, we can interpret the random bits as correct input of the deterministic Cole–Vishkin algorithm relying on unique identifiers.

When this technique is to be employed in the self-stabilizing world, we cannot guarantee globally unique identifiers any more unless accepting a stabilization time of $\Omega(D)$, since there is no way to distinguish a corrupted memory state from a correct one if not comparing the identifiers. However, for many algorithms, in particular routines such as Cole–Vishkin dealing with breaking of *local* symmetry, a locally unique labeling, i.e., *any* coloring, will do. Assuming (an approximation of) n is known, we merely need to continuously compare the "random identifiers" of neighbors, and generate a new random bit string if a conflict is observed. This very simple algorithm self-stabilizes w.h.p. within one or two cycles, depending on the precise model, and can be a building block for more sophisticated algorithms.

3.2 Pseudo-Randomization

The general transformation from Sect. 2.4 fails for randomized algorithms. On the one hand, if nodes make their random choices in advance and proceed deterministically, an adversary may tamper with the state of the memory holding the random bits, and the algorithm will be slow, yield bad approximations, or even completely fail. On the other hand, if nodes take random choices in each step of the algorithm "on the fly", the execution of the algorithm itself is not deterministic. In this case, we cannot represent the state of a node in a given (synchronous) round as function of the states of the nodes in the previous round, and thus also not represent the respective computations by a Boolean circuit. Rather, to guarantee uncorrupted random choices, nodes would have to continuously renew their random bits, preventing convergence to a fixed output.

From a practical point of view, this problem can be tackled easily: Instead of generating actual random numbers, we use fixed unique random seeds, i.e.,

[4] Asynchrony might break symmetry, but in the worst case it will certainly not. Here the worst case ironically is the system being perceived as synchronous.

node identifiers, as part of the input. These bits are read-only and can be seen as part of the protocol itself, i.e., they are not subject to the arbitrariness of initial states. Using a pseudo-random function with the node identifier in conjunction with the round number as input, nodes can generate pseudo-random bits for use by a randomized algorithm. Since these bits are computed deterministically at running time, the conversion from Sect. 2.4 can be applied again to infer asynchronous and self-stabilizing algorithms from synchronous randomized counterparts.

Assuming that no correlation between the random seeds and the problem-specific input exists, and provided that a well-behaving pseudo-random function is used, the performance of the algorithm will be indistinguishable from a "true" randomized algorithm's: We simply ensured a supply of random bits in advance by storing a previous random choice in non-volatile memory to avoid corruption. Hence, in practice also randomized local algorithms lead to efficient self-stabilizing solutions in a straightforward manner.

3.3 Theoretical Questions

From a theoretical point of view, the use of pseudo-randomization is noneffective. Regardless of the computations made, previously stored values do not replace randomly generated numbers. At best, if the stored bits have been generated uniformly at random and the other input is independent of these choices, each stored bit can be used once as a random bit. At worst, a sufficiently powerful adversary might learn about nodes' pseudo-random choices by experimentation or having access to the complete state of nodes, and afterwards modify the input in a way such that the pseudo-random choices are badly suited to the created problem instance. After all, pseudo-randomness does not change the deterministic behavior of the algorithm, and therefore any lower bound applicable to deterministic algorithms must hold.

In fact, to the best of our knowledge, little is known about which randomized local algorithms can be made self-stabilizing efficiently. We presented a trivial, yet important example at the beginning of the section which tolerates asynchronicity. Synchronous randomized algorithms may require synchronous systems to self-stabilize quickly, as the random choices of a given round need to be correlated. This however might limit their usability in an asynchronous environment, since a self-stabilizing synchronizer requires time in the order of the diameter of the network to stabilize [20], a bound that—at least when ignoring other complexity measures—is trivial to local algorithms, since nodes may learn about the whole topology and all local inputs in that time.

4 Results on Local Algorithms

In this section, we present selected results from the field of local algorithms, with the main focus on recent discoveries. Most of the results are deterministic algorithms or lower-bound results, each of which has a direct self-stabilizing counterpart. We have also included examples of randomized local algorithms—some

of these can be made self-stabilizing by using the symmetry breaking technique discussed in Sect. 3, while developing self-stabilizing versions of others provides challenges for future research. We begin with the theme that we have used as a running example in Sections 2 and 3, graph coloring.

4.1 Colorings, Independent Sets, and Matchings

In the study of traditional centralized algorithms, graph coloring is often seen from the perspective of *optimization*: the goal is to minimize the number of colors. This perspective leads to many famous results in graph theory and computer science; finding an optimal coloring is a classical NP-hard problem, and numerous (in)approximability results, practical heuristics, and exponential-time exact algorithms are known.

However, in distributed computing, graph coloring is usually regarded as a fundamental *symmetry-breaking* primitive. From this point of view, minimizing the number of colors is not necessary—a coloring with $\Delta + 1$ colors is sufficient for symmetry-breaking purposes. While such a coloring is trivial to find in a centralized setting by using a greedy algorithm, the problem of finding such colorings efficiently in a distributed setting has been a central question from the very first days of the field to the present day. These efforts have resulted in fast algorithms and tight impossibility results, both of which transfer directly to a self-stabilizing setting.

Before reviewing the key results, it is worth mentioning that there is another symmetry-breaking problem that is essentially equal to graph coloring: the problem of finding a *maximal independent set*. Given a k-coloring, it is easy to find a maximal independent set in time k. Conversely, if we have an algorithm for finding a maximal independent set, we can use it to find a $(\Delta + 1)$-coloring [5]. Another related symmetry-breaking problem is finding a *maximal matching*. In particular, in the case of directed cycles a maximal matching is equivalent to a maximal independent set: the outgoing edges of independent nodes form a matching and vice versa.

From this background it comes as no surprise that all these problems have essentially the same time complexity in bounded-degree graphs, already familiar from Sect. 2: if $\Delta = O(1)$, then it is possible to find a $(\Delta+1)$-coloring, a maximal independent set, and a maximal matching in $O(\log^* n)$ rounds, and not faster.

Deterministic Algorithms. Naturally, all deterministic algorithms that break the symmetry require some kind of initial symmetry-breaking information [21]. The algorithms that we discuss here assume that each node has a unique identifier. The unique identifiers do not make the problems trivial, though. Linial's results [5] show that even in the case of directed cycles with unique identifiers, there is no constant-time algorithm for finding a graph coloring, maximal independent set, or maximal matching. Any such algorithm requires $\Omega(\log^* n)$ communication rounds.

We already presented the Cole–Vishkin algorithm [4] for coloring a cycle in Sect. 2; the running time of this algorithm matches Linial's lower bound. Since the publication of Cole and Vishkin's seminal work in 1986, numerous algorithms have been presented for the problem of coloring an arbitrary graph with $\Delta + 1$ colors; typically, such algorithms have time complexity of the form $O(f(\Delta)+\log^* n)$. Examples of these include an algorithm by Goldberg et al. [22] with a running time of $O(\Delta^2 + \log^* n)$ rounds, and by Kuhn et al. [23] with running time $O(\Delta \log \Delta+\log^* n)$. The recent algorithms by Barenboim and Elkin [24] and Kuhn [25] finally push the running time down to $O(\Delta + \log^* n)$. These results also provide a deterministic algorithm for finding a maximal independent set in $O(\Delta + \log^* n)$ rounds. Schneider et al. [14] study *bounded-independence graphs*, i.e., graphs in which any constant-radius subgraph contains at most $O(1)$ independent nodes. In this family, a maximal independent set, a maximal matching, or a $(\Delta + 1)$-coloring can be found in $O(\log^* n)$ rounds.

There are also efficient distributed algorithms that directly solve the problem of finding a maximal matching. Some of the algorithms have running times of the familiar form $O(f(\Delta)+\log^* n)$: Panconesi and Rizzi [26] find a maximal matching in $O(\Delta + \log^* n)$ rounds. However, there are also algorithms that perform well even if $\Delta = \Theta(n)$. For example, Hańćkowiak et al. [27] find a maximal matching in $O(\log^4 n)$ rounds.

In summary, at least for bounded-degree graphs (or more general bounded-independence graphs), these three symmetry-breaking problems admit very efficient and asymptotically optimal deterministic solutions.

Randomized Algorithms. In the case of deterministic algorithms, we assumed that we have unique identifiers in the network. However, a much weaker assumption is usually sufficient: it is enough to have a graph coloring (possibly with an unnecessarily large number of colors). Many deterministic graph coloring algorithms, including the original Cole–Vishkin algorithm, simply perform *color reductions* steps: in each iteration, a k-coloring is replaced with an $O(\log k)$-coloring.

Therefore we can apply a randomized graph coloring algorithm, such as the one mentioned in Sect. 3.1, to obtain an initial k-coloring, and then use deterministic local algorithms to find a $(\Delta+1)$-coloring, a maximal independent set, or a maximal matching. Such a composition results in a randomized self-stabilizing algorithm that can be used in anonymous networks without unique identifiers.

There are also randomized local algorithms that find a maximal independent set directly, without resorting to a randomized graph coloring algorithm. The most famous example is Luby's [3] randomized algorithm from 1986 that finds a maximal independent set in $O(\log n)$ rounds w.h.p.; similar results were also presented by Alon et al. [28] and Israeli et al. [29] around the same time. Recently, Métivier et al. [30] presented a new variant featuring a simpler analysis. While we are not aware of a self-stabilizing version of Luby's algorithm, there has been progress in this direction. Already in 1988, Awerbuch and Sipser [8] studied Luby's algorithm in dynamic, asynchronous networks, and more recently the algorithm has been studied in a fault-tolerant setting by Kutten and Peleg [31].

4.2 Linear Programs

Now we change the perspective from symmetry-breaking problems to optimization problems. Many resource-allocation questions in computer networking can be naturally formulated as *distributed linear programs* (LPs): each (physical or virtual) node in the network represents a variable or a constraint, with an edge between a variable and each constraint that depends on it. Papadimitriou and Yannakakis [32] raised the question of solving such linear programs in a local manner so that the value of each variable is chosen using only information that is available in its local neighborhood in the network.

Clearly such algorithms cannot produce an optimal solution—in some cases even finding a feasible solution requires essentially global information on the problem instance. However, there are important families of linear programs that admit *local approximation algorithms*, i.e., algorithms that find a solution that is guaranteed to be feasible and near-optimal.

The most widely-studied families are *packing and covering LPs*. In a packing LP, the objective is to maximize $\boldsymbol{c}^\top \boldsymbol{x}$ subject to $A\boldsymbol{x} \leq \mathbf{1}$ and $\boldsymbol{x} \geq \mathbf{0}$ for a non-negative matrix A; a covering LP is the dual of a packing LP. Distributed approximation algorithms for packing and covering LPs have been presented by Bartal et al. [33] and by Kuhn et al. [34, 35].

For example, in the case of $\{0, 1\}$ coefficients, Kuhn et al. [35] find a $(1+\epsilon)$-approximation in $O(\epsilon^{-4} \log^2 \Delta)$ rounds; here the degree bound Δ is the maximum number of non-zero elements in any row or column of the matrix A. If Δ is a constant, the algorithm is strictly local in the sense that the approximation ratio and the running time are independent of the number of nodes. Moreover, it is a *local approximation scheme*: an arbitrarily good approximation ratio can be achieved. The algorithm is deterministic, an therefore it can be easily converted into a self-stabilizing algorithm.

It is also known that the dependency on Δ in the running time is unavoidable. Kuhn et al. [35, 36] present lower bound constructions that, in essence, show that finding a constant-factor approximation of a packing or covering LP requires $\Omega(\log \Delta / \log \log \Delta)$ rounds, even in various special cases such as the LP relaxations of minimum vertex cover and maximum matching. The same construction also gives a lower bound of $\Omega(\sqrt{\log n / \log \log n})$ rounds as a function of n. Such lower bounds have applications far beyond linear programming, as they also give lower bounds for the original combinatorial problems. Incidentally, the lower bounds by Kuhn et al. hold even in the case of randomized algorithms with probabilistic approximation guarantees.

The family of *max-min LPs* combines packing and covering constraints. In a max-min LP, the objective is to maximize ω subject to $A\boldsymbol{x} \leq \mathbf{1}$, $C\boldsymbol{x} \geq \omega\mathbf{1}$, and $\boldsymbol{x} \geq \mathbf{0}$ for non-negative matrices A and C. While arbitrarily good approximation factors can be achieved for packing and covering LPs in bounded-degree graphs with a strictly local algorithm, this is no longer the case for max-min LPs—indeed, a tight pair of positive [37] and negative [38] results is known for the approximation factor achievable with a strictly local algorithm. Nevertheless, for certain families of graphs better approximation algorithms are known [39].

4.3 Randomized LP Rounding

In addition to being the workhorse of operations research, linear programming has found numerous applications in the field of combinatorial optimization [40]. Many of the best polynomial-time approximation algorithms build on the theory of linear programming [41]. The case is the same in the field of local algorithms.

Putting together the LP approximation schemes discussed in Sect. 4.2 and the technique of *randomized rounding* [34, 35, 42], it is possible to find good approximations for many classical combinatorial problems. For example, in the case of the minimum dominating set problem, we can study the *LP relaxation* of the problem. This is a covering LP, and using the LP approximation schemes, we can find a near-optimal solution, i.e., a near-optimal fractional dominating set.[5] Now the challenge is to construct an integral dominating set whose size is not much worse than the size of the fractional dominating set; this can be solved by using a two-step randomized algorithm which provides an $O(\log \Delta)$-approximation in expectation. In addition to covering problems such as dominating set, this approach can be applied to solve packing problems: the expected approximation factor is $O(\Delta)$ for maximum independent sets and $O(1)$ for maximum matchings. The running time is essentially equal to the running time of the LP approximation scheme.

One of the main drawbacks of this approach is that the use of randomness seems to be unavoidable, and it is not obvious how to design a self-stabilizing algorithm with the same performance guarantees. However, there are various other techniques that can be used to design local approximation algorithms; we review these in the following section.

4.4 Other Combinatorial Approximation Algorithms

The classical problem of finding a minimum-size vertex cover serves as a good example of alternatives to randomized LP rounding. There are at least three other approaches. First, it turns out that vertex cover can be approximated well by using a deterministic LP-based algorithm. The LP approximation schemes by Kuhn et al. [35] together with a simple deterministic rounding technique [43] yield a $(2+\epsilon)$-approximation in $O(\epsilon^{-4} \log \Delta)$ rounds. This algorithm as a whole can be made self-stabilizing directly by using the approach from Sect. 2.

Second, we can use maximal matchings. The endpoints of a maximal matching form a 2-approximation of vertex cover. Hence from the results mentioned in Sect. 4.1, we immediately have deterministic 2-approximation algorithms for vertex cover with running times $O(\log^4 n)$ [27] and $O(\Delta + \log^* n)$ [26].

Third, there is a recent deterministic algorithm that finds a 2-approximation of a minimum vertex cover in $O(\Delta^2)$ rounds [44] without resorting to maximal matchings. The algorithm does not require unique identifiers, making it particularly easy to convert into a self-stabilizing algorithm even in anonymous networks.

[5] A fractional dominating set assigns to each node a weight from $[0, 1]$ such that the sum of a node's own and neighbors' weights is at least 1.

Finally, there are also strong lower-bound results. For example, in the case of a constant Δ, the above-mentioned algorithm finds a 2-approximation of a minimum vertex cover in constant time. This approximation factor is tight: lower bound results [45, 46] show that a $(2-\epsilon)$-approximation is not possible in constant time for any constant $\Delta \geq 2$. Furthermore, the lower bound result by Kuhn et al. [36] proves that a constant Δ is necessary if we want constant running time and constant approximation factor.

In summary, the problem of approximating vertex covers by distributed algorithms is nowadays well understood: there is a whole range of deterministic algorithms from which to choose, and there are also strong lower-bound results. All these results have straightforward corollaries in a self-stabilization setting.

Also the minimum dominating set problem that we used as an example in Sect. 4.3 admits deterministic approximation algorithms—at least for special cases and variants of the problem. Recent results include two constant-time distributed algorithms that find a constant-factor approximation of a minimum dominating set in a planar graph [45, 47]. There is also a deterministic $O(\log^* n)$-time algorithm that finds a constant-factor approximation of a minimum connected dominating set in bounded-independence graphs [14].

The classical optimization problem of finding a maximum-size independent set can be used to illustrate the trade-off between randomization and running time. As the maximum independent set problem is hard to approximate even in a centralized setting, we focus on the special case of planar graphs. Czygrinow et al. [45] present both deterministic and randomized local approximation schemes: the deterministic algorithm finds a good approximation in $O(\log^* n)$ rounds, while the randomized algorithm finds a good approximation in $O(1)$ rounds w.h.p. Together with the recent lower bound results [45, 46], this work shows that randomized local algorithms are asymptotically faster than deterministic local algorithms in some optimization problems, giving additional motivation for studying the conversion of local randomized algorithms into self-stabilizing randomized algorithms.

We refer to Elkin's [48] survey for more information on distributed approximation algorithms. There is also a recent survey [11] that focuses specifically on constant-time distributed algorithms.

5 Conclusion

We misused this invited paper to remind the local algorithms and self-stabilization communities that they share a long history. After recapitulating the elementary observation that *any* deterministic local algorithm has a self-stabilizing analogon, we highlighted recent results on efficient local algorithms. We are convinced the relation goes in both directions—we believe that a similar article could be written from a vantage point of self-stabilization.

Several issues are still open. In our view randomization is not fully understood in this context. We thus encourage experts from both fields to explore to what extent randomization techniques can be transferred between the two areas. Also,

we merely touched the surface of bit complexity, the quality of an algorithm in terms of the number of exchanged bits. In the last decades considerable progress has been made both in minimizing the bit complexity of local algorithms as well as in establishing lower bounds. We conjecture that both communities can profit from ascertaining each others' results. And finally, there are several areas related to both local algorithms and self-stabilization, e.g. dynamic networks or self-assembly [49].

Acknowledgements. This work was supported in part by the Academy of Finland, Grant 116547, by Helsinki Graduate School in Computer Science and Engineering (Hecse), and by the Foundation of Nokia Corporation.

References

1. Dijkstra, E.W.: Self-stabilization in spite of distributed control. Manuscript EWD391 (October 1973)
2. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17**(11) (1974) 643–644
3. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing **15**(4) (1986) 1036–1053
4. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control **70**(1) (1986) 32–53
5. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing **21**(1) (1992) 193–201
6. Naor, M., Stockmeyer, L.: What can be computed locally? SIAM Journal on Computing **24**(6) (1995) 1259–1277
7. Suomela, J.: Optimisation Problems in Wireless Sensor Networks: Local Algorithms and Local Graphs. PhD thesis, University of Helsinki, Department of Computer Science, Helsinki, Finland (May 2009)
8. Awerbuch, B., Sipser, M.: Dynamic networks are as fast as static networks. In: Proc. 29th Symposium on Foundations of Computer Science (FOCS), IEEE (1988) 206–219
9. Awerbuch, B., Varghese, G.: Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In: Proc. 32nd Symposium on Foundations of Computer Science (FOCS), IEEE (1991) 258–267
10. Awerbuch, B.: Complexity of network synchronization. Journal of the ACM **32**(4) (1985) 804–823
11. Suomela, J.: Survey of local algorithms. Manuscript (2009)
12. Goldberg, A.V., Plotkin, S.A.: Parallel $(\Delta+1)$-coloring of constant-degree graphs. Information Processing Letters **25**(4) (1987) 241–245
13. Peleg, D.: Distributed Computing – A Locality-Sensitive Approach. SIAM (2000)
14. Schneider, J., Wattenhofer, R.: A log-star distributed maximal independent set algorithm for growth-bounded graphs. In: Proc. 27th Symposium on Principles of Distributed Computing (PODC), ACM Press (2008) 35–44
15. Schneider, M.: Self-stabilization. ACM Computing Surveys **25**(1) (1993) 45–67
16. Dolev, S.: Self-Stabilization. The MIT Press, Cambridge, MA (2000)
17. Kothapalli, K., Scheideler, C., Onus, M., Schindelhauer, C.: Distributed coloring in $\tilde{O}(\sqrt{\log n})$ bit rounds. In: Proc. 20th International Parallel and Distributed Processing Symposium (IPDPS), IEEE (2006)

18. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction. In: Proc. 32nd Symposium on Foundations of Computer Science (FOCS), IEEE (1991) 268–277
19. Mayer, A., Naor, M., Stockmeryer, L.: Local computations on static and dynamic graphs. In: Proc. 3rd Israel Symposium on the Theory of Computing and Systems (ISTCS), IEEE (1995) 268–278
20. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: Proc. 25th Symposium on Theory of Computing (STOC), ACM Press (1993) 652–661
21. Angluin, D.: Local and global properties in networks of processors. In: Proc. 12th Symposium on Theory of Computing (STOC), ACM Press (1980) 82–93
22. Goldberg, A.V., Plotkin, S.A., Shannon, G.E.: Parallel symmetry-breaking in sparse graphs. SIAM Journal on Discrete Mathematics **1**(4) (1988) 434–446
23. Kuhn, F., Wattenhofer, R.: On the complexity of distributed graph coloring. In: Proc. 25th Symposium on Principles of Distributed Computing (PODC), ACM Press (2006) 7–15
24. Barenboim, L., Elkin, M.: Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. In: Proc. 41st Symposium on Theory of Computing (STOC), ACM Press (2009) 111–120
25. Kuhn, F.: Weak graph colorings: Distributed algorithms and applications. In: Proc. 21st Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM Press (2009) To appear.
26. Panconesi, A., Rizzi, R.: Some simple distributed algorithms for sparse networks. Distributed Computing **14**(2) (2001) 97–100
27. Hańćkowiak, M., Karoński, M., Panconesi, A.: On the distributed complexity of computing maximal matchings. SIAM Journal on Discrete Mathematics **15**(1) (2001) 41–57
28. Alon, N., Babai, L., Itai, A.: A fast and simple randomized parallel algorithm for the maximal independent set problem. Journal of Algorithms **7**(4) (1986) 567–583
29. Israeli, A., Itai, A.: A fast and simple randomized parallel algorithm for maximal matching. Information Processing Letters **22**(2) (1986) 77–80
30. Métivier, Y., Robson, J.M., Nasser, S.D., Zemmari, A.: An optimal bit complexity randomised distributed MIS algorithm. In: Proc. 16th International Colloquium on Structural Information and Communication Complexity (SIROCCO), Springer (2009) To appear
31. Kutten, S., Peleg, D.: Tight fault locality. SIAM Journal on Computing **30**(1) (2000) 247–268
32. Papadimitriou, C.H., Yannakakis, M.: Linear programming without the matrix. In: Proc. 25th Symposium on Theory of Computing (STOC), ACM Press (1993) 121–129
33. Bartal, Y., Byers, J.W., Raz, D.: Global optimization using local information with applications to flow control. In: Proc. 38th Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press (1997) 303–312
34. Kuhn, F., Wattenhofer, R.: Constant-time distributed dominating set approximation. Distributed Computing **17**(4) (2005) 303–310
35. Kuhn, F., Moscibroda, T., Wattenhofer, R.: The price of being near-sighted. In: Proc. 17th Symposium on Discrete Algorithms (SODA), ACM Press (2006) 980–989
36. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally! In: Proc. 23rd Symposium on Principles of Distributed Computing (PODC), ACM Press (2004) 300–309

37. Floréen, P., Kaasinen, J., Kaski, P., Suomela, J.: An optimal local approximation algorithm for max-min linear programs. In: Proc. 21st Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM Press (2009) To appear.
38. Floréen, P., Hassinen, M., Kaski, P., Suomela, J.: Tight local approximation results for max-min linear programs. In: Proc. 4th Workshop on Algorithmic Aspects of Wireless Sensor Networks (Algosensors). Volume 5389 of LNCS, Springer (2008) 2–17
39. Floréen, P., Kaski, P., Musto, T., Suomela, J.: Approximating max-min linear programs with local algorithms. In: Proc. 22nd International Parallel and Distributed Processing Symposium (IPDPS), IEEE (2008)
40. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, Inc., Mineola, NY, USA (1998)
41. Vazirani, V.V.: Approximation Algorithms. Springer (2001)
42. Kuhn, F., Moscibroda, T., Wattenhofer, R.: Fault-tolerant clustering in ad hoc and sensor networks. In: Proc. 26th International Conference on Distributed Computing Systems (ICDCS), IEEE Computer Society Press (2006)
43. Hochbaum, D.S.: Approximation algorithms for the set covering and vertex cover problems. SIAM Journal on Computing **11**(3) (1982) 555–556
44. Åstrand, M., Floréen, P., Polishchuk, V., Rybicki, J., Suomela, J., Uitto, J.: A local 2-approximation algorithm for the vertex cover problem. In: Proc. 23rd Symposium on Distributed Computing (DISC), Springer (2009) To appear.
45. Czygrinow, A., Hańćkowiak, M., Wawrzyniak, W.: Fast distributed approximations in planar graphs. In: Proc. 22nd Symposium on Distributed Computing (DISC). Volume 5218 of LNCS, Springer (2008) 78–92
46. Lenzen, C., Wattenhofer, R.: Leveraging Linial's locality limit. In: Proc. 22nd Symposium on Distributed Computing (DISC). Volume 5218 of LNCS, Springer (2008) 394–407
47. Lenzen, C., Oswald, Y.A., Wattenhofer, R.: What can be approximated locally? In: Proc. 20th Symposium on Parallelism in Algorithms and Architectures (SPAA), ACM Press (2008) 46–54
48. Elkin, M.: Distributed approximation: a survey. ACM SIGACT News **35**(4) (2004) 40–57
49. Sterling, A.: Self-assembling systems are distributed systems. Manuscript, arXiv:0907.1072 [cs.DC] (July 2009)