Making Self-Stabilizing Algorithms for Any Locally Greedy Problem

Johanne Cohen
Université Paris-Saclay, CNRS, LISN, 91405, Orsay, France

Laurence Pilard
LI-PaRAD, UVSQ, Université Paris-Saclay, France

Mikaël Rabie
IRIF-CNRS, Université Paris Cité, France

Jonas Sénizergues
Université Paris-Saclay, CNRS, LISN, 91405, Orsay, France

Abstract

Self-stabilizing algorithms are a way to deal with network dynamicity, as it will update itself after a network change (addition or removal of nodes or edges), as long as changes are not frequent. We propose an automatic transformation of synchronous distributed algorithms that solve locally greedy and mendable problems into self-stabilizing algorithms in anonymous networks.

Mendable problems are a generalization of greedy problems where any partial solution may be transformed -instead of completed- into a global solution: every time we extend the partial solution, we are allowed to change the previous partial solution up to a given distance. Locally here means that to extend a solution for a node, we need to look at a constant distance from it.

In order to do this, we propose the first explicit self-stabilizing algorithm computing a $(k, k-1)$-ruling set (*i.e.* a "maximal independent set at distance k"). By combining this technique multiple times, we compute a distance-K coloring of the graph. With this coloring we can finally simulate Local model algorithms running in a constant number of rounds, using the colors as unique identifiers.

Our algorithms work under the Gouda daemon, similar to the probabilistic daemon: if an event should eventually happen, it will occur.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Greedy Problem, Ruling Set, Distance-K Coloring, Self-Stabilizing Algorithm

Digital Object Identifier 10.4230/LIPIcs.SAND.2023.11

Related Version *Full Version*: https://arxiv.org/abs/2208.14700

1 Introduction

The greedy approach is often considered to solve a problem: Is it possible to build up a solution step by step by completing a partial solution? For example, in graph theory, one can consider the Maximal Independent Set (MIS) problem that consists in selecting a set of nodes such that no two chosen nodes are adjacent and any unselected node is a neighbor of a selected one. To produce a MIS, a simple algorithm selects a node, rejects all its neighbors, and then repeats this operation until no node is left. Another classical greedy algorithm is the one that produces a $(\Delta+1)$-coloring of a graph, where Δ is the maximum degree in the graph. Each time a node is considered, as it has at most Δ different colors in its neighborhood, one can always choose a different color to extend the current partial solution. Observe that most graphs admit a Δ-coloring, which cannot be found with this heuristic. We can also notice that the size of a MIS can be arbitrarily smaller than the size of a *maximum* independent set. More generally, greedy algorithm are simple algorithm that build not necessarily optimal

© Johanne Cohen, Laurence Pilard, Mikaël Rabie, and Jonas Sénizergues;
licensed under Creative Commons License CC-BY 4.0

solutions. *Greedy problems* are problems that can be solved using a greedy algorithm. We say that a problem is *Locally greedy* if you only need to have information at some constant distance from a node to complete a partial solution on that node. For example Δ-coloring and MIS problems are *Locally greedy* problems while the spanning tree problem is not.

Sometimes, it is not possible to complete a partial solution, and it might be necessary to change some of the outputs to reach a feasible solution. The idea of fixing, or mending a solution, in distributed computing, has been studied a lot, for example in [29, 30]. A formal definition of *Mendable* problems has recently been introduced in [8]. In a graph, we compute the output of each node one after another. For each chosen node, it is possible to change the output of its neighborhood, but only up to some distance. The set of mendable problems is larger than the set of greedy ones. For instance, the 4-coloring of the grid is a mendable problem, but it cannot be solved greedily, as its maximal degree Δ is equal to 4.

A more generalized way to consider MIS are *ruling sets*. Given a graph $G = (V, E)$, a *(a, b)-ruling set* is a subset $S \subset V$ such that the distance between any two nodes in S is at least a, and any node in V is at distance at most b from some node in S.

In particular, a $(2, 1)$-ruling set is a MIS of G. A $(k, k-1)$-ruling set S is a maximal independent set at distance k (also called maximal distance-k independent set): all the elements of S are at distance at least k from each other, every other node is at distance at most $k-1$ from S, and thus cannot be added. Note that it is a MIS of G^{k-1} (the graph with the same vertices as G, and with edges between two vertices if there are at distance $k-1$ or less from each other in G), and this problem can be greedily solved.

A *distance-K coloring* of a graph $G = (V, E)$ is a mapping $\mathcal{C} : V \to \mathbb{N}$ such that for any pair of nodes u v at distance at most K from each other, we have $\mathcal{C}(u) \neq \mathcal{C}(v)$. A way to produce a distance-K coloring is to partition V into sets of nodes at distance at least $k > K$ from each other, *i.e.* distance-k independent sets, each one representing a color. One can construct such a partition sequentially by constructing a partition into $X \geq \Delta^k$ distance-k independent sets $\{S^{(i)}\}_{i \leq X}$, where $S^{(i)}$ is a distance-k independent set of G maximal under the constraint that every node of the independent set must be in $V \setminus \bigcup_{j<i} S^{(j)}$. These distance-$k$ independent sets can be computed similarly to $(k, k-1)$-ruling sets.

The LOCAL model [36] is a synchronous model with unlimited bound on memory where each node starts with a unique identifier. In particular, after k communication rounds, each node knows everything about its neighborhood at distance k. A distance-$2K$ coloring allows to simulate LOCAL algorithms running in at most K rounds, as no node can see twice the same identifier in its neighborhood at distance K (see for example [8, 12]).

An algorithm is *Self-stabilizing* if, from any configuration (system state), the system will eventually reach a configuration with a good output/solution (see [3, 17]). In particular, such an algorithm permits one to get back to a good configuration if some faults occur (for example, a node accidentally switches its state). In such situations, being able to locally mend around the fault is key, as it minimises the information and time needed to fix the issue. Self-stabilizing algorithms using mending techniques have been extensively studied, for example in [1, 22, 38]. Self-stabilization can manage the updates in a dynamic network when they occur not too often as adding or removing nodes or links can be viewed as transient faults.

1.1 Our Contribution

This paper aims to adapt the idea of LOCAL mending from [8] to produce a self-stabilizing algorithm working in anonymous networks for any constant-radius mendable problem. It uses $f(\Delta) = \Delta^{\Delta^{O(1)}}$ states (in particular, it becomes constant for bounded degree graphs).

In Section 2, we provide a self-stabilizing algorithm that computes a $(k, k-1)$-ruling set in an anonymous network under the Gouda daemon. This algorithm will be used as a sub-routine for our construction. The algorithm detects when a leader can be added (*i.e.* there is a ball of radius k without leader) or two leaders are too close (*i.e.* at distance less than k from each other). To that end, each node computes its distance from the leaders. If a node and its neighbors are at distance at least $k-1$ from the leaders, that node can try to add itself to the ruling set. If two leaders are too close, thanks to a clock system consisting of a mosaic of local synchronizers beta of Awerbuch [4], a node in the middle of the path will eventually detect the problem and initiate the removal of the leaders from the set. Thanks to the Gouda daemon, we ensure that only a few nodes will try to add themselves simultaneously and that the clock system will eventually detect collisions. Section 3 contains the proof that a stable configuration can always be reached, and the Gouda daemon ensures that it ultimately happens.

In Section 4, by combining this algorithm Δ^k times, we partition the graph into distance-k independent sets, which corresponds to a distance-K coloring for any $K < k$. This coloring allows us to consider nodes of each set sequentially to compute a solution to some greedy problem. In Section 5, we present a solution allowing us to solve any T-mendable problem in anonymous networks, where T is a constant corresponding to the radius up to which we are permitted to change the output of a node. To that end, we use the fact that a Local algorithm runs in r rounds for some constant r, when a distance-$2T+1$ coloring is given. To do that, we compute a distance-$2T+1$ and a distance-$2r+1$ coloring. That way, each node can access their neighborhood at the proper distance and compute the output the Local algorithm would have given in that situation.

1.2 Related Work

The notion of *checking locally* was introduced by Afek et al. [2] and its relationship with the idea of *solving locally* by Naor and Stockmeyer [31]. This work, along with Cole and Vishkin's algorithm that efficiently computes a 3-coloring of a ring [14], leads to the notion of *Locally Checkable Labelling problems* (Lcl) and the Local model. Locally checkable problems are problems such that when the output is locally correct for each node, the global output is guaranteed to be correct too. Coloring and MIS belong to that field. Ruling Sets are also Lcl problems: to check locally that the solution is correct, the distance to the set must be given in the output. The Local model (see [36] for a survey) is a synchronous model that requires unique identifiers but does not impose any restriction on communication bandwidth or computation complexity. The goal is to find sublinear time algorithms. An adaptation of the Local model, the Slocal model [21] considers algorithms executed on nodes one after another, only one time each, but are allowed to see the state of every node up to some distance when they do. In particular, this model solves locally greedy problems with a constant distance of sight.

Bitton et al. [11] designed a self-stabilizing transformer for Local problems. Their probabilistic transformer converts a given fault-free synchronous algorithm for Lcl problems into a self-stabilizing synchronous algorithm for the same problem in anonymous networks. The overheads of this transformation in terms of message complexity and average time complexity are upper bounded: the produced algorithms stabilize in time proportional to $\log(\alpha + \Delta)$ in expectation, where α is the number of faulty nodes. Afek and Dolev [1] designed a self-stabilizing transformer. It converts any distributed algorithm that works in a network with identifiers and diameter less than D under the synchronous daemon into a self-stabilizing one adding additional costs in time (additional $O(D)$), memory ($O(nD)$ multiplier), and communication ($O(nD)$ multiplier).

Awerbuch et al. [5] introduced the ruling set as a tool for decomposing the graph into small-diameter connected components. As for the seminal work, the ruling set problems have been used as a sub-routine function to solve some other distributed problems (network decompositions [5, 10], colorings [32], shortest paths [27]).

The MIS problem has been extensively studied in the LOCAL model, [19, 34, 13] for instance and in the CONGEST model [33] (synchronous model where messages are $O(\log n)$ bits long). In the LOCAL model, Barenboim et al. [9] focused on systems with unique identifiers and gave a self-stabilizing algorithm producing an MIS within $O(\Delta+\log^* n)$ rounds. Balliu et al. [6] prove that the previous algorithm [9] is optimal for a wide range of parameters in the LOCAL model. In the CONGEST model, Ghaffari et al. [20] prove that there exists a randomized distributed algorithm that computes a maximal independent set in $O(\log \Delta \cdot \log\log n + \log^6 \log n)$ rounds with high probability. Considering the problem (α, β)-ruling set in a more general way, Balliu et al. [7] give some lower bound for computing a $(2, \beta)$- ruling set in the LOCAL model: any deterministic algorithm requires $\Omega\left(\min\left\{\frac{\log \Delta}{\beta \log\log \Delta}, \log n\right\}\right)$ rounds.

Up to our knowledge, no self-stabilizing algorithm has been designed for only computing $(k, k-1)$-ruling sets where $k > 2$ under the Gouda daemon. Self-stabilizing algorithms for maximal independent set have been designed in various models (anonymous network [35, 40, 39] or not [23, 28, 37]). Shukla et al. [35] present the first self-stabilization algorithm for finding a MIS for anonymous networks. Turau [37] gives the best-known result with $O(n)$ moves under the distributed daemon. Recently, some works improved the results in the synchronous model. For non-anonymous networks, Hedetniemi [26] designed a self-stabilization algorithm that stabilizes in $O(n)$ synchronous rounds. Moreover, for anonymous networks, Turau [39] designs some randomized self-stabilizing algorithms for maximal independent set that stabilizes in $O(\log n)$ rounds w.h.p. See the survey [25] for more details on MIS self-stabilizing algorithms.

Our algorithm uses a clock system close to information propagation with feedback (or PIF) mechanism, however more than these classical solutions are needed. Indeed, while we assume multiple leaders, in classical PIF algorithms, only one leader is usually assumed under identified system [15] or anonymous one [16]. Their mechanism relies on waves of information from a source to the network, layer by layer.

1.3 Model

A distributed system consists of a set of processes where two adjacent processes can communicate. The communication relation is represented by a graph $G = (V, E)$ where V is the set of the processes (we call *node* any element of V from now on) and E represents the neighborhood relation between them, *i.e.*, $uv \in E$ when u and v are adjacent nodes. The set of *neighbors* of a node u is denoted by $N(u)$. We assume the system to be *anonymous*, meaning that a node has no identifier. Moreover, we consider undirected networks (i.e. $uv \in E \iff vu \in E$). We denote by Δ the maximum degree in the graph.

We denote by $dist(u, v)$ the distance between the two nodes u and v in the graph. When S is a subset of V, $dist(u, S)$ is the smallest distance from u to an element in S. In what follows, the concept of *ball* will play an important role. Formally, the *ball* of radius i and center s, $\mathcal{B}(s, i)$, is the set of nodes that are at distance at most i from s. Observe that a ball of radius $a-1$ centered in a node of the ruling set S contains only one node in S.

For communication, we consider the *shared memory model*: the *local state* of each node corresponds to a set of *local variables*. A node can read its local variables and its neighbors' but can only rewrite its local variables. A *configuration* is the value of the local states of all nodes in the system. When u is a node and x is a local variable, the *x-value* of u is the

value x_u. Each node executes the same algorithm that consists of a set of *rules*. Each rule is of the form *"**if** ⟨guard⟩ **then** ⟨command⟩"* and is parameterized by the node where it would be applied. Each rule also has a priority number. The *guard* is a predicate over the variables of the current node and its neighbors. The *command* is a sequence of actions that may change the values of the node's variables (but not those of its neighbors). A rule is *activable* in a configuration C if its guard in C is true. A process is *eligible* for the rule $\mathcal{R}$ in a configuration C if its rule $\mathcal{R}$ is activable and no rule of lower priority number is activable for that node in C. We say in that case that the process is *activable* in C. An *execution* is an alternate sequence of configurations and actions $\sigma = C_0, A_0, \ldots, C_i, A_i, \ldots$, such that $\forall i \in \mathbb{N}^*$, configuration C_{i+1} is obtained by executing the command of at least one rule that is activable in configuration C_i. More precisely, the set of actions A_i is the non-empty set of activable processes in C_i such that their activable rules have been executed to reach C_{i+1}.

The goal of a self-stabilizing algorithm is to be robust to perturbations. An initial configuration cannot follow any restriction, and failures can occur, changing the state of some of the nodes. A self-stabilizing algorithm must be able to recover and reach a correct general output from any configuration.

In a distributed system, multiple nodes can be active simultaneously, meaning they are in a state where they can make a computation. The definition of a self-stabilizing algorithm is centred around the notion of *daemon*. A *daemon* captures which set of activable rules some scheduler choose during the execution. See [18] for a taxonomy. Our algorithm cannot work on a fully synchronous deterministic anonymous network, as it relies on using asynchronous clocks from different leaders. To that end, we use the *Gouda* daemon to break symmetries, as it ensures asynchronous activation of the nodes. We aim to create algorithms for any mendable problems to solve the computability question. Hence, we do not focus on the complexity time, which could be captured by a probabilistic daemon. The Gouda daemon captures the same computable problems as the probabilistic daemon. If something happens with probability 1 with the probabilistic daemon (where each rule has a probability < 1 to be activated), it eventually happens with the Gouda daemon.

▶ **Definition 1** ([18, 24]). *We say that an execution $\sigma = C_0 \to C_1 \to C_2 \ldots$ is under the Gouda daemon if: for any configurations C and C' such that $C \to C'$ can be executed, if C appears infinitely often in σ, then C' also appears infinitely often in σ.*

An algorithm is *self-stabilizing* for a given specification (i.e. a set of restrictions over the configurations) under some daemon if there exists a subset $\mathcal{L}$ of the set of all configurations, called the *legitimate configurations*, such that: (i) any configuration in $\mathcal{L}$ verifies the specification, and any execution under the said daemon starting in $\mathcal{L}$ stays in $\mathcal{L}$ (*correctness*). and (ii) any execution under the said daemon eventually reaches a configuration in $\mathcal{L}$ (*convergence*). The set $\mathcal{L}$ is called the set of *legitimate configurations*.

2 Self-Stabilizing Algorithm for Computing a $(k, k-1)$-Ruling Set

2.1 General Overview

As we want to compute a $(k, k-1)$-ruling set, a node needs to detect when it is currently "too far" from the nodes pretending to be in the ruling set. When $k = 2$, a $(2, 1)$-ruling set is an MIS, and some self-stabilization algorithms are designed for finding an MIS [35, 40, 39]. For the remaining of the document, we assume $k > 2$.

To this aim, the local variable d represents the distance at which the node thinks it is from the ruling set. In particular, a d-value of 0 indicates that a node is (or thinks it is) in the ruling set, and we denote by $S(C)$ the set of those nodes in a given configuration C.

Algorithm 1 Algorithm for the $(k, k-1)$-Ruling Set.

Attributes of the nodes

$\mathbf{d_u} \in [\![0, k-1]\!]$
$\mathbf{err_u} \in \{0, 1\}$
For every $i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!]$: $\mathbf{c_{i,u}} \in \mathbb{Z}/4\mathbb{Z}$ and $\mathbf{b_{i,u}} \in \{\uparrow, \downarrow\}$

Predicates

well_defined$(u) \equiv err_u = 0 \wedge \forall v \in N(u), |d_u - d_v| \leq 1 \wedge (d_u > 0 \Rightarrow (\exists v \in N(u), d_v = d_u - 1))$
leader_down$(u) \equiv d_u = 0 \Rightarrow \forall i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!], b_{i,u} =\downarrow$
branch_coherence_up$(u, i) \equiv$
$\forall v \in N(u), d_v = d_u - 1 \Rightarrow (b_{i,u}, b_{i,v}, c_{i,v}) \in \{(\uparrow, \uparrow, c_{i,u}), (\uparrow, \downarrow, c_{i,u}), (\uparrow, \downarrow, c_{i,u} + 1), (\downarrow, \downarrow, c_{i,u})\}$
branch_coherence_down$(u, i) \equiv$
$\forall v \in N(u), d_v = d_u + 1 \Rightarrow (b_{i,u}, b_{i,v}, c_{i,v}) \in \{(\uparrow, \uparrow, c_{i,u}), (\downarrow, \uparrow, c_{i,u}), (\downarrow, \uparrow, c_{i,u} - 1), (\downarrow, \downarrow, c_{i,u})\}$
branch_coherence$(u) \equiv d_u \geq \lfloor \frac{k}{2} \rfloor \vee \big(branch_coherence_up(u, d_u) \wedge$
$\forall i \in [\![d_u + 1, \lfloor \frac{k}{2} \rfloor - 1]\!], branch_coherence_up(u, i) \wedge branch_coherence_down(u, i)\big)$

Rules

Incr Leader:: (priority 2)
if well_defined$(u) \wedge (d_u = 0) \wedge (\exists i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!], \forall v \in N(u), d_v = 1 \wedge c_{i,u} - c_{i,v} = 0)$
then For all such i, $c_{i,u} := c_{i,u} + 1$

Sync 1 down:: (priority 2)
if well_defined$(u) \wedge \exists! v \in N(u), \exists i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!], d_u = 1 \wedge d_v = 0 \wedge c_{i,u} = c_{i,v} - 1 \wedge b_{i,u} =\uparrow)$
then For all such i, $c_{i,u} := c_{i,v}$; $b_{i,u} :=\downarrow$

Sync 2+ down:: (priority 2)
if well_defined$(u) \wedge 1 < d_u < \lfloor \frac{k}{2} \rfloor$
$\wedge(\exists i \in [\![d_u, \lfloor \frac{k}{2} \rfloor - 1]\!], b_{i,u} =\uparrow \wedge \forall v \in N(u), d_v = d_u - 1 \Rightarrow (c_{i,u} = c_{i,v} - 1 \wedge b_{i,v} =\downarrow))$
then For all such i, $c_{i,u} := c_{i,v}$; $b_{i,u} :=\downarrow$

Sync 1+ up:: (priority 2)
if well_defined$(u) \wedge 0 < d_u < \lfloor \frac{k}{2} \rfloor$
$\wedge(\exists i \in [\![d_u + 1, \lfloor \frac{k}{2} \rfloor - 1]\!], b_{i,u} =\downarrow \wedge \forall v \in N(u), d_v = d_u + 1 \Rightarrow (c_{i,u} = c_{i,v} \wedge b_{i,v} =\uparrow))$
then For all such i, $b_{i,u} :=\uparrow$

Sync end-of-chain:: (priority 2)
if well_defined$(u) \wedge 0 < d_u < \lfloor \frac{k}{2} \rfloor \wedge \forall v \in N(u), d_v = d_u - 1 \Rightarrow (c_{d_u,u} = c_{d_u,v} - 1 \wedge b_{i,v} =\downarrow))$
then $b_{d_u,u} :=\uparrow$; $c_{d_u,u} := c_{i,v}$

Update distance :: (priority 0)
if $(d_u \neq 0) \wedge d_u \neq \min(\min\{d_v | v \in N(u)\} + 1, k-1)$
then $d_u := \min(\min\{d_v | v \in N(u)\} + 1, k-1)$
If $d_u < \lfloor \frac{k}{2} \rfloor$: Let $v :=$ choose$(\{w \in N(u) | d_w = d_u - 1\})$
For each $i \in [\![d_u, \lfloor \frac{k}{2} \rfloor - 1]\!], c_{i,u} := c_{i,v}$; $b_{i,u} := b_{i,v}$

Become Leader :: (priority 2)
if $err_u = 0 \wedge (d_u = k-1) \wedge \forall v \in N(u), d_v = k-1$
then $d_u := 0$, For each $i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!], c_{i,u} := 0, b_{i,u} :=\downarrow$

Leader down :: (priority 1)
if $well_defined(u) \wedge d_u = 0 \wedge \exists i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!], b_{i,u} =\uparrow$ **then** For each $i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!]$, $b_{i,u} :=\downarrow$

Two Heads:: (priority 1)
if $err_u = 0 \wedge \exists v, v' \in (N(u) \cup \{u\})^2, v \neq v' \wedge d_v = d_{v'} = 0)$ **then** $err_u := 1$

Branch incoherence:: (priority 1)
if $err_u = 0 \wedge \neg branch_coherence(u)$ **then** $err_u := 1$

Error Spread :: (priority 2)
if $err_u = 0 \wedge (d_u \leq \lfloor \frac{k}{2} \rfloor - 1) \wedge (\exists v \in N(u), err_v = 1 \wedge d_u < d_v)$ **then** $err_u := 1$

Reset Error :: (priority 2)
if $(err_u = 1) \wedge ([d_u > \lfloor \frac{k}{2} \rfloor] \vee [\forall v \in N(u), d_v \geq d_u \vee err_v = 1])$
then $err_u := 0$, If $d_u = 0$, $d_u := 1$, For each i, $c_{i,u} := 0$, $b_{i,u} :=\uparrow$

Any other value of d_u represents the distance to $S(C)$ (the minimum between $k-1$ and the said distance). The rule **Update distance** has the highest priority. Its goal is to ensure that each node eventually gets its distance to $S(C)$ accurately. When a node u has its local variable d_u equal to $k-1$ and is surrounded by nodes of d-value $k-1$, it "knows" that it is far enough from $S(C)$ to be added to it. Node u can then execute rule **Become Leader** to do so. Update of d-values will then spread from the new member of $S(C)$ through the execution of rule **Update distance**.

The way to insert new nodes into $S(C)$ cannot avoid the fact that two new members of $S(C)$ may be too close. A way to detect those problems is needed to guarantee that we will not let those nodes in $S(C)$.

If they are close enough (distance 2 or less), it can be directly detected by a node (either a common neighbor if they are at distance 2 or one of them if they are at distance 1). The rule **Two Heads** is here to detect this.

No node can detect this problem when problematic nodes are too far away. To remedy this, each node maintains a synchronized clock system around each node of $S(C)$ by executing the *stationary rules*. For this reason, we split the set of rules into two groups:

- The *stationary* rules are the rules **Incr Leader**, **Sync 1 down**, **Sync 2+ down**, **Sync 1+ up**, and **Sync end-of-chain**;
- The *convergence* rules are the rules **Remote Collision**, **Two Heads**, **Branch Incoherence**, **Update Distance**, **Become Leader**, **Error Spread**, **Reset Error**, and **Leader down**.

We say that a node in $S(C)$ is the *leader* of the nodes under its influence, corresponding to the nodes in its ball at distance $\lfloor \frac{k}{2} \rfloor$. Assuming d-value has already been spread, the clock of index i of nodes that gave the same leader will always be either equal or out-of-sync by 1. Thus, a node detects that two nodes in $S(C)$ are too close when it sees in its neighbourhood two nodes with clocks out-of-sync by 2. It will raise an error when activated by executing rule **Remote Collision**. The error is then propagated toward the problematic members of $S(C)$ by rule **Error Spread**.

In both previous cases, the problematic nodes of $S(C)$ end up having err-value 1, which makes them leave $S(C)$ by executing rule **Reset Error**. Afterwards, rule **Update distance** will, over time, update the d-values of the nodes at distance up to k to that node.

The goal of our algorithm is to ensure that we reach locally a configuration from which, when a node is inserted in $S(C)$, and no node gets added at distance at most $k-1$ away, it remains in $S(C)$ forever. Note that when it is executed, rule **Update distance** setup the clock values and arrows (variables c and b) so that the newly updated node is synchronized to its "parent" (the node it takes as a reference to update its d-value).

The target configuration is **not** a *stable* configuration, and from it, all the nodes can only execute *stationary rules*. In this configuration, $S(C)$ is guaranteed to be a $(k, k-1)$-ruling set of the underlying graph. Note that the predicate $well_defined$ appears in the guard of every stationary rule. The predicate guarantees that the considered node neither is in error-detection mode nor has some incorrect d-values in its neighbourhood before executing any clock-related rule.

2.2 The Clock System

Now, we describe the clock system that detects two leader nodes in $S(C)$ at a distance less than k. The leaders are the nodes that update the clock value c_i and propagate it to its "children" and so on. For a given clock index i, when every neighbour of a leader s has the same clock c_i and their corresponding arrow b_i pointed up, node s increments its clock value by 1 by executing rule **Incr Leader**.

After that, the clock value is propagated downward (toward nodes of greater d-value) using rules **Sync 1 down** and **Sync 2+ down**. Note that it is performed locally by layers: one node of a given d-value cannot update its clock value and arrow before every neighbour with a smaller d-value does so. This is necessary to guarantee the global synchronization of the clock.

There are two ways for the propagation of (c_i, b_i) to reach the limit of the area it should spread in: either it has reached nodes with d-value i, or there is no node having a greater d-value to spread the clock further.

- In the first case, rule **Sync end-of-chain** flips the arrow b_i.
- In the second case, the nodes execute rule **Sync 1+ up** to flip b_i.

In both cases, it allows rule **Sync 1+ up** to propagate upward (toward smaller d-values) with the b_i-value switching to $\uparrow$ from the nodes to their parents. Note that it is done locally by layers: one node of a given d-value may not update its clock value and arrow before every neighbour with a greater d-value has done so.

When the propagation reaches the neighbors of s, node s "detects" that its current clock value has been successfully propagated, and it will execute rule **Incr Leader** to increase it.

The point of this clock system is that two nodes under the same leader cannot have clock values out-of-sync by 2, but two nodes that have different leaders may. It allows them to detect a "collision" (*i.e.* two nodes of $S(C)$ too close from each other) when the d-values of two such nodes are smaller than $\lfloor \frac{k}{2} \rfloor$. Observe that the clock of index i is only reliable for detecting collision between nodes of $S(C)$ at distance $2i$ or $2i+1$ from each other. For smaller distances, this clock may be forcefully synchronized between two nodes of $S(C)$ by layer-by-layer updating, and for greater distances, no node may detect an out-of-sync from it. This process differs from the PIF mechanism [15]: we need to run one clock for each layer, as a clock of higher layer will be synchronized for the two conflicting leaders because of further nodes. Thus, we have $\lfloor \frac{k}{2} \rfloor - 1$ parallel clock systems to capture every possible distance of collision.

The Gouda daemon ensures that if two nodes of $S(C)$ are too close, this will only be the case for a while. The clock system will eventually detect it and propagate an error.

2.3 Handling Initial and Perturbed Configurations

Rules **Leader Down** and **Branch Incoherence** are only executed to solve problems coming from the initial configuration or after a perturbation has occurred. Rule **Leader Down** is executed when a leader has some of its arrows b_i in the wrong direction. Rule **Branch Incoherence** is executed when some "impossible" patterns are produced in the clock systems due to wrong clock values and arrows in the initial state. Standard patterns are shown in Figure 1. Any other pattern will make an activated node to execute rule **Branch Incoherence**.

3 Proof of the Algorithm

3.1 Stability of Legitimate Configurations

The ruling set algorithm presented in this section uses the state model. It constructs the set of vertices whose d-value is 0. We will prove that this set is a ruling set in legitimate configurations. Formally, we require the following specification for the legitimate configurations:

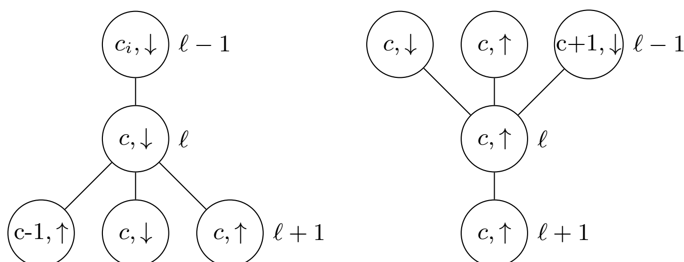

Figure 1 Branch coherence condition. The couples $(c, \downarrow)$ or $(c, \uparrow)$ represent the local variables (c_i, b_i) of the nodes. The value on the right of the node represents its distance to the leader node (i.e. its d-value). The central node in both figures is the reference, and the other nodes represent the possible couples for its neighbors with different d-value.

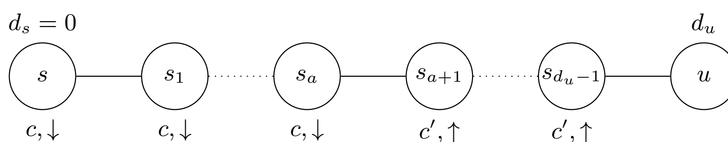

Figure 2 Node s propagates its clock value along a shortest path from s to u where $c' \in \{c, c-1\}$.

▶ **Definition 2.** *Let $S(C)$ be the set of nodes s such that $d_s = 0$ in a given configuration C. Configuration C is said to be legitimate if:*
1. *for any u we have $well_defined(u)$, $leader_down(u)$ and $branch_coherence(u)$ hold;*
2. *for any two distinct nodes u and v of $S(C)$, we have $dist(u, v) \geq k$.*

▶ **Theorem 3.** *The set of legitimate configurations is closed. Moreover, all the d-values do not change from a legitimate configuration C.*

Thanks to Theorem 3, we know that, from a legitimate configuration, we keep the same set of leaders $S(C)$, which forms a $(k, k-1)$-ruling set. Hence, under the Gouda daemon, the set of leaders will eventually be a stable $(k, k-1)$-ruling set.

The goal of the following lemmas will be to prove Theorem 3. Lemma 4 ensures that $S(C)$ forms a ruling set when the values of all the local variables are correct.

▶ **Lemma 4.** *Let C be a legitimate configuration. For any node u, $d_u = dist(u, S(C))$, and $S(C)$ is a $(k, k-1)$-ruling set of the underlying graph.*

Now we focus on the clock system. We prove the following property on the ruling set to run the clock system.

▶ **Lemma 5.** *Let C be a legitimate configuration and s be a node in $S(C)$. For every node u, $dist(u, s) \leq \lfloor \frac{k}{2} \rfloor$ implies that $d_u = dist(u, s)$.*

This property allows us to deduce that a node u such that $dist(u, s) \leq \lfloor \frac{k}{2} \rfloor$ has only one node s of $S(C)$ in its ball at distance $\lfloor \frac{k}{2} \rfloor$. Thus, all the nodes in $\mathcal{B}(s, \lfloor \frac{k}{2} \rfloor - 1)$ must be synchronized with s. We explain how the values representing the clock of the local variable of nodes with d-value smaller than $\lfloor \frac{k}{2} \rfloor$ are spread from their leader. Figure 2 illustrates how the pairs (c_i, b_i) go from nodes in $S(C)$.

▶ **Lemma 6.** *Let C be a legitimate configuration and s a node in $S(C)$. For every node u such that $dist(u, s) \leq \lfloor \frac{k}{2} \rfloor - 1$, every shortest path $(s_0, s_1, \cdots, s_{d_u})$ from s to u satisfies the following property in C:*

For every clock index $i \in [\![d_u+1, \lfloor \frac{k}{2} \rfloor - 1]\!]$*, there exists some integer* $a \in [\![0, d_u]\!]$ *such that:*
1. $\forall \ell \in [\![0, a]\!]$, $(b_{i,s_\ell}, c_{i,s_\ell}) = (\downarrow, c_{i,s})$*;*
2. $\exists c' \in \{c_{i,s} - 1, c_{i,s}\}, \forall \ell \in [\![a+1, d_u]\!]$, $(b_{i,s_\ell}, c_{i,s_\ell}) = (\uparrow, c')$*.*

Lemma 7 proves that only rules to update clocks are executed from legitimate configurations:

▶ **Lemma 7.** *Let C be a legitimate configuration. Let u be a node. Node u only executes stationary rules from C.*

Once the execution reaches a legitimate configuration C, we have proved that only stationary rules can be executed. The goal is to use that result and the previous lemmas to prove that only legitimate configurations can be reached from C. This result will lead to the proof of Theorem 3.

3.2 Reaching a Legitimate Configuration

The goal of the following lemmas is to prove that, from any configuration C, we can reach a configuration C' that is legitimate. The Gouda daemon's property concludes that a legitimate configuration will always eventually be reached. Indeed, let C be a configuration that is infinitely often reached during an execution. Under the Gouda daemon, as a legitimate configuration C' is reachable from configuration C, C' will also be reached infinitely often.

To that end, we introduce the notion of *locally legitimate* node for leaders satisfying conditions close to the legitimate ones in their ball of radius $k-1$. We prove that if a node s is locally legitimate, then it will remain so forever (Lemma 11).

We explain how to make locally legitimate a node with no leader at a distance smaller than k to it in Lemmas 13 and 16. We explain how, when some leaders are too close to each other, we can reach a configuration where none of the remaining ones are at distance smaller than k from another (Lemma 16).

From here, we can conclude with the proof of the following theorem:

▶ **Theorem 8.** *Under the Gouda daemon, any execution eventually reaches a legitimate configuration.*

We first introduce the notion we will use in this section for nodes in $S(C)$:

▶ **Definition 9.** *Let C be a configuration. A node s in $S(C)$ is locally legitimate if*
1. *all the nodes u in $\mathcal{B}(s, \lfloor \frac{k}{2} \rfloor)$ are such that $well_defined(u)$, $leader_down(u)$ and $branch_coherence(u)$ hold and $d_u = dist(u,s)$;*
2. *all the nodes u in $\mathcal{B}(s, k-1) \setminus \mathcal{B}(s, \lfloor \frac{k}{2} \rfloor)$ are such that $k - dist(u,s) \leq d_u \leq dist(u,s)$.*

We denote $\mathcal{LL}(C)$ the set of those nodes in C.

Let s be a locally legitimate node. The first property means that in its neighbourhood at distance at most $\lfloor \frac{k}{2} \rfloor$, nodes behave like in a legitimate configuration. Therefore, they cannot detect errors. The second property implies that all nodes in $\mathcal{B}(s, k-1)$ have coherent d-values according to s and to potential leaders at distance at least k from s. A direct observation is the following:

▶ **Lemma 10.** *Let $s \in \mathcal{LL}(C)$. We have $\mathcal{B}(s, k-1) \cap S(C) = \{s\}$.*

Combining Lemma 10 and the first property of the legitimated node, we can deduce that once a node is legitimate, it remains legitimate during the rest of the execution.

▶ **Lemma 11.** *Let C, C' be two configurations such that $C \rightarrow C'$. We have $\mathcal{LL}(C) \subset \mathcal{LL}(C')$.*

We focus now on how to create locally legitimate nodes. First of all, we can make sure that the d-values of all the nodes are coherent with regards to their distance to $S(C)$:

▶ **Lemma 12.** *For any configuration C, we can reach a configuration C' such that $S(C) = S(C')$, and $d_u = \min(dist(u, S(C')), k-1)$ for every node u, and there is no node with err-value 1 among nodes with d-value greater than $\lfloor \frac{k}{2} \rfloor$.*

Let s be a node at distance at least k from $S(C)$. We explain how to make that node locally legitimate:

▶ **Lemma 13.** *Let C be a configuration where there exists a node s such that $dist(s, S(C)) \geq k$. A configuration C' can be reached from C such that $s \in \mathcal{LL}(C')$.*

Now, we need to deal with leaders that are too close from each other. To do this, we introduce the function that measures the number of nodes in this situation in a configuration, and Lemma 15 shows how to decrease it.

▶ **Definition 14.** *Let C be a configuration. We define $\phi(C)$ as the set of leaders in C having a conflict with another one due to being at distance less than k to each other, i.e. $\phi(C) = \{u \in S(C) \mid \exists v \in S(C) \setminus \{u\}, dist(u, v) < k\}$.*

▶ **Lemma 15.** *Let C be a configuration such that $\phi(C) \neq \emptyset$. There exists a node u in $\phi(C)$ and a configuration C' such that we can reach C' from C with $S(C') = S(C) \setminus \{u\}$.*

Thanks to this result, we prove that we can reach a configuration C such that the set of conflicting nodes is empty:

▶ **Lemma 16.** *From any configuration C, we can reach a configuration C' such that $\phi(C') = 0$.*

Now we focus on how to make leaders locally legitimate if they do not have any other leaders at distance smaller than k from them.

▶ **Lemma 17.** *Let C and s be a configuration and a node such that $\mathcal{B}(s, k-1) \cap S(C) = \{s\}$. We can reach a configuration C' such that $s \in \mathcal{LL}(C')$.*

Now, we can prove that the number of legitimate nodes increases during the execution up until we converge to a legitimate configuration:

▶ **Lemma 18.** *Let C be a configuration. From C, we can reach a configuration C' such that either $\mathcal{LL}(C) \subsetneq \mathcal{LL}(C')$ or C' is legitimate.*

This last lemma allows us to conclude with the proof of Theorem 8.

4 From Ruling Sets to Distance-K Colorings

In this section, we focus on the *distance-K coloring* problem. A distance-K coloring is a coloring such that any pair of nodes cannot share a color unless they are at distance greater than K. If the nodes having the same color form a $(K+1, K)$-ruling set, then those nodes respect the coloring constraint.

Let choose $k > K$ for our $(k, k-1)$-ruling sets. We partition the set of nodes into two-by-two disjoint sets $S^{(i)}$ such that each set corresponds to nodes of the same color. We build these sets one after another. Each of these sets is a distance-k independent set of the graph, which is maximal among the nodes of $V \setminus \bigcup_{j<i} S^{(j)}(C)$. These sets will be built by composing an adaptation of our $(k, k-1)$-ruling set algorithm. Since the maximum degree

of the graph is Δ, any ball of radius $k-1$ contains at most $\Delta^{k-1}+1$ nodes. Hence we can partition the nodes into Δ^k ruling sets (we use this majoration in order to simplify the reading of the following proofs).

For this reason, the distance K-coloring algorithm is composed of Δ^k parallel algorithms, each one of them computing an adapted $(k,k-1)$-ruling set. For Algorithm i and configuration C, we note $S^{(i)}(C)$ (or $S^{(i)}$ if there is no ambiguity) the corresponding set $S(C)$. Each time a node u is active, it applies a rule (if it can) for each ruling set algorithm.

It is necessary to ensure that a node belongs to exactly one ruling set. To perform this, we number the ruling set algorithms: we denote by $d_u^{(j)}$ the local variable d_u of u of the j-th algorithm. By convention, we assume that u belongs to the j-th ruling set (or it has color j) if $j = min\{1 \leq p \leq \Delta^k \mid d_u^{(p)} = 0\}$. To form a partition with the sets, we need to reach a configuration where for each node u, $|\{i \leq \Delta^k \mid d_u^{(i)} = 0\}| = 1$. To achieve this, we modify rule **Become Leader** and add a rule to detect if a node is a leader in different layers (for Algorithm j).

Become Leader$^{(j)}$ :: (priority 1)
if $err_u^{(j)} = 0 \wedge (d_u^{(j)} = k-1) \wedge \forall v \in N(u), d_v^{(j)} = k-1 \wedge \forall p < j : d_u^{(p)} > 0$
then $d_u^{(j)} := 0$
$\forall i \in [\![1, \lfloor \frac{k}{2} \rfloor - 1]\!], c_{i,u}^{(j)} := 0, b_{i,u}^{(j)} :=\downarrow$

Belong To Two ruling sets$^{(j)}$ :: (priority 0)
if $d_u^{(j)} = 0 \wedge \exists p < j : d_u^{(p)} = 0$
then $d_u^{(j)} := 1$

We also modify the predicate $well_defined$ (for Algorithm j) as follows, which impacts the definition of legitimate configuration. In particular, now, a node u such that $d_u^{(j)} = k-1$ does not need to have a neighbor closer to a leader if $d_u^{(i)} = 0$ for some $i < j$.

$$\text{well_defined}^{(j)}(u) \equiv err_u^{(j)} = 0 \wedge \forall v \in N(u), |d_u^{(j)} - d_v^{(j)}| \leq 1 \wedge$$
$$((\forall p \leq j, d_u^{(p)} > 0) \vee d_u^{(j)} < k-1 \Rightarrow (\exists v \in N(u), d_v^{(j)} = d_u^{(j)} - 1)) \wedge (d_u^{(j)} = 0 \Rightarrow \forall p < j, d_u^{(p)} > 0)$$

We give a new definition of *legitimate configuration*:

▶ **Definition 19.** *Let $j \leq \Delta^k$. A configuration C is said to be legitimate for Algorithm j if, for all $i \leq j$:*
1. *for any u we have $well_defined^{(i)}(u)$, $leader_down^{(i)}(u)$ and $branch_coherence^{(i)}(u)$ hold;*
2. *for any $u \neq v$ in $S^{(i)}(C)^2$, we have $dist^{(i)}(u,v) \geq k$.*

From this, we get the following adaptation of Lemma 4. The proof remains slightly the same, with the exception that in the case of $d_u^{(j)} = k-1$, only nodes that have not a variable $d_u^{(i)} = 0$ for some $i < j$ are considered.

▶ **Lemma 20.** *Let C be a legitimate configuration for Algorithm j.*
- *For any node u, if for all $i < j$, $d_u^{(i)} > 0$, we have $d_u^{(j)} = dist(u, S^{(j)}(C))$;*
- *For any node u, if $d_u^{(i)} = 0$, for all $j > i$, we have $d_u^{(j)} = \min(dist(u, S^{(j)}(C)), k-1)$;*
- *$S^{(j)}(C)$ is a $(k, k-1)$-ruling set of $V \setminus \bigcup_{i<j} S^{(i)}(C)$.*

With these modifications, we have the following adaptation of Theorem 3:

▶ **Theorem 21.** *For all $j \leq \Delta^k$, the set of legitimate configurations for Algorithm j is closed. Moreover, from a legitimate configuration C for Algorithm j, all the $d^{(j)}$-value do not change.*

The proof to reach a legitimate configuration for Algorithm Δ^k works in the same way as the proof of Theorem 8. We need to do it one algorithm after another, from 1 to Δ^k. The main difference is that we only consider nodes that are not a leader in a smaller algorithm when we increase the set of locally legitimate nodes. This leads to the result:

▶ **Theorem 22.** *Under the Gouda daemon, any execution eventually reaches a legitimate configuration in Algorithm Δ^k.*

These two theorems lead to the main result of distance-K coloring:

▶ **Theorem 23.** *Let k and K be two integers such that $k > K$. Under the Gouda daemon, any execution eventually reaches a configuration C such that*

- $S^{(i)}(C) = \{u : d_u^{(i)} = 0\}$ *forms a distance-k MIS of* $V \setminus \bigcup_{j<i} S^{(j)}(C)$ *in G*
- *The sets $S^{(1)}(C), \dots S^{(\Delta^k)}(C)$ form a distance-K coloring.*
- *Every configuration in any execution starting in C verifies the two above properties with the same sets as C.*

5 Solving Mendable Problems

In this section, we want to solve a generalisation of *Greedy Problems*: $O(1)$-*Mendable Problems*, introduced in [8]. Greedy problems, such as $\Delta + 1$-coloring and Maximal Independent Set, have the property that if some of the nodes have chosen an output that is locally valid (no pair of neighbors sharing a color, no adjacent nodes selected in the set), then any single node can choose an output that will keep the global solution locally valid. In a distributed setting, we cannot do this process sequentially from one node to another, but we can do it in parallel: if a set of nodes that are far enough from each other choose their output at each step, the solution can be completed. The global solution is valid if we repeat this process until all nodes have chosen an output. To that end, we first introduce some definitions.

5.1 Definitions

We call a *Locally Checkable Problem* (LCL) Π a problem where each node can check locally that its output is compatible with its neighbours. Let $\mathcal{O}$ be the set of outputs. The output $\Gamma : V \to \mathcal{O}$ is good if and only if, for all $u \in V$, $\Gamma(u)$ is compatible with the multiset $\{\Gamma(v) \mid v \in N(u)\}$. For example, in the case of Maximal Independent Set, with $\mathcal{O} = \{0, 1\}$, 1 is compatible with $\{0^k \mid k \leq \Delta\}$, and 0 is compatible with $\{11^x0^y \mid x + y < \Delta\}$. Note that we can consider radius-r neighbourhood for the compatibility in the general case, which we will not do here out of simplicity. Our results can be adapted to the general version.

Let $\mathcal{O}$ be the set of outputs, and $\Gamma^* : V \to \mathcal{O} \cup \{\bot\}$. We say that Γ^* is a partial solution if, for any $u \in V$ such that $\Gamma^*(u) \neq \bot$, we can complete the labels of the neighbors v of u (i.e. give an output to the nodes v such that $\Gamma^*(v) = \bot$) to make u compatible with it neighbors.

A problem is T-*mendable* if, from any partial solution Γ^* and any $v \in V$ such that $\Gamma^*(v) = \bot$, there exists a partial solution Γ' such that $\Gamma'(v) \neq \bot$, $\forall u \neq v$, and $\Gamma'(u) = \bot \Leftrightarrow \Gamma^*(u) = \bot$, and $\forall u \in V$, $dist(u, v) > T \Rightarrow \Gamma'(u) = \Gamma^*(u)$. Intuitively, we can change the output of nodes at distance at most T from a node v when we select the output of v.

The LOCAL *model* is a synchronous model where each node is given a unique identifier. As there is no limit on the size of the messages for communication, after r rounds, each node knows the topology of their neighborhood at distance r.

▶ **Theorem 24** (Restated Theorem 6.2 from [8]). *Let Π be a T-mendable* LCL *problem. Π can be solved in $O(T\Delta^{2T})$ rounds in the* LOCAL *model if we are given a distance-$2T + 1$ coloring.*

One can observe that unicity of identifiers provided by the Local model is not necessary to solve an Lcl problem as long as nodes do not see twice the same identifier in the run. If we know that an algorithm runs on a graph of size at most n in $r(n) = o(\log n)$ rounds, then we can have it run on any graph of size at least n with a distance-$r(n)$ coloring, using those colors as the new identifiers. The algorithm will not notice that the identifiers are not unique, producing a correct output. This technique has been used, for example, in [8, 12].

Hence, for a constant T, we can produce a distance-$r(T)$ coloring to then use the algorithm of Theorem 24.

5.2 Solving Greedy and Mendable Problems

The goal now is to use distance-k colorings to solve other problems. Let us say we want to solve K-mendable problem Π for which we already have a Local algorithm $\mathcal{A}$ from Theorem 24 (the output of node u will be denoted out_u). To that end, we first build $\mathcal{A}'$, a self-stabilizing version of $\mathcal{A}$ that solves Π assuming unique identifiers at distance r. Then we compose $\mathcal{A}'$ with our distance k-coloring algorithm (for k big enough) - described in Section 4 - and obtain then a self-stabilizing anonymous algorithm solving Π. To simulate r rounds in the Local model, we need to compute the graph's topology at distance r for each node. To compute the output of node u, $\mathcal{A}'$ will compute the exact mapping of the ball of radius r centered on u. From it, $\mathcal{A}'$ will provide the output $\mathcal{A}$ would produce on this ball if the colors were identifiers.

In the following, we describe how each node will compute its ball. If we have beforehand a distance-$2r+1$ coloring, each node will have at most one node of some given color in its neighborhood at distance r. Hence, each node can compute a mapping of its neighborhood at distance r. At the beginning, each node knows its mapping at distance 0. If all the neighbors of a node u know their mapping at distance i, u can deduce its topology up to distance $i+1$. Note that we consider only cases where r does not depend on the size of the graph.

▶ **Lemma 25.** *Let C be a configuration where each node u has a color c_u corresponding to a distance-$2r+1$ coloring and outputs $out_u = \bot$. From this configuration, under the Gouda daemon, we will reach a configuration C' where each node outputs a mapping of their neighborhood at distance r.*

With this lemma and Theorem 24, we can conclude to the end result of this section:

▶ **Theorem 26.** *Let Π be an Lcl problem with mending radius k, that can be solved in $r = O(k\Delta^{2k})$ rounds in the Local model. Let C be a configuration where each node u has a color c_u corresponding to a distance-$2k+1$ coloring, a color c'_u corresponding to a distance-$2r+1$ coloring, and outputs $out_u = \bot$. From this configuration, under the Gouda daemon, we will reach a configuration C' where each node outputs a solution to Π.*

Note that in a ball of radius $2r+1$ in a graph of maximal degree Δ, there are at most Δ^{2r+1} nodes. Hence, we need Δ^{2r+1} colors. For graphs where Δ is constant, we get a constant number of colors. As we also consider constant radius r for the mendability, there are a finite number of possible mappings of balls at distance r using those colors. Hence, in that case, our algorithms use a finite memory that does not depend on the size of the graph.

6 Conclusion

This work provides a self-stabilizing algorithm under the Gouda daemon for any locally mendable problem by first introducing an explicit algorithm to compute a $(k, k-1)$-ruling set. This construction generalises well to probabilistic daemons if stationary rules and rule

Become Leader have some probability smaller than 1 to be activated. This algorithm permits building up distance-k colorings, which helps solve greedy and mendable problems by simulating the LOCAL model. In the case of constant bounded degree Δ, our algorithm uses a constant memory. We did not consider complexity questions. Considering a probabilistic daemon, an open question would be what complexities can be aimed, as our algorithm did not optimize this question at all.

The presented algorithm for the ruling set should adapt well in the Byzantine case, as the influence of a Byzantine node is naturally confined by the algorithm. In such context, Distance-K identifiers computed in Section 4 would be unique at distance K for nodes far enough from Byzantine nodes. It should be of interest to investigate this point.

References

1. Yehuda Afek and Shlomi Dolev. Local stabilizer. *Journal of Parallel and Distributed Computing*, 62(5):745–765, 2002.
2. Yehuda Afek, Shay Kutten, and Moti Yung. Local detection for global self stabilization. *Theoretical Computer Science*, 186(1-2):339, 1991.
3. Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. Introduction to distributed self-stabilizing algorithms. *Synthesis Lectures on Distributed Computing Theory*, 8(1):1–165, 2019.
4. Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.
5. Baruch Awerbuch, Andrew V Goldberg, Michael Luby, and Serge A Plotkin. Network decomposition and locality in distributed computation. In *FOCS*, volume 30, pages 364–369. Citeseer, 1989.
6. Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. *Journal of the ACM (JACM)*, 68(5):1–30, 2021.
7. Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed lower bounds for ruling sets. *SIAM Journal on Computing*, 51(1):70–115, 2022.
8. Alkida Balliu, Juho Hirvonen, Darya Melnyk, Dennis Olivetti, Joel Rybicki, and Jukka Suomela. Local mending. In *International Colloquium on Structural Information and Communication Complexity*, pages 1–20. Springer, 2022.
9. Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed (δ+ 1) -coloring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 437–446, 2018.
10. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM (JACM)*, 63(3):1–45, 2016.
11. Shimon Bitton, Yuval Emek, Taisuke Izumi, and Shay Kutten. Fully adaptive self-stabilizing transformer for lcl problems. *arXiv preprint*, 2021. `arXiv:2105.09756`.
12. Sebastian Brandt, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Patric RJ Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. Lcl problems on grids. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 101–110, 2017.
13. Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. *Distributed Computing*, 33(3):349–366, 2020.
14. Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.
15. Alain Cournier, AK Datta, Franck Petit, and Vincent Villain. Self-stabilizing pif algorithm in arbitrary rooted networks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 91–98. IEEE, 2001.

16 Alain Cournier, Stéphane Devismes, and Vincent Villain. Snap-stabilizing pif and useless computations. In *12th International Conference on Parallel and Distributed Systems-(ICPADS'06)*, volume 1, pages 8–pp. IEEE, 2006.

17 Shlomi Dolev. *Self-stabilization*. MIT press, 2000.

18 Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. *arXiv preprint*, 2011. arXiv:1110.0334.

19 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 270–277. SIAM, 2016.

20 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2904–2923. SIAM, 2021.

21 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 784–797, 2017.

22 Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 45–54, 1996.

23 Wayne Goddard, Stephen T Hedetniemi, David Pokrass Jacobs, and Pradip K Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 14–pp. IEEE, 2003.

24 Mohamed G Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, pages 114–123. Springer, 2001.

25 Nabil Guellati and Hamamache Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010.

26 Stephen T Hedetniemi. Self-stabilizing domination algorithms. *Structures of Domination in Graphs*, pages 485–520, 2021.

27 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. *SIAM Journal on Computing*, 50(3):STOC16–98, 2019.

28 Michiyo Ikeda, Sayaka Kamei, and Hirotsugu Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *the Third International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 70–74. Citeseer, 2002.

29 Shay Kutten and David Peleg. Fault-local distributed mending. *Journal of Algorithms*, 30(1):144–165, 1999.

30 Shay Kutten and David Peleg. Tight fault locality. *SIAM Journal on Computing*, 30(1):247–268, 2000.

31 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.

32 Alessandro Panconesi and Aravind Srinivasan. The local nature of δ-coloring and its algorithmic applications. *Combinatorica*, 15(2):255–280, 1995.

33 David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

34 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 350–363, 2020.

35 Sandeep K Shukla, Daniel J Rosenkrantz, S Sekharipuram Ravi, et al. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the second workshop on self-stabilizing systems*, volume 7, page 15, 1995.

36 Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys (CSUR)*, 45(2):1–40, 2013.
37 Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, 2007.
38 Volker Turau. Computing fault-containment times of self-stabilizing algorithms using lumped markov chains. *Algorithms*, 11(5):58, 2018.
39 Volker Turau. Making randomized algorithms self-stabilizing. In *International Colloquium on Structural Information and Communication Complexity*, pages 309–324. Springer, 2019.
40 Volker Turau and Christoph Weyer. Randomized self-stabilizing algorithms for wireless sensor networks. In *Self-Organizing Systems*, pages 74–89. Springer, 2006.