Asynchronous Fault-Tolerant Distributed Proper Coloring of Graphs

Alkida Balliu
Gran Sasso Science Institute, L'Aquila, Italy

Pierre Fraigniaud
IRIF - CNRS & Univ. Paris Cité, France

Patrick Lambein-Monette
Unaffiliated

Dennis Olivetti
Gran Sasso Science Institute, L'Aquila, Italy

Mikaël Rabie
IRIF - Université Paris Cité, France

Abstract

We revisit *asynchronous* computing in networks of *crash-prone* processes, under the asynchronous variant of the standard LOCAL model, recently introduced by Fraigniaud et al. [DISC 2022]. We focus on the vertex coloring problem, and our contributions concern both lower and upper bounds for this problem.

On the upper bound side, we design an algorithm tolerating an arbitrarily large number of crash failures that computes an $O(\Delta^2)$-coloring of any n-node graph of maximum degree Δ, in $O(\log^\star n)$ rounds. This extends Linial's seminal result from the (synchronous failure-free) LOCAL model to its asynchronous crash-prone variant. Then, by allowing a dependency on Δ on the runtime, we show that we can reduce the colors to $\left(\frac{1}{2}(\Delta+1)(\Delta+2)-1\right)$. For cycles (i.e., for $\Delta=2$), our algorithm achieves a 5-coloring of any n-node cycle, in $O(\log^\star n)$ rounds. This improves the known 6-coloring algorithm by Fraigniaud et al., and fixes a bug in their algorithm, which was erroneously claimed to produce a 5-coloring.

On the lower bound side, we show that, for $k<5$, and for every prime integer n, no algorithm can k-color the n-node cycle in the asynchronous crash-prone variant of LOCAL, independently from the round-complexities of the algorithms. This lower bound is obtained by reduction from an original extension of the impossibility of solving *weak symmetry-breaking* in the wait-free shared-memory model. We show that this impossibility still holds even if the processes are provided with inputs susceptible to help breaking symmetry.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases LOCAL model, Graph Coloring, Renaming, Weak Symmetry-Breaking, Fault-Tolerance, Wait-Free Computing

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.5

Related Version *Full Version*: https://arxiv.org/abs/2408.10971

Funding Partially funded by MUR (Italy) Department of Excellence 2023 - 2027, the PNRR MIUR research project GAMING "Graph Algorithms and MinING for Green agents" (PE0000013, CUP D13C24000430001), and by the French ANR projects DUCAT (ANR-20-CE48-0006) and QuDATA (ANR-18-CE47-0010).

1 Introduction

1.1 Asynchrony, Failures, and Networks

To what extent a global solution to a computational problem can be obtained from locally available data? What can be computed locally? These are some of the questions that were asked, and partially answered 30 years ago in two seminal papers [23, 25] in the field of distributed network computing. Since then, tremendous progress has been made about these questions, and even detailed books [22, 27] can only touch a small fraction of the content of the current literature on this topic. Nevertheless, the vast majority of the achievements on *local computing* have been obtained in *synchronous failure-free* models, among which the most common ones are referred to as LOCAL [23] and CONGEST [27].

In both models, processing nodes occupy the vertices of a graph, and exchange messages along the edges of that graph. They all start at the same time, and computing proceeds as a sequence of synchronous rounds. At each round, every pair of adjacent nodes can exchange messages (one in each direction), and every node can perform some individual computation. CONGEST differs from LOCAL only as far as the message size is concerned: messages are bounded to be of size at most B bits in CONGEST (it is common to set $B = O(\log n)$). There are at least two solid reasons why such elegant but simplistic models should be considered. First, they ideally capture the notion of *spatial locality*, as algorithms performing in t rounds produce an output at each node that is solely based on the t-neighborhood of the node. Second, the existence of efficient synchronizers [3, 4, 19] enables to implement algorithms designed for synchronous models on asynchronous networks, with only limited slowdown.

Yet, models such as LOCAL and CONGEST suffer from one notable limitation: they ignore the potential presence of failures. Indeed, transient failures have been addressed in the framework of *self-stabilization*, but crash or malign failures are mostly ignored in the framework of local computing in networks. Instead, studying the interplay of asynchrony and failures has been the main topic of interest of distributed computing in general [2, 24, 28], since the seminal "FLP impossibility result" stating that consensus is impossible in asynchronous systems with failures, even under the restriction that at most one crash failure may occur [15]. However, the design of algorithms dedicated to asynchronous crash-prone systems have been mostly performed in *shared-memory* or *message-passing* models: the former assumes that processes exchange information by writing and reading in a shared memory; the latter assumes that any two processes can exchange messages directly along a private channel. While these two models are excellent abstractions of very many types of distributed systems, ranging from multi-core architectures to large-scale computing platforms, they do not enable the study of spatial locality, as the structure of the physical network is abstracted away.

An attempt to resolve this tension between synchronous failure-free computing in networks, and asynchronous computing in crash-prone systems has been recently proposed [16], by considering asynchronous networks subject to crash failures.

1.2 The ASYNC LOCAL Model

The asynchronous crash-prone LOCAL model[1] (ASYNC LOCAL in short), introduced in [16], aims at capturing a setting that is a hybrid between shared memory and network computing.

[1] One could also consider the variant ASYNC CONGEST of ASYNC LOCAL by limiting to $O(\log n)$ bits the size of the registers in which nodes read and write, but we restrict ourselves to the LOCAL variant, as standard wait-free computing does not generally restrict the size of the registers.

This model can be described conceptually in two possible ways (see Section 2.1 for more details):

- The ASYNC LOCAL model can be viewed as the standard wait-free shared-memory model [2, 21] in which the read-access to other process's registers is restricted. It bears similarities with the *atomic state* model in self-stabilization [9]. In an n-process system, each process $i \in [n]$ can solely read the registers of processes $j \in N_G(i)$, where $N_G(i)$ denotes the set of neighbors of vertex i in a graph G. That is, the wait-free shared-memory model is the ASYNC LOCAL model in which the graph G is fixed to be the complete graph (or clique) K_n.
- The ASYNC LOCAL model can alternatively be viewed as the standard LOCAL model [22, 27] in which each node writes in its local register(s) instead of sending messages, and reads the registers of its neighbors instead of receiving messages from them. In addition, ASYNC LOCAL allows asynchronous executions, that is, each process reads and writes at its own pace, which may vary with time, and it may even crash (i.e., stop functioning, and never recover). Note that, as for LOCAL, the graph G is unknown to the nodes in ASYNC LOCAL, as it is typically the *input* to the problems of interest in network computing.

In the framework of asynchronous computing, the computing elements are referred to as *processes*, whereas they are referred to as *nodes* in the context of computing in networks, but we use these two terms indistinctly. The terminology "wait-free" refers to the fact that (1) an arbitrarily large number of processes can crash, and (2) a node cannot distinguish whether a neighboring node has crashed or is simply slow, from which it follows that a node must never "wait" for some action performed by another node, and must terminate independently from which of the other nodes have crashed (unless itself has crashed).

It was shown in [16] that the computing power of ASYNC LOCAL is radically different from the one of LOCAL. Indeed, the authors proved that constructing a maximal independent set (MIS) is simply *impossible* in ASYNC LOCAL, even in the n-node cycles C_n, $n \geq 3$, while, on cycles, it just takes $\Theta(\log^\star n)$ rounds in LOCAL [12, 23]. However, the authors show also that proper coloring C_n is possible in ASYNC LOCAL, to the expense of using a larger palette of colors, i.e., 6 colors instead of just 3 as in LOCAL (a 5-coloring algorithm is also claimed in [16], but, as we shall show later, there is a bug in that algorithm). Indeed, a simple reduction to *renaming* (see [2] for the definition) shows that, under the ASYNC LOCAL model, no algorithms can proper color all graphs of maximum degree Δ using less than $2\Delta + 1$ colors whenever $\Delta + 1$ is a power of a prime. This is because ASYNC LOCAL and standard shared-memory coincides when the graph is a clique of $n = \Delta + 1$ nodes. The main result in [16] is a distributed asynchronous algorithm in the ASYNC LOCAL model that achieves proper 6-coloring of any n-node cycle, $n \geq 3$, in $O(\log^\star n)$ rounds, which is optimal thanks to [23]. In ASYNC LOCAL, the *round-complexity* of an algorithm is the maximum, taken over all nodes, and all executions, of the number of times a node writes in its register, and reads the registers of its neighbors.

1.3 Our results

In a nutshell, we show that there exists an algorithm for proper coloring graphs with maximum degree Δ in the ASYNC LOCAL model, using a palette of $\frac{1}{2}(\Delta+1)(\Delta+2)-1$ colors, resulting into a 5-coloring algorithm for the cycles. This result was obtained by first showing how to implement Linial's coloring algorithm in the asynchronous setting, and then by developing a new technique based on reallocating identifiers to nodes. Note that even implementing

Linial's coloring algorithm asynchronously is not straightforward, as it is not even clear whether the trivial recoloring algorithm that proceeds iteratively over all color classes can be implemented in the ASYNC LOCAL model. Moreover, we show that, for infinitely many values of n, 5-coloring the n-node cycles is the best that can be achieved in ASYNC LOCAL. This significantly improves the lower bound in [16] on the number of colors required for proper coloring cycles under ASYNC LOCAL, which held for $n = 3$ only.

Obtaining our lower bound required to revisit entirely the known lower bound on weak symmetry breaking[2] in the standard asynchronous shared-memory model, by considering the impact of a priori "knowledge" given to the processes. For instance, if the processes know a priori that one process is given advice 0, and one process is given advice 1, then weak symmetry breaking becomes trivially solvable. For which a priori knowledge weak symmetry breaking becomes trivially solvable, and for which it remains unsolvable? We show that answering this novel question for specific types of a priori knowledge results into new impossibility results for the standard asynchronous shared-memory model, which translate into lower bounds and impossibility results in the ASYNC LOCAL model.

We stress the fact that while all (Turing computable) tasks are solvable in the LOCAL model, not all taks are solvable in ASYNC LOCAL, yet we also address *complexity* issues, by showing that, for constant Δ, our $\big(\frac{1}{2}(\Delta+1)(\Delta+2)-1\big)$-coloring algorithm performs in $O(\log^\star n)$ rounds in ASYNC LOCAL, that is, as fast as the $\Omega(\log^\star n)$ lower bound [23] on the number of rounds required for coloring cycles in the synchronous failure-free LOCAL model. These results are detailed next.

1.3.1 Proper Coloring

We mostly focus on distributed proper coloring, arguably one of the most important and thoroughly studied symmetry-breaking tasks in network computing – see, e.g., [17, 18, 20] for recent results on the matter[3]. First, we show that Linial's technique from [23] based on cover-free families of set systems can be used asynchronously, for the design of an $O(\Delta^2)$-coloring of graphs of maximum degree Δ, running in $O(\log^\star n)$ rounds in n-node graphs under ASYNC LOCAL. Then we show that the approach from [16] for 6-coloring cycles can be generalized to color arbitrary graphs. Specifically, we design an algorithm computing a $\frac{(\Delta+1)(\Delta+2)}{2}$-coloring in graphs of maximum degree Δ running in $O(\log^\star n)+f(\Delta)$ rounds under ASYNC LOCAL, where the additional term $f(\Delta)$ depends on Δ only. This line of results culminates in the design of an algorithm enabling to save one color, i.e., that computes a $\big(\frac{(\Delta+1)(\Delta+2)}{2}-1\big)$-coloring, still running in $O(\log^\star n)+f(\Delta)$ for some function f. Reducing the color palette by just one color may seem of little importance, but it is not, for two reasons. First, a palette of size $\frac{(\Delta+1)(\Delta+2)}{2}-1$ is the best that we are aware of for which it is possible to proper color all graphs of maximum degree Δ in $O(\log^\star n)$ rounds in ASYNC LOCAL (ignoring the additional term depending on Δ only). Saving one more color appears to be challenging. Second, in the case of cycles, i.e., $\Delta = 2$, this allows us to fix a bug in the 5-coloring algorithm from [16]. Indeed, this latter algorithm is shown to be erroneous, as

[2] Weak symmetry breaking is the task in which processes start with no inputs, and each process must output 0 or 1, under the constraint that, whenever all processes terminate, at least one process must output 0, and at least one process must output 1.

[3] In the context of distributed computing in networks, especially in the LOCAL and CONGEST models, one is interested in properly coloring graphs with maximum degree Δ using a palette of $f(\Delta)$ colors, where $f(\Delta)$ grows slowly with Δ. One typical example is $f(\Delta) = \Delta + 1$ as all graphs of maximum degree Δ can be properly colored with $\Delta + 1$ colors, but one is also interested in larger functions f, e.g., $f(\Delta) = \Theta(\Delta^2)$, whenever this choice enables to obtain faster algorithms.

there are schedulings of the nodes that result in livelocks preventing the algorithm from terminating. Nevertheless, our algorithm shows that 5-coloring the n-node cycles in $O(\log^\star n)$ rounds under ASYNC LOCAL is indeed possible.

1.3.2 Lower Bounds and Impossibility Results

Our second line of contribution is related to lower bounds on the size of the color palette enabling to proper color graphs asynchronously. It was observed in [16] that since the class of graphs with maximum degree Δ includes the clique with $n = \Delta + 1$ nodes, and since *renaming* [2] in a set of less than $2N - 1$ names cannot be done wait-free in N-process shared-memory systems whenever N is a power of a prime, proper coloring graphs of maximum degree Δ in ASYNC LOCAL cannot be achieved with a color palette smaller than $2\Delta + 1$ colors, i.e., 5 colors in the case of cycles (independently from the number of rounds). However, the question of whether one can 4- or even 3-color long cycles (i.e., excluding the specific case of the clique C_3) under ASYNC LOCAL was left open in [16]. We show that this is impossible whenever n is prime, that is, there are infinitely many values of n for which 5-coloring the n-node cycle is the best that can be achieved in ASYNC LOCAL.

1.3.3 Reduction from Weak Symmetry-Breaking with Inputs

We achieve our lower bound on the number of colors thanks to a result of independent interest in the standard framework of wait-free shared-memory computing. We show that there are no symmetric wait-free algorithms solving *weak symmetry-breaking* [2] in n-process asynchronous shared-memory systems whenever n is prime, *even if processes are provided with inputs from a non-prime-divisible and order-invariant set of inputs*. We achieve this impossibility result by extending the proof in [1] for weak symmetry-breaking to the case in which processes have inputs that do not trivially break symmetry. Our impossibility result for weak symmetry-breaking with inputs has other consequences on the ASYNC LOCAL model, including the facts that weak 2-coloring is impossible in cycles of prime size, and that, for every even $\Delta \geq 2$, there is an infinite family of regular graphs for which $(\Delta + 2)$-coloring cannot be solved in ASYNC LOCAL.

Finally, using different techniques, we also show that even a weak variant of maximal independent set (MIS) cannot be solved in cycles with at least 7 nodes, and that, for every $\Delta \geq 2$, $(\Delta + 1)$-coloring trees of maximum degree Δ is impossible under ASYNC LOCAL.

1.4 Related Work

The combination of asynchrony *and* failures in the general framework of distributed computing in networks has been studied a lot in the context of *self-stabilization*. The latter deals with *transient* failures susceptible to modify the content of some of the variables defining the states of the nodes. The role of a self-stabilizing algorithm is therefore to guarantee that if the network is in an illegal configuration (i.e., a configuration not satisfying some specific correctness condition), then it will automatically return to a legal configuration, and will remain in a legal configuration, unless some other failure(s) occur. Self-stabilizing graph coloring algorithms have been designed [5, 6, 7, 8]. However, these algorithms provide solutions only for executions during which there are no failures. Instead, in ASYNC LOCAL, failures may occur at any time during the execution, and once a process crashes it never recovers. This has important consequences on what can or cannot be computed in ASYNC LOCAL. For instance, 3-coloring the n-node cycle is possible in a self-stabilizing manner for every $n \geq 3$, while we show that even 4-coloring the n-node cycle is impossible for infinitely many n (namely, for all prime n).

It is also worth mentioning [11, 13], which introduced the DECOUPLED model, where crash-prone processes occupy the nodes of a *reliable* and *synchronous* network. The DECOUPLED model is stronger than ASYNC LOCAL, and indeed it was shown that if there exists an algorithm solving a task in the LOCAL model, then there exists an algorithm solving that task in the DECOUPLED model as well, with limited slowdown. Instead, we show that even a weak variant of MIS is impossible in large cycles under ASYNC LOCAL.

Another field of research very much related to our work is the study of *synchronous* networks with failures, whether it be crash or even malicious process failures, or message omission failures (see, e.g., [10, 26, 29, 30]). In these models, the focus has mostly been put on the study of tasks such as consensus and set-agreement. The ASYNC LOCAL model somehow mixes some of the key aspects of the models considered in these work, including the presence of crash failures, and the fact that the communications are mediated by a graph distinct from the complete graph. The same way standard wait-free computing in shared-memory systems can be viewed as one specific instance of the oblivious message adversary model, wait-free computing in the ASYNC LOCAL model in a graph G may be viewed as the instance of the oblivious message adversary model in which messages can only be sent along the edges of the graph G. We however focus on solving graphs problems such as coloring or independent set, motivated by the need to solve various symmetry breaking problems in networks, including frequency assignment and cluster decomposition. For such problems, it is more more convenient to use the framework of ASYNC LOCAL, in which the graph G is part of the input, as in the LOCAL model.

2 Model and Definitions

We first recall the ASYNC LOCAL model as introduced in [16], and then provide an example for an algorithm in this model.

2.1 The ASYNC LOCAL model

Like the LOCAL model [27], the ASYNC LOCAL model assumes a set of $n \geq 1$ processes, each process occupying a distinct node of an n-vertex graph $G = (V, E)$, which is supposed to be simple and connected. Each process, i.e., each node $v \in V$, has an identifier id_v that is supposed to be unique in the graph. The identifiers are not necessarily between 1 and n, but they are supposed to be stored on $O(\log n)$ bits. That is, all node identifiers lie in the integer interval $[1, N]$ for some bound $N = \mathrm{poly}(n)$. Like in the asynchronous shared-memory model, every node v comes equipped with a single-writer/multiple-reader register $R(v)$ in which it can write values. However, in contrast with the shared-memory model, *only v's neighbors in the graph G* are able to read its register $R(v)$, and symmetrically, node v can only read the registers $R(w)$ of nodes $w \in N_G(v) = \{u \in V \mid \{u, v\} \in E\}$. We assume that each node can write in its register, and then read all its neighbors' registers, in a single atomic operation. Neighboring nodes can perform this write&read operation concurrently, in which case they both read the value concurrently written in the other node's register. This communication primitive is thus akin to an *immediate snapshot* object with read accesses mediated by a graph, in a similar manner to the *atomic state* model in the context of self-stabilizing algorithms [9]. Computation proceeds asynchronously, and each node may crash, in which case it stops functioning, and it never recovers. Therefore, in the particular case of the clique $G = K_n$, ASYNC LOCAL boils down to the standard asynchronous crash-prone shared-memory model with immediate snapshots [2]. The registers are of unbounded size. Therefore, as in the LOCAL model, and as in most wait-free computing models [21] as well, we can assume *full-information protocols*, in which every node writes its entire state in its register, and read the states of its neighbors in their registers.

Remark. Due to its nature, the ASYNC LOCAL model may have also been named "iterated immediate local snapshot". Nevertheless, for its close connection to the standard LOCAL model used for the study of graph problems (e.g., coloring) in distributed computing, we preferred to stick to the terminology ASYNC LOCAL.

Input. In addition to its identifier id_v, every node v may be provided with some input, denoted by input_v. The latter may be the number n of nodes in the graph, or an upper bound N on n, or any label $\ell(v) \in \{0,1\}^*$ whose semantic depends on the context (e.g., it may represent a boolean mark, or a color, etc.). Note that the network G is typically unknown to the nodes, even if some specific parameters may be provided to each node as input, such as the maximum degree Δ of G.

Algorithm. An algorithm $\mathcal{A}$ for the ASYNC LOCAL model may be described by two functions:

- Init: used to initialize the state of each node, as a function of its input;
- Alg: used to update the state of a node, as a function of its current state, and of the states of its neighbors.

Scheduling. An execution of an algorithm $\mathcal{A}$ depends on how the nodes are scheduled. A *scheduling* is a sequence $\mathcal{S} = S_1, S_2, \ldots$ of subsets $S_i \subseteq V$ of nodes. For every $i \geq 1$, the set S_i denotes the set of nodes that are activated at *step* i. Each of these nodes performs an immediate-snapshot, and updates its state accordingly. For instance, the scheduling $\{u,v\},\{v\},\{v\},\{v\},\ldots$ represents the execution in which nodes u and v run concurrently at the first step, and then v runs solo, i.e., v is the only node activated at every step $i \geq 2$. That is, u has crashed after step 1, and all the nodes $w \notin \{u,v\}$ had crashed initially, none of them taking any step. Instead, the scheduling $V, V, V, \ldots$ represents a synchronous execution in which no node crashes.

Full-Information Protocols. For every $v \in V$, let $\mathrm{OldState}_{v,1} \leftarrow \perp$, and $\mathrm{NewState}_{v,1} \leftarrow \mathrm{Init}(\mathrm{id}_v, \mathrm{input}_v)$. For every $i \geq 1$, the variable $\mathrm{OldState}_{v,i}$ represents what a neighbor of v gets whenever reading the memory of v, and $\mathrm{NewState}_{v,i}$ represents the updated state of v, which will become visible to its neighbors the next time v is scheduled. More specifically, for every $i \geq 1$, if $v \notin S_i$, then $\mathrm{OldState}_{v,i+1} \leftarrow \mathrm{OldState}_{v,i}$ and $\mathrm{NewState}_{v,i+1} \leftarrow \mathrm{NewState}_{v,i}$. Instead, if $v \in S_i$, then $\mathrm{OldState}_{v,i+1} \leftarrow \mathrm{NewState}_{v,i}$, and $\mathrm{NewState}_{v,i+1} \leftarrow \mathrm{Alg}(\mathrm{OldState}_{v,i+1}, \{\mathrm{OldState}_{u,i+1} \mid u \in N_G(v)\})$. In other words, all nodes that are scheduled at step i write their current state, then read the state of their neighbors, and then use the obtained knowledge in order to update their state. The new states resulting from these updates will become visible to their neighbors the next time that they are scheduled. That is, we model a setting in which writing and then reading the state of the neighbors is an atomic operation, but it may take some time to compute a new state.

Termination. We let $\mathrm{Terminated}(x)$ be a special state denoting that a node terminates with output x. If a node v satisfies $\mathrm{NewState}_{v,t} = \mathrm{Terminated}(x)$ at some step $t \geq 1$, then v decides the output x, and it is assumed that if v is scheduled again in the future, then its state does not change, that is, $\mathrm{NewState}_{v,t+i} = \mathrm{NewState}_{v,t}$ for all $i \geq 1$.

Algorithm 1 An algorithm for 6-coloring cycles. Code of node v, with sole input id_v.

```
procedure CycleSixColoring(id_v)
    x ← id_v;  a ← 0;  b ← 0;                          ▷ (x, a, b) is the state s of v
    repeat forever
        (s_1, s_2) ← WriteSnapshot(s)   ▷ s_1 and s_2 are the states of the two neighbors of v
        if (a, b) ∉ {(s_1.a, s_1.b), (s_2.a, s_2.b)} then return (a, b)
        else          ▷ In the following: s_i = ⊥ ⟹ (s_i.x = ⊥) ∧ (s_i.a = ⊥) ∧ (s_i.b = ⊥).
            a ← min ℕ ∖ {s_i.a | (i ∈ {1,2}) ∧ (s_i ≠ ⊥) ∧ (s_i.x > x)}
            b ← min ℕ ∖ {s_i.b | (i ∈ {1,2}) ∧ (s_i ≠ ⊥) ∧ (s_i.x < x)}
        end if
    end repeat
end procedure
```

Round complexity. The runtime of a node v is defined as

$$T_v = |\{i \geq 1 \mid v \in S_i \text{ and } \mathrm{NewState}_{v,i} \neq \mathrm{Terminated}(x) \text{ for any possible output } x\}|.$$

That is, the runtime of a v is equal to how many times v is scheduled before it terminates. The runtime of an algorithm on a graph $G = (V, E)$ is then $\max\{T_v \mid v \in V\}$. The runtime of an algorithm in a graph class $\mathcal{G}$ is the maximum runtime of the algorithm, over all graphs $G \in \mathcal{G}$. The runtime of an algorithm may depend on the identifiers given to the nodes. However, as said before, we use the standard assumption that the identifiers are from the interval $[1, N]$ where $N = \mathrm{poly}(n)$. The runtime is thus typically expressed as a function of n (the order of the graph) and Δ (the maximum degree of the graph). The complexity of a problem is the minimum runtime (as a function of n and Δ) among all possible algorithms that solve the problem. The typical graph class we are interested in is $\mathcal{G}_\Delta$, the class of all graphs with maximum degree Δ.

Remark. In absence of failures, and if all nodes run synchronously, the runtime of an algorithm in the ASYNC LOCAL model is identical to its runtime in the LOCAL model.

2.2 Algorithm Description

While an algorithm can be formally described by providing the two functions Init and Alg, we now describe an alternative, and possibly easier way of describing an algorithm. An example is provided in Algorithm 1 from [16], which is aiming at solving 6-coloring in cycles. This algorithm uses the function WriteSnapshot(s), which allows to perform an immediate snapshot (i.e., a write of the current state s immediately followed by a snapshot of all the states of the neighbors), and uses the function return, which explicitly provides the output (instead of using Terminated(x)).

In Algorithm 1, the state s of each (non terminated) node is a triplet $s = (x, a, b)$ of natural numbers. Given a state s, $s.x$, $s.a$, and $s.b$ respectively denote the first, second, and third element in s. The state of a terminated node is a pair (a, b) of natural numbers. One can check (see [16]) that the output pairs (a, b) can take at most 6 different values.

The state s of a node v is updated by updating some of all of its components x, a, or b. Actually, the entry $x = \mathrm{id}_v$ does not change. The entry a is updated to the smallest natural number excluding the a-values used by neighbors of larger identifiers, and b is updated to the smallest natural number excluding the b-values used by the neighbors of smaller identifiers. These values are equal to $\perp$ if they have not yet been written in the register (i.e., if a neighbor

has not yet performed a single write). If a node v notices that its current state (x, a, b) is such that (a, b) is different from the (a, b)-pairs of both neighbors, then v terminates, and decides color (a, b). An example of an execution of Algorithm 1 is provided in Appendix A.

3 Results and Road Map

We have now all ingredients sufficient to formally state our results.

3.1 Algorithms for ASYNC LOCAL

We first show (cf. Section 4) that Linial's $O(\Delta^2)$-coloring algorithm can be adapted to work in the asynchronous wait-free setting.

▶ **Theorem 1.** *For every $\Delta \geq 2$, the round-complexity of $O(\Delta^2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n)$.*

Then, we show (cf. Section 5) that, at the cost of increasing the runtime by an additive factor depending on Δ, it is possible to reduce the number of colors from $O(\Delta^2)$ to $(\Delta + 1)(\Delta + 2)/2$.

▶ **Theorem 2.** *For every $\Delta \geq 2$, the round-complexity of $\frac{1}{2}(\Delta + 1)(\Delta + 2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, where f is a function depending on Δ only.*

Finally, we show (cf. Section 6) that we can exploit the fact that the coloring produced by Theorem 2 satisfies special properties for reducing the size of the color palette by one.

▶ **Theorem 3.** *For every $\Delta \geq 2$, the round-complexity of $(\frac{1}{2}(\Delta + 1)(\Delta + 2) - 1)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, for some function f that only depends on Δ.*

An important consequence of this result is the case $\Delta = 2$. Theorem 3 shows that there is an algorithm for 5-coloring cycles. While such an algorithm was already claimed to exist in [16], we show (cf. Appendix B) that the algorithm supporting that claim is erroneous. Specifically, we provide an instance in which the algorithm does not terminate. Theorem 3 provides a novel algorithm, which allows us to establish the following result.

▶ **Corollary 4.** *The round-complexity of 5-coloring cycles in the ASYNC LOCAL model is $O(\log^* n)$.*

3.2 Impossibility Results

As pointed out in [16] several impossibility results for ASYNC LOCAL are mere consequences of the fact the this model coincides with the standard wait-free shared-memory model whenever the underlying graph G is a clique K_n. This is for instance the case of the impossibility of 4-coloring C_3 (by reduction from renaming), and the impossibility of constructing a maximal independent set, i.e., MIS (by reduction from strong symmetry breaking). Whether or not it is possible to 4-color cycles C_n for $n > 3$ was left open in [16]. We show that, for infinitely many values of n, the problem of 4-coloring the n-node cycle C_n is not solvable in ASYNC LOCAL. To establish this result, we prove a result of independent interest, in the framework of wait-free shared memory computing. Specifically, we extend the proof in [1] that weak symmetry breaking is impossible in the wait-free shared memory systems. We show

that this problem remains impossible even if some input are provided to the processes, which may potentially help them to break symmetry. The set of possible inputs has to agree with some restrictions, called non-prime-divisible and order-invariant (with respect to a particular subset of processes). Roughly, the set of possible input assignments must not be divisible by the number n of processes whenever n is prime, and it must be closed under permuting the identifiers of a particular subset of the processes by an order-invariant permutation. Also recall that an algorithm is *symmetric* if for every execution α on a subset P of processes, and for every permutation $\pi : [n] \to [n]$ order preserving on P, we have that, for every $i \in P$, process i outputs x in α if and only if process $\pi(i)$ outputs x on the execution $\pi(\alpha)$ resulting from permuting the scheduling of the processes in P according to α. Our impossibility results are shown in the full version.

▶ **Theorem 5.** *Let n be a prime number. There are no symmetric wait-free deterministic algorithms solving weak symmetry break in the asynchronous wait-free shared memory model with n processes, even if the processes are provided with inputs from a non-prime-divisible and order-invariant set of inputs.*

Theorem 5 has three important consequences.

▶ **Corollary 6.** *Let $n \geq 3$ be a prime number. The problem of 4-coloring the n-node cycle cannot be solved deterministically in ASYNC LOCAL.*

A weaker form of symmetry breaking is weak 2-coloring [25]. It is required to 2-color the input graph such that every (non isolated) node has at least one neighbor colored with a different color.

▶ **Corollary 7.** *Let $n \geq 3$ be a prime number. The problem of weak 2-coloring the n-node cycle cannot be solved deterministically in ASYNC LOCAL.*

Finally, we prove that, for even values of Δ, there are a infinitely many Δ-regular graphs that cannot be $(\Delta + 2)$-colored in ASYNC LOCAL. This extends the lower bound of $2\Delta + 1$ colors, which applies only for the clique of $\Delta + 1$ nodes with $\Delta + 1$ power of a prime, to an infinite family of graphs with maximum degree Δ.

▶ **Corollary 8.** *Let Δ be an even number, and let $n > \Delta$ be a prime number. The problem of $(\Delta+2)$-coloring n-node Δ-regular graphs cannot be solved deterministically in ASYNC LOCAL.*

We complete the lower bound analysis with some additional results. The version of MIS considered in [16], which was proved impossible to solve, asks the nodes to output a set of vertices which forms an MIS in the graph induced by the *correct* nodes. Instead, we consider a weaker variant of MIS, asking the nodes to output a set of vertices which forms an MIS in the graph whenever all processes are correct, i.e., no crashes occurred. We show that even this weaker variant of MIS is impossible in ASYNC LOCAL.

▶ **Theorem 9.** *For every $n \geq 7$, no deterministic algorithms can solve weak MIS in the n-node cycle under ASYNC LOCAL.*

Finally, we show impossibility results for coloring general graphs.

▶ **Theorem 10.** *For every $\Delta \geq 2$, no deterministic algorithms can solve $(\Delta + 1)$-coloring in trees of maximum degree Δ under ASYNC LOCAL.*

We conclude, in Section 7, with some open questions.

4 Coloring General Graphs with $O(\Delta^2)$ Colors

In this section, we provide a simple algorithm for coloring a graph with $O(\Delta^2)$ colors. This algorithm is an adaptation of Linial's coloring algorithm [23] (which is designed to work in the LOCAL model) to the asynchronous setting. More in detail, we prove the following result.

▶ **Theorem 1.** *For every $\Delta \geq 2$, the round-complexity of $O(\Delta^2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n)$.*

In order to prove this result, we start by summarizing Linial's coloring algorithm, and then we show how to adapt it to the wait-free setting. We start by recalling the notion of set systems and of cover-free family of sets.

▶ **Definition 11.** *A* set system *is a pair $(X, \mathcal{F})$, where X is a set, and $\mathcal{F}$ is a collection of subsets of X. A set system $(X, \mathcal{F})$ is a* k-cover-free *family if, for every choice of $k+1$ distinct sets $S_0, S_1, \ldots, S_k$ in $\mathcal{F}$, the following holds: $S_0 \setminus \bigcup_{i=1}^{k} S_i \neq \varnothing$.*

To provide an intuition about how to use these two definitions, let us assume that the nodes of the input graph G are properly c-colored, and let us assume that there exists a Δ-cover-free family $(X, \mathcal{F})$ satisfying $c \leq |\mathcal{F}|$. It follows from these assumptions that there exists a one-to-one function f from the set of colors to $\mathcal{F}$. W.l.o.g., assume that X contains the numbers in $\{1, \ldots, |X|\}$. One step of Linial's algorithm is able to recolor the nodes with $c' = |X|$ colors, as follows.

1. Every node v communicates with its d neighbors to get their current colors $c_1, \ldots, c_d$, where $d \leq \Delta$ is the degree of v.
2. Every node v computes $X_v = f(c_v) \setminus \bigcup_{i=1}^{d} f(c_i)$, where c_v is the color of v, and then recolors itself with the minimum value in X_v.

Note that X_v is guaranteed to be non-empty by the fact that $(X, \mathcal{F})$ is a Δ-cover-free family, and that the obtained color c'_v satisfies $1 \leq c'_v \leq c'$. Linial's coloring algorithm repeats this process multiple times, each time using a different cover-free family. The runtime and the resulting number of colors depend on the choice of cover-free families. We summarize the cover-free families used by Linial's algorithm in the following two lemma.

▶ **Lemma 12** ([23])**.** **(a)** *For any $c > \Delta$, there exists a Δ-cover-free family $(X, \mathcal{F})$ with $c \leq |\mathcal{F}|$, and $|X| \leq 5\lceil \Delta^2 \log c \rceil$.* **(b)** *There exists a Δ-cover-free family $(X, \mathcal{F})$ with $10\Delta^3 \leq |\mathcal{F}|$, and $|X| \leq (4\Delta + 1)^2$.*

In [23], Lemma 12 has been proved in a non-constructive way. However, it is possible to obtain a similar statement by using polynomials over finite fields [14]. We will use the above lemma as a black-box. However, the correctness of our algorithm will be independent from which specific cover-free family construction is used.

We now discuss how these cover-free families are used. Linial's algorithm, in its standard formulation for LOCAL, requires the nodes to be aware of an upper bound N on the size of the identifier space. At the first round, nodes recolor themselves by using $5\lceil \Delta^2 \log N \rceil$ colors, thanks to a cover-free family from Lemma 12(a) with parameter $c = N$. We denote by f_1 the one-to-one function used by the nodes to map their color to the elements of the cover-free family. At the second round, nodes use the cover-free family from Lemma 12(a) with parameter $c = 5\lceil \Delta^2 \log N \rceil$, from which they obtain a coloring that uses $5\lceil \Delta^2 \log(5\lceil \Delta^2 \log N \rceil) \rceil$ colors. We denote by f_2 the one-to-one function used by the nodes to map their color to the elements of the cover-free family. The nodes repeat this process multiple times, each time using a cover-free family from Lemma 12(a) with parameter c equal to the amount of colors obtained

Algorithm 2 $O(\Delta^2)$-coloring arbitrary graph. Code of node v: $\mathrm{id}_v \in \{1,\dots,N\}$; $\mathrm{input}_v = N$.

```
1: procedure WaitFreeLinial(id_v, input_v)
2:     S ← (id_v, ⊥, ... ⊥); ▷ S is an array of length T + 1 = O(log* N), and is the state s of v
3:     for i = 1 to T do
4:         (s_1, ..., s_d) ← WriteSnapshot(s)
5:         A ← {s_j.S[i] | (j ∈ {1, ..., d}) ∧ (s_j.S[i] ≠ ⊥)}      ▷ ith entry of each array s_j.S
6:         S[i + 1] ← min f_i(S[i]) ∖ ⋃_{a∈A} f_i(a)
7:     end for
8:     return s[T + 1]
9: end procedure
```

in the previous rounds. Linial proved that it takes $O(\log^* N)$ rounds to reach a coloring that uses at most $10\Delta^3$ colors. Since it is typically assumed that $N = \mathrm{poly}(n)$, the runtime is $O(\log^* n)$. At this point, the cover-free family from Lemma 12(b) is used to get a coloring that uses $(4\Delta+1)^2 = O(\Delta^2)$ colors.

Let us denote by T the number of rounds performed in total, including the last round that uses the family from Lemma 12(b) for reducing the number of colors to at most $(4\Delta+1)^2$. For $1 \le i \le T$, let f_i be the one-to-one function used by the nodes to map their colors to the elements of the cover-free family while executing the ith round of Linial's algorithm.

The Algorithm. Let us show that the approach used in Linial's LOCAL algorithm can be adapted to work in ASYNC LOCAL as well. We assume that input_v contains the same upper bound N on the range of identifiers. So, in particular, every node v can compute T as a function of input_v. The adaptation of Linial's coloring algorithm to ASYNC LOCAL is displayed as Algorithm 2. The main challenge when running Linial's algorithm in the ASYNC LOCAL model comes from the fact that a vertex v may be in the ith iteration of Linial's algorithm, while a neighbor u of v may be in iteration $j \neq i$. Nevertheless, we will prove that our adaptation of Linial's algorithm correctly handles these cases. The runtime of Algorithm 2 is clearly $O(\log^* n)$. The proof that Algorithm 2 is correct can be found in the full version.

5 Reducing the Colors to $(\Delta+1)(\Delta+2)/2$

In this section, we show that, at the cost of increasing the running time by an additive factor depending on Δ only, we can decrease the amount of colors from $O(\Delta^2)$ to $\frac{1}{2}(\Delta+1)(\Delta+2)$.

▶ **Theorem 2.** *For every $\Delta \ge 2$, the round-complexity of $\frac{1}{2}(\Delta+1)(\Delta+2)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, where f is a function depending on Δ only.*

The algorithm that we provide is a generalization to general graphs of the 6-coloring algorithm for cycles presented in [16], and restated in Algorithm 1. On a high-level, the algorithms works as follows. First, we compute an initial $O(\Delta^2)$-coloring of the nodes. Then, the final color of each node is given by a pair (a,b). This pair is computed by repeatedly updating the values of a and b until the pair is different from the pairs of the neighbors. The value of a is updated as a function of the a-values of the neighbors with larger initial color, while the value of b is updated as a function of the b-values of the neighbors with smaller initial color.

Algorithm 3 Reducing the number of colors from $O(\Delta^2)$ to $(\Delta+1)(\Delta+2)/2$.

```
1: procedure SaveColors(id_v,input_v)
2:     x ← input_v; (a,b) ← (0,0)                    ▷ x ∈ [O(Δ²)] is the original color of v
3:     repeat forever                                  ▷ s = (x,a,b) is the state of v
4:         (s_1,...,s_d) ← WriteSnapshot(s)
5:         if (a,b) ∉ {(s_i.a, s_i.b) | (i ∈ {1,...,d_v}) ∧ (s_i ≠ ⊥)} then return (a,b)
6:         else
7:             a ← min ℕ ∖ {s_i.a | (i ∈ {1,...,d_v}) ∧ (s_i ≠ ⊥) ∧ (x < s_i.x)}
8:             b ← min ℕ ∖ {s_i.b | (i ∈ {1,...,d_v}) ∧ (s_i ≠ ⊥) ∧ (x > s_i.x)}
9:         end if
10:    end repeat
11: end procedure
```

The algorithm. In order to prove Theorem 2, we first analyze the algorithm SaveColors, displayed as Algorithm 3. Given an $O(\Delta^2)$-coloring as input, this procedure produces a $((\Delta+1)(\Delta+2)/2)$-coloring, in $f(\Delta)$ rounds for some function f. Theorem 2 follows by running Algorithm WaitFreeLinialReduced below, in which if a node v is running SaveColors while some neighbor u of v is still running WaitFreeLinial, then v treats the memory of u as $\perp$.

```
procedure WaitFreeLinialReduced(id_v,input_v)
    c_v ← WaitFreeLinial(id_v, input_v)
    return SaveColors(id_v, c_v)
end procedure
```

The proofs of correctness and runtime of Algorithm 3 can be found in the full version of the paper.

6 Saving One More Color

We now modify Algorithm 3 in order to save one additional color. This new algorithm, shown in Algorithm 4, allows us to establish the following theorem.

▶ **Theorem 3.** *For every $\Delta \geq 2$, the round-complexity of $(\frac{1}{2}(\Delta+1)(\Delta+2)-1)$-coloring graphs of maximum degree Δ in the ASYNC LOCAL model is $O(\log^* n) + f(\Delta)$, for some function f that only depends on Δ.*

An important consequence of this result is Corollary 4, that is, the existence of a 5-coloring algorithm for the cycles in the ASYNC LOCAL model. This result is original because the 5-coloring algorithm proposed in [16] has a bug (cf. Appendix B where we exhibit an instance of 5-coloring C_4 for which the algorithm in [16] does not terminate).

6.1 Intuition of the algorithm

We start by providing the high level idea of the algorithm. The algorithm that we provide is similar to Algorithm 3, and it exploits some special properties of the pairs (a,b) that it produces. Specifically, we modify Algorithm 3 such that, if a node outputs the pair $(\Delta,0)$, then none of its neighbors output the pair $(0,\Delta)$. In this case, we can identify the pairs $(\Delta,0)$ and $(0,\Delta)$ as the same color, reducing the amount of colors in use by one.

Notice that a node that outputs the pair $(\Delta,0)$ is necessarily a local minimum with respect to the node identifiers, and similarly a node that outputs the pair $(0,\Delta)$ is necessarily a local maximum. The problematic case of neighbors outputting both pairs $(\Delta,0)$ and $(0,\Delta)$

can therefore only happen when the both neighbors are local extrema. However, for such neighboring nodes to reach a state where they would output problematic pairs, some specific conditions must hold which can be handled by the nodes as a specific case.

More in detail, in such a situation, we make nodes flip their relative ordering: if node u is a local minimum and node v is a local maximum, then u will treat v as smaller when comparing their x variables, and v will treat u as larger. By flipping relative ordering, we are forcing neighboring local extrema with pairs $(\Delta, 0)$ and $(0, \Delta)$ to stop being local extrema, leading them to change their output pairs. This modification will affect the termination time, and hence we also need to introduce new terminating conditions.

6.2 Formal Description

The algorithm is displayed in Algorithm 4, but some of its functions are presented below.

Treating special pairs as equal. The first modification applied to Algorithm 3 is the following. In line 5, instead of directly using the pairs (a, b) of the node, and the pairs of its neighbors, we first map them by using the function MAP shown below. Observe that MAP behaves as the identity function for all pairs different from $(\Delta, 0)$, and it maps $(\Delta, 0)$ to $(0, \Delta)$. In this way, the algorithm behaves similarly as the original one, except that it forbids neighboring nodes with pairs $(0, \Delta)$ and $(\Delta, 0)$ to terminate, since after applying MAP, they are both mapped to $(0, \Delta)$, and hence they are treated as having the same pair.

procedure MAP(a,b)
  if $(a, b) = (\Delta, 0)$ **then return** $(0, \Delta)$
  else return (a, b)
  end if
end procedure

A new ordering relation. In Algorithm 3, nodes exploit their variables x (that is, the given coloring) to determine an ordering relation between them. In the new algorithm, each node keeps an additional variable f, which is a set of identifiers. The semantic is the following. For two nodes u and v, if $u \in v.f$ or $v \in u.f$, then the ordering w.r.t. their variables x is flipped. We call an edge $\{u, v\}$ flipped whenever $u \in v.f$ or $v \in u.f$.

Let us define two auxiliary Boolean functions that are used by a node v to determine whether the ordering relation with a neighbor u should be considered flipped or not. These functions take as input the state s_v and s_u of the two (neighboring) nodes. The variable z, as will be shown in the algorithm, stores the identifier of the node.

procedure ISNOTFLIPPED(s_v, s_u)
  return $(s_v \neq \perp) \wedge (s_u \neq \perp) \wedge (s_u.z \notin s_v.f) \wedge (s_v.z \notin s_u.f)$
end procedure
procedure ISFLIPPED(s_v, s_u)
  return $(s_v \neq \perp) \wedge (s_u \neq \perp) \wedge \big((s_u.z \in s_v.f) \vee (s_v.z \in s_u.f)\big)$
end procedure

We are now ready to define the new ordering relation. For this purpose, we define two functions that, given the state s of the node, and the state s_i of its ith neighbors, return the neighbors that are considered smaller, and the neighbors that are considered larger, respectively.

procedure SMALLER(s,$(s_1, \ldots, s_k)$)
return $\{i \in \{1, \ldots, k\} \mid ((\text{ISNOTFLIPPED}(s, s_i) \wedge (s.x > s_i.x)) \vee (\text{ISFLIPPED}(s, s_i) \wedge (s.x < s_i.x))\}$
end procedure
procedure LARGER(s,$(s_1, \ldots, s_k)$)
return $\{i \in \{1, \ldots, k\} \mid ((\text{ISNOTFLIPPED}(s, s_i) \wedge (s.x < s_i.x)) \vee (\text{ISFLIPPED}(s, s_i) \wedge (s.x > s_i.x))\}$
end procedure

Special termination. We also define a function that provides an extra termination condition. It relies on an additional function that detects a neighborhood with special properties. It uses some variables α and β that are both set to true if a node has at least one smaller neighbor (that is, it is not a local minima), and it has at least one larger neighbor (that is, it is not a local maxima). We assume that the maximum degree Δ is part of the input provided to the nodes.

procedure SPECIALNEIGHBORHOOD(s,$(s_1, \ldots, s_\Delta)$)
return $\Big(\big(\bigwedge_{i=1}^{\Delta} (s_i \neq \perp) \big) \wedge \big(\{s.a, s.b\} \cup (\cup_{i=1}^{\Delta} \{s_i.a, s_i.b\}) \subseteq \{0, \ldots, \Delta - 1\} \big) \wedge s.\alpha \wedge s.\beta \wedge \Big(\bigwedge_{i=1}^{\Delta} \big((s_i.\alpha \vee |\text{SMALLER}(s_i, [s])| = 1) \wedge (s_i.\beta \vee |\text{LARGER}(s_i, [s])| = 1) \big) \Big) \Big)$
end procedure

That is, a neighborhood of a node v is *special* if (1) node v has seen all its neighbors, (2) they are precisely Δ, (3) the a and b variables of the node and of all its neighbors are in $\{0, \ldots, \Delta - 1\}$, and (4) node v and all its neighbors have at least one smaller, and at least one larger neighbor. The reason why we use the condition $s_i.\alpha \vee |\text{SMALLER}(s_i, [s])| = 1$ for checking whether a node has at least one smaller neighbor, instead of just using $s_i.\alpha$ is the following. Let u be the node with state s_i, and v be the node with state s. It could be the case that v is smaller than u, but u has been scheduled earlier than v. So it may be the case that u has never seen v. In this case, we could get that $u.\alpha$ is false, even though u has v as smaller neighbor. For this reason, node v computes whether $u.\alpha$ would become true if u were to be scheduled one additional round, by checking whether v is smaller than u using the condition $|\text{SMALLER}(s_i, [s])| = 1$. A similar reasoning is applied for checking whether a node has at least one larger neighbor. Note that, in the algorithm, once a node sets α (resp. β) to true, that is when it realizes that it is not a local minima (resp., maxima), it will never change its value. The reason is that, as we will prove later, a node never becomes a local minima (resp., maxima) by flipping edges. We now introduce the special termination condition. According to this special condition, a node terminates if (1) its neighborhood is special, and (2) it is a local maxima according to the original ordering, that is, before flipping any edge.

procedure SPECIALTERMINATION(s,$(s_1, \ldots, s_\Delta)$)
return $\text{SPECIALNEIGHBORHOOD}(s, (s_1, \ldots, s_\Delta)) \wedge \big(\forall i \in \{1, \ldots, \Delta\}, s.x > s_i.x\big)$
end procedure

The new algorithm. The algorithm is displayed as Algorithm 4. Like in the case of Algorithm 3, we assume that input_v is the result of running WAITFREELINIAL. Observe that the algorithm is similar to Algorithm 3, with only three exceptions. First, it identifies $(0, \Delta)$ with $(\Delta, 0)$ when checking for termination at line 6. Second, it uses the custom ordering relation induced by the functions SMALLER and LARGER at lines 11 and 12. Third, it has an additional termination condition at line 17. The proof that Algorithm 4 is correct, and the analysis of its runtime can be found in the full version.

Algorithm 4 Saving 1 color from palette $[\frac{1}{2}(\Delta+1)(\Delta+2)]$. Algorithm of node v with color input_v.

```
1: procedure SaveOneMoreColor(id_v,input_v)
2:     a ← 0; b ← 0 ; x ← input_v ; z ← id_v
3:     f ← {}; α ← false; β ← false          ▷ s = (a, b, x, f, α, β, z) is the state of node v
4:     repeat forever
5:         (s_1, ..., s_Δ) ← WriteSnapshot(s)          ▷ if d_v < Δ, we assume s_i = ⊥, ∀i > d_v
6:         if Map(a, b) ∉ {Map(s_i.a, s_i.b) | i ∈ {1, ..., Δ} ∧ s_i ≠ ⊥} then return Map(a, b)
7:         else
8:             if (a = Δ) ∨ (b = Δ) then          ▷ We compute the flipped edges.
9:                 f ← f ∪ {s_i.z | (i ∈ {1, ..., Δ})(s_i ≠ ⊥) ∧ ((s_i.a = Δ) ∨ (s_i.b = Δ))}
10:            end if
11:            a ← ℕ ∖ {s_i.a | i ∈ Larger(s, (s_1, ..., s_Δ))}
12:            b ← ℕ ∖ {s_i.b | i ∈ Smaller(s, (s_1, ..., s_Δ))}
13:            if |Smaller(s, (s_1, ..., s_Δ))| ≥ 1 then α ← true
14:            end if
15:            if |Larger(s, (s_1, ..., s_Δ))| ≥ 1 then β ← true
16:            end if
17:            if SpecialTermination(s, (s_1, ..., s_Δ)) then return (0, Δ)
18:            end if
19:        end if
20:    end repeat
21: end procedure
```

7 Open Questions

We have shown that every n-node graph of maximum degree Δ can be properly colored with $\frac{1}{2}(\Delta+1)(\Delta+2)-1$ colors in ASYNC LOCAL, in $O(\log^\star n)+f(\Delta)$ rounds. The number of colors may seem large, but the ASYNC LOCAL model is considerably weaker than the (synchronous and failure-free) LOCAL model. In particular, it is known that even the clique with $n=\Delta+1$ nodes cannot be colored with less than $2\Delta+1$ colors in ASYNC LOCAL (whenever $\Delta+1$ is power of a prime), and we have shown that there exists an infinite family of regular graphs with even degree Δ that cannot be colored with less than $\Delta+3$ colors in ASYNC LOCAL. One major question as far as solving graph problems in asynchronous crash-prone networks is thus the following.

Open Problem: Is there a $(2\Delta+1)$-coloring algorithm for graphs with maximum degree Δ in the ASYNC LOCAL model, for every $\Delta \geq 2$?

Of course, if one puts aside cliques, there might be a coloring algorithm for ASYNC LOCAL using a palette of less than $2\Delta+1$ colors. However, we have shown that, for $\Delta=2$, the bound $2\Delta+1=5$ is tight for infinitely many cycles. The only generic bound applying to infinitely many graphs of maximum degree Δ is however only $\Delta+3$, so there might be room for improvement. Yet, saving even just a single color in a palette of $\frac{1}{2}(\Delta+1)(\Delta+2)$ colors was very delicate and difficult. So, progressing from a quadratic number of colors to a linear number of colors appears to be a challenge in ASYNC LOCAL.

Finally, we question the efficiency of randomized algorithms in the ASYNC LOCAL model.

Open Problem: To which extent randomized algorithms help in the ASYNC LOCAL model, in term of both complexity and computability?

References

1 Hagit Attiya and Ami Paz. Counting-based impossibility proofs for set agreement and renaming. *J. Parallel Distributed Comput.*, 87:1–12, 2016. doi:10.1016/J.JPDC.2015.09.002.

2 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. Wiley, 2004.

3 Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Michael E. Saks. Adapting to asynchronous dynamic networks. In *24th ACM Symposium on Theory of Computing (STOC)*, pages 557–570, 1992.

4 Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 514–522, 1990. doi:10.1109/FSCS.1990.89572.

5 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\delta + 1)$-coloring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 437–446, 2018. URL: https://dl.acm.org/citation.cfm?id=3212769.

6 Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8, 2009. doi:10.1109/IPDPS.2009.5161053.

7 Jean R. S. Blair and Fredrik Manne. An efficient self-stabilizing distance-2 coloring algorithm. *Theoretical Computer Science*, 444:28–39, 2012. doi:10.1016/J.TCS.2012.01.034.

8 Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Brief announcement: Memory lower bounds for self-stabilization. In *33rd International Symposium on Distributed Computing (DISC)*, volume 146 of *LIPIcs*, pages 37:1–37:3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.DISC.2019.37.

9 Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election - the exponential advantage of being talkative. In *27th Int. Symp. on Distributed Computing (DISC)*, volume 8205 of *LNCS*, pages 76–90. Springer, 2013. doi:10.1007/978-3-642-41527-2_6.

10 Armando Castañeda, Pierre Fraigniaud, Ami Paz, Sergio Rajsbaum, Matthieu Roy, and Corentin Travers. Synchronous t-resilient consensus in arbitrary graphs. *Inf. Comput.*, 292:105035, 2023. doi:10.1016/J.IC.2023.105035.

11 Armando Castañeda, Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Making local algorithms wait-free: the case of ring coloring. *Theory of Computing Systems*, 63(2):344–365, 2019. doi:10.1007/S00224-017-9772-Y.

12 Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *18th ACM Symposium on Theory of Computing (STOC)*, pages 206–219, 1986. doi:10.1145/12130.12151.

13 Carole Delporte-Gallet, Hugues Fauconnier, Pierre Fraigniaud, and Mikaël Rabie. Distributed computing in the asynchronous LOCAL model. In *21st International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS 11914, pages 105–110. Springer, 2019.

14 P. Erdös, P. Frankl, and Z. Füredi. Families of finite sets in which no set is covered by the union ofr others. *Israel Journal of Mathematics*, 51(1):79–89, 1985.

15 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

16 Pierre Fraigniaud, Patrick Lambein-Monette, and Mikaël Rabie. Fault tolerant coloring of the asynchronous cycle. In *36th Int. Symp. on Distributed Computing (DISC)*, volume 246 of *LIPIcs*, pages 23:1–23:22. Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.DISC.2022.23.

17 Marc Fuchs and Fabian Kuhn. List defective colorings: Distributed algorithms and applications. In *37th Int. Symp. on Distributed Computing (DISC)*, volume 281 of *LIPIcs*, pages 22:1–22:23. Schloss Dagstuhl - Leibniz-Zentrum für Inf., 2023. doi:10.4230/LIPICS.DISC.2023.22.

18 Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *62nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1009–1020, 2021. doi:10.1109/FOCS52979.2021.00101.

19 Mohsen Ghaffari and Anton Trygub. A near-optimal deterministic distributed synchronizer. In *42th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 180–189, 2023. doi:10.1145/3583668.3594598.

20 Magnús M. Halldórsson, Fabian Kuhn, Alexandre Nolin, and Tigran Tonoyan. Near-optimal distributed degree+1 coloring. In *54th ACM Symposium on Theory of Computing (STOC)*, pages 450–463, 2022. doi:10.1145/3519935.3520023.

21 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013.

22 Juho Hirvonen and Jukka Suomela. *Distributed Algorithms*. Creative Commons, 2020.

23 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

24 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

25 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

26 Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. Topological characterization of consensus under general message adversaries. In *38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 218–227, 2019. doi:10.1145/3293611.3331624.

27 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

28 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018. doi:10.1007/978-3-319-94141-7.

29 Nicola Santoro and Peter Widmayer. Time is not a healer. In *6th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 349 of *LNCS*, pages 304–313. Springer, 1989. doi:10.1007/BFB0028994.

30 Kyrill Winkler, Ami Paz, Hugo Rincon Galeana, Stefan Schmid, and Ulrich Schmid. The time complexity of consensus under oblivious message adversaries. In *14th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 251 of *LIPIcs*, pages 100:1–100:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ITCS.2023.100.

A Example of an execution of an algorithm for 6-coloring cycles

An example of an execution of Algorithm 1 is provided in Table 1, in which the old and new states of each node after each step is displayed.

Table 1 An example of execution of Algorithm 1, for the cycle C_5 with consecutive node identifiers $(3, 5, 4, 1, 6)$. The example corresponds to the scheduling $\mathcal{S} = \{1, 3, 5\}, \{4, 5\}, \{3, 4\}, \{6\}, \{6\}, \ldots$, and T stands for Terminated. At each step, the states that are updated are highlighted in bold.

	3		5		4		1		6	
	Old	New	Old	New	Old	New	Old	New	Old	New
Initialization	$\perp$	$(3,0,0)$	$\perp$	$(5,0,0)$	$\perp$	$(4,0,0)$	$\perp$	$(1,0,0)$	$\perp$	$(6,0,0)$
$\{1,3,5\}$										
after write	$\mathbf{(3,0,0)}$	$(3,0,0)$	$\mathbf{(5,0,0)}$	$(5,0,0)$	$\perp$	$(4,0,0)$	$\mathbf{(1,0,0)}$	$(1,0,0)$	$\perp$	$(6,0,0)$
update	$(3,0,0)$	$\mathbf{(3,1,0)}$	$(5,0,0)$	$\mathbf{(5,0,1)}$	$\perp$	$(4,0,0)$	$(1,0,0)$	$\boldsymbol{T(0,0)}$	$\perp$	$(6,0,0)$
$\{4,5\}$										
after write	$(3,0,0)$	$(3,1,0)$	$\mathbf{(5,0,1)}$	$(5,0,1)$	$\mathbf{(4,0,0)}$	$(4,0,0)$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
update	$(3,0,0)$	$(3,1,0)$	$(5,0,1)$	$\boldsymbol{T(0,1)}$	$(4,0,0)$	$\mathbf{(4,1,1)}$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
$\{3,4\}$										
after write	$\mathbf{(3,1,0)}$	$(3,1,0)$	$(5,0,1)$	$T(0,1)$	$\mathbf{(4,1,1)}$	$(4,1,1)$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
update	$(3,1,0)$	$\boldsymbol{T(1,0)}$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$\boldsymbol{T(1,1)}$	$(1,0,0)$	$T(0,0)$	$\perp$	$(6,0,0)$
$\{6\}$										
after write	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$\mathbf{(6,0,0)}$	$(6,0,0)$
update	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$(6,0,0)$	$\mathbf{(6,0,1)}$
$\{6\}$										
after write	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$\mathbf{(6,0,1)}$	$(6,0,1)$
update	$(3,1,0)$	$T(1,0)$	$(5,0,1)$	$T(0,1)$	$(4,1,1)$	$T(1,1)$	$(1,0,0)$	$T(0,0)$	$(6,0,1)$	$\boldsymbol{T(0,1)}$

B A Counterexample for an Existing Algorithm for 5-Coloring Cycles

We merely exhibit an instance of 5-coloring C_4 for which the algorithm in [16] does not terminate[4]. The algorithm presented in [16] is shown in Algorithm 5. In Table 2, we provide an example of execution where the algorithm loops forever.

[4] For the interested reader, we found this counterexample by implementing a simulator for the ASYNC LOCAL model. This simulator tests a given algorithm with random schedulings.

Algorithm 5 The (erroneous) 5-coloring algorithm of [16].

```
procedure FiveColoring(id_v, input_v)
    x ← id_v;  a ← 0;  b ← 0                          ▷ s = (x, a, b) is the state of node v
    repeat forever
        (s_1, s_2) ← WriteSnapshot(s)
        P^+ ← {i ∈ {1,2} | s_i ≠ ⊥ ∧ s_i.x > x}          ▷ neighbors with larger id
        C^+ ← {x_i.a | i ∈ P^+} ∪ {x_i.b | i ∈ P^+}      ▷ a and b of neighbors with larger id
        C ← {x_i.a | i ∈ {1,2} ∧ s_i ≠ ⊥} ∪ {x_i.b | i ∈ {1,2} ∧ s_i ≠ ⊥}   ▷ a and b of all neighbors
        if a ∉ C then return a
        else
            if b ∉ C then return b
            else
                a ← min ℕ ∖ C^+
                b ← min ℕ ∖ C
            end if
        end if
    end repeat
end procedure
```

Table 2 An example of execution where Algorithm 5 loops, for a 4-cycle with nodes' identifiers $(3, 4, 2, 1)$ in consecutive order. The example is for the scheduling $\{2, 3, 4\}, \{1, 3, 4\}, \{3, 4\}, \{3, 4\}, \ldots$. Observe that the state obtained after scheduling $\{1, 3, 4\}$ is the same state as the one obtained after the fourth step (when $\{3, 4\}$ is scheduled for the second time). Therefore, there exists a scheduling that makes the algorithm looping forever.

	3		4		2		1	
	Old	New	Old	New	Old	New	Old	New
Initialization	$\bot$	$(3,0,0)$	$\bot$	$(4,0,0)$	$\bot$	$(2,0,0)$	$\bot$	$(1,0,0)$
$\{2,3,4\}$								
after write	**(3, 0, 0)**	$(3,0,0)$	**(4, 0, 0)**	$(4,0,0)$	**(2, 0, 0)**	$(2,0,0)$	$\bot$	$(1,0,0)$
update	$(3,0,0)$	**(3, 1, 1)**	$(4,0,0)$	**(4, 0, 1)**	$(2,0,0)$	**(2, 1, 1)**	$\bot$	$(1,0,0)$
$\{1,3,4\}$								
after write	**(3, 1, 1)**	$(3,1,1)$	**(4, 0, 1)**	$(4,0,1)$	$(2,0,0)$	$(2,1,1)$	**(1, 0, 0)**	$(1,0,0)$
update	$(3,1,1)$	**(3, 2, 2)**	$(4,0,1)$	**(4, 0, 2)**	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	**(1, 2, 2)**
$\{3,4\}$								
after write	**(3, 2, 2)**	$(3,2,2)$	**(4, 0, 2)**	$(4,0,2)$	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$
update	$(3,2,2)$	**(3, 1, 1)**	$(4,0,2)$	**(4, 0, 1)**	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$
$\{3,4\}$								
after write	**(3, 1, 1)**	$(3,1,1)$	**(4, 0, 1)**	$(4,0,1)$	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$
update	$(3,1,1)$	**(3, 2, 2)**	$(4,0,1)$	**(4, 0, 2)**	$(2,0,0)$	$(2,1,1)$	$(1,0,0)$	$(1,2,2)$