Stabilization Time in Weighted Minority Processes

Pál András Papp
ETH Zürich, Switzerland
apapp@ethz.ch

Roger Wattenhofer
ETH Zürich, Switzerland
wattenhofer@ethz.ch

Abstract

A minority process in a weighted graph is a dynamically changing coloring. Each node repeatedly changes its color in order to minimize the sum of weighted conflicts with its neighbors. We study the number of steps until such a process stabilizes. Our main contribution is an exponential lower bound on stabilization time. We first present a construction showing this bound in the adversarial sequential model, and then we show how to extend the construction to establish the same bound in the benevolent sequential model, as well as in any reasonable concurrent model. Furthermore, we show that the stabilization time of our construction remains exponential even for very strict switching conditions, namely, if a node only changes color when almost all (i.e., any specific fraction) of its neighbors have the same color. Our lower bound works in a wide range of settings, both for node-weighted and edge-weighted graphs, or if we restrict minority processes to the class of sparse graphs.

2012 ACM Subject Classification Mathematics of computing → Graph coloring; Mathematics of computing → Graph algorithms; Theory of computation → Distributed computing models; Theory of computation → Self-organization

Keywords and phrases Minority process, Benevolent model

Digital Object Identifier 10.4230/LIPIcs.STACS.2019.54

Related Version The full version of the paper is available online as arXiv preprint `https://arxiv.org/abs/1902.01228`.

1 Introduction

Given a simple graph and an initial coloring of its nodes, a minority process is a sequence of states (colorings) such that each state is obtained from the previous state by some of the nodes deciding to change their color. Each node, when it has the opportunity to act, switches to the least frequent color in its neighborhood. This may then prompt other neighbors of the node to switch their color, too, leading to a sequence of steps and a dynamically changing coloring. A state is stable when no node in the graph wants to change its color anymore, and the number of steps until a stable state is reached is known as the stabilization time of the process.

Minority processes have numerous applications in different areas where agents in a system are motivated to anti-coordinate with their neighbors. Assume, for instance, a set of wireless devices, each using a given frequency from a predefined set of frequencies for communication. In order to minimize interference with their neighbors, each device may repeatedly decide to switch to the frequency which is the least used in its neighborhood. In another setting, assume that some companies need to decide which product or commodity to produce, and they repeatedly adjust their strategy to avoid competition with specific other companies (that are e.g. geographically close, or share the same costumer base) [16]. Minority processes also appear in a wide range of other areas, including cellular biology [10], physics [6, 7] and social sciences [9].

36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019).
Editors: Rolf Niedermeier and Christophe Paul; Article No. 54; pp. 54:1–54:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

It is often quite natural to model such settings not only as graphs, but as weighted graphs, since in many applications, either the nodes or edges of the graphs naturally exhibit some kind of weights that define their importance in the minority setting. For example, when selecting products, some competitors may be larger or more resourceful than others, and thus it is more crucial for their neighbors to differentiate from these specific nodes. In the frequency allocation setting, some nodes may handle much more traffic than others, and thus it is more important to avoid interference with such neighbors. Frequency allocation also provides a natural example for edge weights, since the severity of interference can also depend on the distance between neighboring devices, and thus it might be more imperative for nodes to avoid interference with closer neighbors.

The paper considers minority processes in these weighted cases, when the cost function of a node to minimize is not simply the number of its conflicts, but the sum of these conflicts multiplied by the weight of the neighboring node or by the weight of the connecting edge. In such a weighted setting, the only straightforward upper bound on the number of steps is exponential. In this paper, we prove an asymptotically matching lower bound of $2^{\Theta(n)}$, showing that there are weighted graphs where stabilization can indeed last for an exponential number of steps.

For a realistic analysis of stabilization time in applications, some further aspects of the processes are also worth studying. To avoid unreasonably many switches, nodes may decide not to switch color if this benefit is too small. Thus it is often more reasonable to assume a proportional switching rule in the weighted setting, i.e. that a node only decides to change its color if this reduces its cost at least by a given fraction of its weighted degree (or, equivalently, if a large fraction of its neighborhood has the same color). Note that this is a significantly stricter switching rule, and thus proving a lower bound on the number of steps under this rule is a stronger result. Furthermore, in most application areas, the underlying graphs are sparse, i.e. contain only $O(n)$ edges, so it is also interesting to study if the behavior is different when restricting ourselves to sparse graph instances.

There are multiple different models to study minority processes, sequential and concurrent alike. Even in the sequential setting, when only one node switches in each step, we can observe different behaviors depending on the order in which the nodes are selected. For example, this order may be chosen by a benevolent player who aims to minimize stabilization time, or an adversarial player aiming to maximize it. While stabilization time in these models have been studied thoroughly in the related area of majority processes, stabilization time in minority processes has remained open.

In the paper, we present weighted graph constructions that prove an exponential lower bound on stabilization time. Our lower bound holds both for node-weighted and edge-weighted graphs, for any number of colors, and also if we restrict the process to the class of sparse graphs.

The main contributions of the paper are as follows. We first present a construction that shows an exponential lower bound in the adversarial model. Then with further improvements to the construction, we prove that the same bound also holds in the benevolent model. This shows that there are graphs where not only one, but every possible run of the process takes exponential time. Moreover, we also show that the lower bound holds not only for the sequential process, but also in any reasonable concurrent setting. Our lower bounds are shown for a very strict switching rule, when a node is only allowed to switch if a given fraction of its neighbors have the same color. Most surprisingly, our results show that even with this rule, the exponential lower bound holds for any non-trivial fraction of the neighborhood.

2 Related Work

The question of stabilization time has only been studied in detail for majority processes. In [15], the authors devise a weighted graph construction, which exhibits a majority process with $2^{\Theta(n)}$ stabilization time both in the synchronous and the adversarial sequential models (benevolent models are not discussed in this paper). For the unweighted case, the stabilization time of majority processes has been characterized by [11] in the synchronous, sequential adversarial and sequential benevolent models. The study of [15] also shows further results on some slightly different variants of majority processes in unweigthed graphs. On the other hand, apart from a straightforward $O(n^2)$ upper bound in the unweighted case [14, 15], to our knowledge, the stabilization time of minority processes in these models has remained open so far.

However, for unweighted graphs, there are numerous theoretical studies that focus on different properties of stable states, both in case of minority [16, 3, 21, 1, 8] and majority [3, 12, 13, 20, 4, 2, 5] processes.

Minority processes have also been thoroughly studied in special classes of graphs, such as grids, trees or cycles, by the cellular automata community [17, 18, 19]. However, these results work with unweighted graphs, and a different variant of the minority process which considers the closed neighborhood of nodes. Besides the theoretical results, some of these studies also include an experimental analysis of the process on grids.

Papers working with minority processes almost always consider the basic switching rule, i.e. when nodes switch color for any small amount of improvement (although they sometimes assume different rules for tie-breaking). Some slightly different switching rules, based on distance-2 neighborhood of nodes, are examined in [14]; however, the aim of these modified rules is not to achieve earlier stabilization, but to reduce the number of conflicts in the final (stable) state. To our knowledge, however, minority processes have not yet been studied under the proportional switching rule.

3 Models and Notation

Preliminaries and notation

In the paper, we consider simple, undirected graphs, denoted by $G = (V, E)$, with V being the set of nodes and E the set of edges. The number of nodes is denoted by n, the edge between vertices u and v is denoted by $e(u, v)$. In case of *node-weighted graphs*, we assume a positive weight function $w : V \to \mathbb{R}^+$ on the nodes of the graph, while for *edge-weighted graphs*, we assume $w : E \to \mathbb{R}^+$ on the edges.

For a specific node $v \in V$, we denote by $N(v)$ the neighborhood of v. In case of node-weighted graphs, for a set $S \subseteq V$, we denote by W_S the sum of weights $\sum_{u \in S} w(u)$. Specifically, we use $W_{N(v)}$ to denote the sum of weights in v's neighborhood.

Given a set of colors Γ, a *coloring* is a function $C : V \to \Gamma$. If for some edge $e(u, v)$ we have $C(u) = C(v)$, then we have a *conflict*, and the edge in question is a *conflicting edge*. Generally, the goal of graph coloring is to minimize the number of conflicts in the graph.

We also use the notation $N_S(v) := \{u \mid u \in N(v) \text{ and } C(u) = C(v)\}$ and $N_O(v) := N(v) \setminus N_S(v)$ for a node v under a coloring C (the *same-color* and *other-color neighborhood* of v, respectively). Note that since we will use these notions in regard to a state of the process (a current coloring of G), we assume that the coloring function C is clear from the context, and thus it is not included in the above notation for simplicity.

In both weighted settings, we have a natural cost function f for each node v of the graph. In node-weighted graphs, we define $f(v) = \sum_{u \in N_S(v)} w(u)$, while in the edge-weighted setting, we define the cost function as $f(v) = \sum_{u \in N_S(v)} w(e(u,v))$. The aim of nodes in the minority process is to minimize this cost function. For a color $c \in \Gamma$, let $f_c(v)$ denote the cost that node v would have if it was recolored to color c, with the colors of all nodes in $N(v)$ remaining unchanged. Let us denote the preferred color of v by $c^* = \arg\min_c f_c(v)$; in case of multiple minimal values, we select an arbitrary one of them as c^*. When v *switches*, it changes its color to c^*. If $f(v) - f_{c^*}(v)$ is above a given threshold, or more generally, if the relation of $f(v)$ and $f_{c^*}(v)$ satisfies a specific condition known as the *switching rule*, then v is *switchable*.

A *minority process* on G is a sequence of colorings S_0, S_1, ..., known as *states*, where, except for S_0, each state S_i can be obtained from S_{i-1} by switching a set of nodes that are switchable in S_{i-1}. The state S_0 is referred to as the *initial state*. Given a graph and an initial state, the set of nodes to be switched in each step (and thus the entire sequence of states) is determined by the *model*, as discussed below.

We say that a state S_i is *stable* if there are no switchable nodes in S_i. A process *stabilizes* if it reaches a stable state; the number of steps until the process stabilizes is the *stabilization time* of the process.

While presenting our construction, we assume node-weighted graphs and $|\Gamma| = 2$ available colors. Section 4 discusses how to generalize our lower bound to edge-weighted graphs or more than 2 colors.

Models

We consider minority processes in the following models:

- **Sequential Adversarial (SA):** In each step, only one node switches. This node is chosen by an adversarial player, who aims to maximize the stabilization time.
- **Sequential Benevolent (SB):** In each step, only one node switches. This node is chosen by a benevolent player, who aims to minimize the stabilization time.
- **Concurrent Benevolent (CB):** In each step, the benevolent player can switch any set of switchable nodes concurrently, in order to minimize the stabilization time.

There are many further popular models of minority processes, for example, with synchronous or randomized behavior. However, these models always exhibit a larger stabilization time than model CB, since in model CB, the benevolent player is free to choose any sequence of (possibly concurrent) steps to minimize stabilization time, and thus he can also simulate the behavior of any of these additional models. Therefore, a lower bound for model CB also implies the same bound in these various other models.

Note that in concurrent models, it is possible that neighboring nodes repeatedly force each other to switch at the same step, cycling through the same colors infinitely. Because of this, related studies in the synchronous model often use an alternative definition of stabilization, also considering a periodically repeating process to be stable. However, the design of our benevolent construction ensures that connected nodes can never be switchable at the same time, and thus in our graphs, even in concurrent models, the process always terminates in a fixed state. Nonetheless, our lower bound also holds with this alternative, more permissive definition of stabilization.

Our lower bound construction for model SA is shown in Section 5. Then Section 6 describes how to extend this construction to the case of model SB. Once we present our construction for model SB, it will follow that this same construction also proves the lower bound in model CB. As the construction heavily restricts the set of selectable sequences, always allowing only a few switchable nodes in the graph, even in model CB, the benevolent player has no other option than to execute exactly the same steps as in the sequential case, possibly some of them at the same time. On the other hand, the construction will have specific nodes that alone switch $2^{\Theta(n)}$ times, and thus even with some of the steps executed simultaneously, stabilization takes $2^{\Theta(n)}$ steps.

Switching rules

Most of the related work studies the following switching rule:

Rule I (*Basic Switching*): v is switchable if $W_{N_S(v)} - W_{N_O(v)} > 0$.

Here we introduce a stricter switching rule, based on a real parameter λ (where $0 < \lambda < 1$):

Rule II (*Proportional Switching*): v is switchable if $W_{N_S(v)} - W_{N_O(v)} \geq \lambda \cdot W_{N(v)}$.

This alternative switching condition is reasonable in many settings where switching comes with a certain cost for the node, and therefore, it is only beneficial when this allows the node to reduce its cost considerably, i.e. by a given factor of $W_{N(v)}$. Since we have $W_{N_S(v)}+W_{N_O(v)} = W_{N(v)}$ in the case of two colors, this condition is equivalent to $W_{N_S(v)} \geq \frac{1+\lambda}{2} \cdot W_{N(v)}$, i.e. that a node is only allowed to switch if $\frac{1+\lambda}{2}$ fraction of its (weighted) neighborhood has the same color. Therefore, if λ is close to 1, then Rule II intuitively means that in order to switch v twice, we also have to switch *almost every* neighbor of v in the meantime to make v switchable again for the second time.

While the above definition of Rule II is more intuitive, for the analysis, it is often convenient to express Rule II in another alternative form: v is switchable if $W_{N_S(v)} \geq \Lambda \cdot W_{N_O(v)}$, for some other constant Λ. One can show that this is equivalent to the definition with a choice of $\Lambda := \frac{1+\lambda}{1-\lambda}$. We will mostly use this alternative Λ parameter throughout our analysis.

Our technique proves the lower bound for Rule II with any $\lambda < 1$. However, for ease of presentation, we are first going to describe our construction for a specific parameter value of $\lambda \approx \frac{2}{3}$. Note that $\lambda = \frac{2}{3}$ corresponds to 5 in the Λ-notation; let us introduce the new notation $\Lambda_B := 5$ for this base value. We need this extra notation because the construction we present is actually not for $\Lambda = 5$, but in fact only for $\Lambda = 5 - \epsilon$ with any $\epsilon > 0$, hence proving the lower bound for Rule II with any $\Lambda < 5$ (or, using the λ-notation, for any $\lambda < \frac{2}{3}$). Note that we have specifically chosen $\lambda > \frac{1}{2}$ for demonstration because some challenges in the construction are easier if $\lambda \leq \frac{1}{2}$.

Given the proof of the lower bound for $\Lambda = 5 - \epsilon$ with any $\epsilon > 0$, we then discuss how to generalize the same construction technique for any other odd integer Λ_B as a base value. This proves the lower bound for $\Lambda = 7 - \epsilon$, $\Lambda = 9 - \epsilon$, and so on, with any $\epsilon > 0$.

Note that $\lim_{\Lambda_B \to \infty} \lambda = 1$, that is, as Λ_B goes to infinity, the λ value corresponding to $\Lambda_B - \epsilon$ gets arbitrarily close to 1 (this follows from the fact that λ can be expressed as $\frac{\Lambda-1}{\Lambda+1}$, by the definition of Λ). Therefore, we can obtain any $\lambda < 1$ value with an appropriate odd integer Λ_B and appropriate $\epsilon > 0$, and since our construction can be generalized for $\Lambda_B - \epsilon$ with any such Λ_B and ϵ, this already establishes the lower bound for every $\lambda \in (0, 1)$.

While it is not required for our lower bound proof, the full version of the paper also presents a general method to prove the monotonicity of the lower bound: that is, for any λ_0 and $\lambda < \lambda_0$ values, given a construction for λ_0, there is a straightforward way to convert it into a construction for λ. Note that this monotonicity is trivial in the adversarial case: since any node that is switchable for Rule II with λ_0 is also switchable for the rule with λ, the construction for λ_0 is, without any change, also a valid construction for λ, exhibiting the same stabilization time. The case is, however, not this simple for benevolent models, where a lower λ value may allow a wider set of moves for the benevolent player, which might reduce the stabilization time significantly. Monotonicity in this model can be shown using so-called fixed nodes; see the full version for a discussion.

Helpful tools and definitions

We say that a node v is *dominated* by a subset $S \subseteq N(v)$ if $W_S \geq \frac{\Lambda}{\Lambda+1} \cdot W_{N(v)}$, that is, if S having the same color as v is enough to make v switchable. If v is dominated by a single-node subset $\{u\}$, then we say that v is a *follower node*, and u is the *dominant node* of v; this implies that the preferred color of v is always simply the opposite of u's color.

One tool we will frequently use in our constructions is the addition of so-called *fixed node neighbors*. A fixed node is a node that is added to the graph construction in a way that ensures it can never become switchable throughout the process, and thus always keeps its initial color. This can easily be achieved by adding a black and a white *stabilizer node* to the graph, and connecting each fixed node to the stabilizer of the opposite color. If we then assign significantly larger weights to the stabilizer nodes than to all other nodes in the graph (i.e., sufficiently large weights such that each fixed node is a followers of its (opposite-colored) stabilizer node neighbor), then the fixed nodes can indeed never switch throughout the process.

In our construction, each fixed node we add is only connected to one specific node v, and its only purpose is to influence the behavior of v in the process (i.e., make it easier or harder to switch v to a specific color). We may add a separate black and a white fixed node neighbor (with any desired weight) to every node v of the construction. However, note that it makes no sense to add more than two fixed neighbors to a node v: if we were to add two same-colored fixed neighbors to v, we could simply combine the two into one fixed neighbor with the sum of the two weights. Therefore, the use of fixed node neighbors adds at most $2n + 2$ extra nodes to the graph, only changing the magnitude of n by a constant factor, and thus it does not affect the exponential nature of stabilization time.

4 Basic Observations

Node or edge weights

We consider minority processes on both node-weighted and edge-weighted graphs. Note that edge weights have at least as much (in fact, more) expressive power than node weights: assume that we have a graph G with some node weights $w(v)$, and consider the edge-weighted graph that consist of the same nodes and edges, and edge weights are defined as $w(e(u, v)) = w(u) \cdot w(v)$. A minority process in this derived graph behaves the exact same way as in the original, node-weighted graph: for any node v, each neighbor $u \in N(v)$ stands for a $\frac{w(u)}{W_{N(v)}}$ portion of $W_{N(v)}$ in the node-weighted case, and u contributes exactly the same $\frac{w(u) \cdot w(v)}{W_{N(v)} \cdot w(v)}$ portion in the derived edge-weighted graph.

This implies that for any node-weighted graph, we can create a corresponding edge-weighted graph with the same stabilization time, regardless of the model. Therefore, when showing the lower bounds of the paper, we only consider node-weighted graph constructions. Our observations imply that the same lower bound will then also hold for edge-weighted graphs.

Number of colors

The constructions in the paper assume there are only two available colors: *black* and *white*. However, it is simple to generalize the lower bound to any number of colors. The main idea is to take the lower bound construction for 2 colors, and for each node of the graph and for every additional color, add an extra neighbor with high weight having this color. The process in the resulting graph will behave as if the graph only consisted of the original nodes and the original two colors. A detailed discussion of the technique is available in the full version of the paper. The method allows us to generalize the lower bound not only to any constant number of, but also up to $\Theta(n)$ colors.

Matching upper bound

While the proof of exponential lower bound is quite involved, it is straightforward to show an exponential upper bound on stabilization time in sequential models. To discuss this upper bound, we briefly return to the case of edge-weighted graphs, as they can exhibit a wider set of behaviors. Since for each node-weighted graph there exists an edge-weighted graph with the same stabilization time, the upper bound on edge-weighted graphs immediately implies the same upper bound on node-weighted graphs.

In an edge-weighted graph, for each state (i.e., coloring of the graph), we can define a *potential* value as the sum of $w(e)$ for all edges e in the graph that are currently conflicting. In sequential models when only one node switches in one step, this potential strictly decreases after every step, since the incentive of the nodes is exactly to reduce the potential in their neighborhood. This allows for a simple upper bound on stabilization time in sequential models: since each state has a fixed potential value and potential is monotonously decreasing throughout the process, each state can be visited at most once. For the case of 2 colors, there are 2^n distinct possible states, which implies that stabilization time is upper bounded by 2^n.

5 Construction for the Adversarial Case

We first present a graph construction to show the exponential lower bound in model SA.

▶ **Theorem 1.** *For Switching Rule II with any $\lambda < 1$, there exists a class of (sparse) weighted graphs with $2^{\Theta(n)}$ stabilization time in model SA.*

While the theorem holds for any $\lambda < 1$, recall that we present the construction for a concrete value of $\lambda \approx \frac{2}{3}$ (that is, $\Lambda = 5 - \epsilon$ for some small $\epsilon > 0$).

Throughout the presentation of our construction, nodes that are shown vertically higher in figures will always have larger weight than nodes that are placed below. Based on this, we also refer to neighbors of nodes as upper or lower neighbors. We will define the weight of each node in the graph as a function of the weights of the nodes below. As such, one can determine a concrete set of node weights for the construction by following these rules in a bottom-to-top fashion, with the lowermost weights chosen arbitrarily.

The basic idea behind our construction is recursive, and as such, the resulting graph consists of multiple *levels*. Given a construction that exhibits a sequence which switches some specific nodes of the graph s times at least, we show how to extend this graph with a constant number of new nodes (a next level) to obtain another construction where, with the correct choice of sequence, a specific new set of nodes switch $\frac{3}{2}s$ times. With a repeated application of this step, after adding ℓ levels, we obtain a set of nodes that switch $\left(\frac{3}{2}\right)^{\ell} \cdot s$ times. Since each new level consists of only $O(1)$ nodes, our graph can contain linearly many levels, yielding a final construction with $2^{\Theta(n)}$ switches.

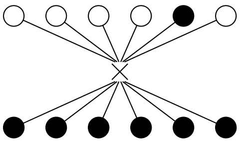

Figure 1 A 6-tuple of base nodes (below) and control nodes (above). The symbol × denotes a complete bipartite connection.

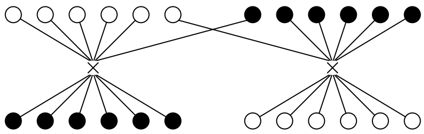

Figure 2 Final structure of a level, with two distinct 6-tuples of base and control nodes.

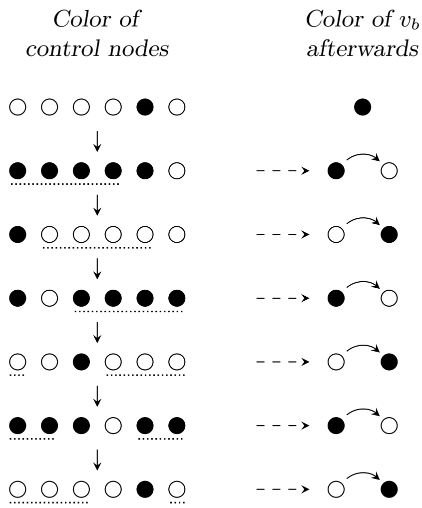

Figure 3 A control sequence of 6 steps, each time switching a 4-node subset of the control nodes (marked by a dotted line). The resulting switch of the base nodes is shown on the right.

The key nodes of our graph are the *base nodes*, which appear in 6-tuples with the same weight and same initial color. Each 6-tuple of base nodes has 6 common upper neighbors, known as the *control nodes* for these base nodes, forming a complete bipartite graph. The two 6-tuples together comprise a level of our construction (see Figure 1).

The 6 control nodes in a level also all have the same weight; let us denote this weight by $w(v_c)$. The main idea of the construction is to choose $w(v_c)$ sufficiently large such that 5 of the 6 control nodes already dominate each of the base nodes below. Assuming that one of the base node v_b has further (lower) neighbors of weight w_L altogether, this requires $5 \cdot w(v_c) \geq \Lambda \cdot (w(v_c) + w_L)$ to hold, which can be ensured by a choice of $w(v_c) \geq \frac{5-\epsilon}{\epsilon} \cdot w_L$ for our current $\Lambda = 5 - \epsilon$. Thus we can select sufficiently large weights such that a base node v_b is indeed switchable whenever 5 out of 6 control nodes have the same color as v_b.

Note that from the initial state shown in Figure 1, we only need to switch 4 of the 6 control nodes (from white to black) in order to force a base node v_b below to switch to white. In fact, we can specify a sequence of 4-node subsets of the control nodes such that every time we switch the next subset in the sequence, we once again have 5 control nodes with the same color that v_b currently has, and therefore v_b can be switched again. A possible such sequence is shown in Figure 3; we refer to this as the *control sequence*. The sequence has a couple of convenient properties: each control node is switched exactly 4 times throughout the sequence, and each control node (and also v_b) returns to its initial color at the end of the sequence.

This is exactly the technique that allows us to increase the number of switches by a factor of $\frac{3}{2}$ within each level of the construction. If the upper levels provide a way to switch each of the 6 control nodes in the current level s times, then this allows us to execute the control sequence $\frac{s}{4}$ times, and each such execution switches the base nodes in the current level 6 times, adding up to $\frac{6}{4}s$ switches for each of the 6 base nodes.

It only remains to connect the different levels of our recursive construction. It comes as a natural first idea that the 6-tuple of base nodes in a level could also directly take the role of the control nodes in the level below. The first difficulty to overcome with this approach

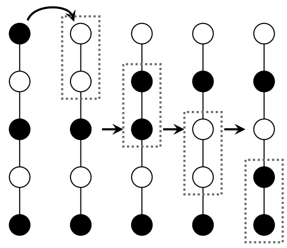

Figure 4 When a conflict is created at the top of the chain, then switching the nodes one by one propagates this conflict down through the chain.

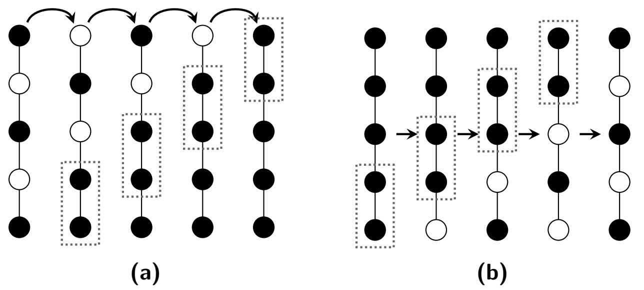

Figure 5 When charging (a), we propagate each new conflict to the next position (Figure 4 shows the first step of (a) in detail). When unloading (b), we always propagate the lowermost stored conflict to the bottom.

is the color of the nodes in question: while all 6 base nodes of a level have the same color (say, initially black), the control nodes initially have mixed color (5 white and 1 black) in the control sequence. We can overcome this by duplicating the structure in Figure 1 in the opposite initial color, and redefining a level as these two bipartite graphs together. Since a level now consists of 12 base nodes, 6 white and 6 black initially, we can reorganize these nodes into two appropriate groups (5 white + 1 black, 5 black + 1 white) to act as the control nodes of the next level (see Figure 2).

There is a further problem with using the base nodes directly as the control nodes of the level below: our level design only provides a way to switch a 6-tuple of base nodes *together* (that is, consecutively in any order). However, in order to execute the control sequence, we need to be able to switch specific subsets of the control nodes. For example, in the sequence of Figure 3, the second node from the left has already switched twice before the rightmost node ever switches. Thus, the fact that we can switch both 6-tuples of base nodes s times does not yet imply that we can switch specific 4-node subsets of them in the given order, as needed for the control sequence.

To provide a way to switch the control nodes in any order of our choice, we connect the levels of the construction with tools known as *storage chains*. A storage chain is a path of 5 nodes, initially colored in an alternating fashion. The weights of the nodes in the chain are chosen such that each node is a follower node of its upper neighbor (this can be ensured by defining node weights in a bottom-to-top fashion, always choosing sufficiently large weight for the next node). The uppermost and lowermost nodes may have other upper and lower neighbors outside of the chain, respectively.

Assume now that the topmost node in the chain is switched by some external condition (i.e., its upper neighbors outside of the chain). This introduces a conflict into the chain between the uppermost two nodes, as shown in Figure 4. However, recall that by our definition of node weights, the second node (from the top) is a follower of the uppermost node, and therefore this conflict makes the second node switchable. Switching the second node (to black) resolves the original conflict, but creates a new conflict between the second and third nodes instead (now making the third node switchable). Generally, whenever there is a conflicting pair of subsequent nodes above an alternating-colored (part of the) chain, we are able to switch the lower node, and thus move the conflict down to the next node pair in the chain. We can use this method to move a conflict down to any point in the chain, as shown in the figure; we refer to this process as *propagating down* the conflict in the chain.

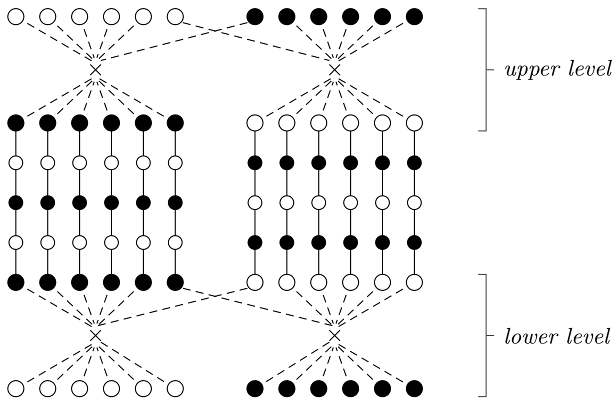

Figure 6 Two levels of the construction, connected by storage chains (edges within a level are shown in dashed). For simpler illustration, the two sides of the lower level are horizontally swapped.

With this technique, we can accumulate and store conflicts in the chain "for later use". If the uppermost node is forced to switch 4 times, then we can propagate down each of the emerging conflicts to a different position (i.e., pair of nodes) in the chain, ending up with 4 conflicts in a completely monochromatic chain. This process (see Figure 5a) is referred to as *charging* the chain. In another sequence of steps, we can then *unload* the chain and propagate these conflicts one by one to the bottom of the chain, essentially using the stored conflicts to switch the lowermost node 4 times in a timing of our own choice (see Figure 5b). When the sequence is finished, each node in the chain once again has its original color.

We use such storage chains to connect subsequent levels of our construction, with the base nodes and control nodes being the uppermost and lowermost nodes in the chains, respectively, as shown in Figure 6. This way, every time after the 6-tuple of base nodes in the upper level switch (together), we can execute the next step in charging each of the storage chains. After each of the base nodes switch 4 times, each of the storage chains are charged. Then, by unloading each chain in 4 steps in the order of our choice, we can switch each of the control nodes below 4 times, in any preferred order; this enables us to execute the control sequence on the lower level. Thus, if the upper-level base nodes are switched 4 times, we can indeed switch the lower-level base nodes 6 times.

For a high-level overview of the process, the execution of the adversarial sequence on a given level L could be summarized by the following recursive pseudocode:

```
Function ProcessLevel(L)
    For each of the 6 steps of the control sequence:
        On both sides, switch the next subset of 4 control nodes
        Switch all 6 + 6 base nodes
        Propagate down the conflict in each chain as far as possible
        If the chains below are fully charged:
            Call ProcessLevel(L + 1) (execution continues on level below)
    Return (execution continues on level above)
```

Even with the storage chain connections, the addition of each new level increases the number of nodes only by a constant value. This implies that a graph on n nodes can contain $\Theta(n)$ levels, and thus each node in the lowermost level indeed switches $2^{\Theta(n)}$ times.

There is one more detail to discuss: for convenience, we assumed that the number of switches s in an upper level is always divisible by 4. However, s switches in each control node in fact allows for only $\lfloor \frac{s}{4} \rfloor$ complete executions of the control sequence, and hence $\lfloor \frac{s}{4} \rfloor \cdot 6$ switches for the base nodes. Nonetheless, this still implies exponential increase for s large enough (for example, $\lfloor \frac{s}{4} \rfloor \cdot 6 \geq \frac{6}{5}s$ holds if $s \geq 20$). Thus to overcome this problem, we ensure that the control nodes in the uppermost level already switch 20 times; this is achieved by adding an initially charged storage chain of 21 nodes above each uppermost control node. Unloading the chains allows us to switch these top-level control nodes 20 times in the preferred order, and thus the exponential increase of switches is guaranteed.

This proves our lower bound in model SA for the case of Rule II with $\Lambda = 5 - \epsilon$ for any $\epsilon > 0$. However, the construction is straightforward to generalize to any other odd integer Λ_B: for most of the analysis, one only needs to replace the value 4 by $(\Lambda_B - 1)$ and the value 6 by $(\Lambda_B + 1)$. This provides a construction with $(\Lambda_B + 1)$-tuples of control and base nodes, and a $\frac{\Lambda_B+1}{\Lambda_B-1}$ factor of increase in switches for every new level. The control sequence can also be generalized for other Λ_B values; details of the generalization are discussed in the full version of the paper.

6 Benevolent Case

It is significantly more difficult to show an exponential lower bound for benevolent models, since such a construction needs to guarantee that every possible sequence lasts for an exponential number of steps. We overcome this problem by heavily restricting the set of selectable sequences in the graph. Specifically, we start from the construction of Section 5, and we show how to add a set of extra nodes which ensure that the previously defined sequence is the only possible sequence the benevolent player can choose. In this section, we outline the main ideas of this benevolent construction; a detailed discussion of the technique is provided in the full version of the paper.

▶ **Theorem 2.** *For Switching Rule II with any $\lambda < 1$, there exists a class of (sparse) weighted graphs that have $2^{\Theta(n)}$ stabilization time in the benevolent models (models SB and CB).*

We basically use two tools (gadgets) to ensure that the player, when selecting the sequence, has to follow the procedure described in the pseudocode above. On the one hand, we show how to build logical AND *gates* and OR *gates*, in order to check that a given step of the procedure is reached, and use these gates to allow the player to proceed to the next step of the procedure. On the other hand, we devise a *state chain* in order to keep track of the current phase of the procedure, which can then be used as a condition in the logical gates that control the execution of the procedure.

With the appropriate combination of these two gadgets, we can ensure that the benevolent player has no other option than to switch the control nodes, base nodes and storage chain nodes in the order described by the recursive procedure. We add a separate such combination of these gadgets to each level of the construction of Section 5. However, since in our recursive procedure, each level of the graph executes the same sequence of steps multiple times (the lower levels exponentially many times), the design of these gadgets also needs to ensure that the gadget can execute its task multiple times. This is achieved through introducing a method to repeatedly "reset" the gadgets to their initial state.

For the purpose of resetting these gadgets, we introduce another tool, the third main ingredient of our benevolent construction, known as a *pacer system.* The main idea of the resetting technique is to connect each gadget (logical gate or state chain) to so-called *pacer* *nodes* higher in the graph, and to ensure that each such pacer node switches at least twice between two consecutive times of using the gadget. The gadgets are designed in a way which guarantees that this pacer node switching twice results in the gadget being reset to its default state (i.e., each node to its initial color).

Such a pacer node essentially "recharges" the gadget with conflicts: since the weighted sum of conflicts in the graph monotonically decreases, the gadget can only return to the same (initial) state repeatedly if it "acquires" new conflicts from some other part of the graph. This is achieved through the connection to the pacer node, which is in a higher level of the graph (with larger weights), and thus has significantly more conflicts to "push down" into the gadget as a byproduct of its switching.

The simplest way to add pacer nodes to our construction is to place a pair of them between a set of control and base nodes, as shown in Figure 7. In this modified level version, the steps of the control sequence do not switch the base nodes directly. Instead, this happens indirectly: after 5 of the 6 control nodes are black, first the upper pacer node, and then the lower pacer node switches, followed by the base nodes in the end. Thus, the addition of pacer nodes leaves the general behavior of the level unchanged: the base nodes will still switch eventually after each step of the control sequence. However, in this new level construction, the newly added pacer nodes will also both switch in each of these steps.

The actual pacer systems used in our construction are more sophisticated constructions based on this idea. They consist of multiple pacer nodes in order to be able to recharge gadgets of both colors, and they are also responsible for checking that the recharging process has indeed been executed on the connected gadgets.

Given the technique to reset gadgets, it only remains to briefly present the behavior of the two gadgets (logical gates and state chains), and to outline how they are used in the construction. For the convenient description of gadgets, we first introduce two special kinds of node concepts. Essentially, these are methods to carefully select the weight of some specific neighbors of nodes such that they fulfill the following roles:

- *Observer node*: given a set of nodes U_0, we can add a new common neighbor v_o to these nodes such that the behavior of v_o depends on the nodes in U_0, but the behavior of U_0 is unaffected by the addition of v_o
- *Enabler node*: given a node u_1 dominated by another node u_d, we can add a new neighbor v_e to u_1, such that u_1 is no longer dominated by $\{u_d\}$, but it is dominated by the subset $\{u_d, v_e\}$

Given a set of input nodes U_0 and an output node u_1, we can use these concepts to build an AND gate which only enables the switching of u_1 if all nodes in U_0 are colored with a given color. This gadget connects to each of the input nodes in U_0 through a common observer node, and connects to the output u_1 through an enabler node. Besides the observer and enabler node, the gadget only requires an extra relay node (and an appropriate choice of weights) to connect these two nodes, and an extra upper neighbor for each node in order to connect the gadget to a pacer system which resets it after use. A brief illustration of the gadget is available in Figure 8. In a very similar fashion, we can also create AND gates for inputs of the other color, OR gates, or even multi-layer gates that allow us to combine different conditions.

Besides logical gates, the other key gadget in our benevolent construction is the state chain. For each level of the construction, we add a separate state chain in order to indicate the current state (i.e., point in the execution) of the procedure on this level. Essentially, a state chain is a vertical chain of nodes, where every node in the chain is dominated by its upper neighbor, similarly to the case of a storage chain. However, while storage chains are

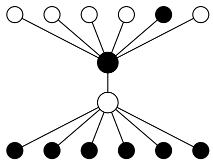

Figure 7 Adding a pair of pacers between a layer of control nodes and base nodes.

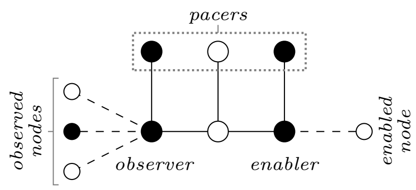

Figure 8 Logical (e.g. AND) gate.

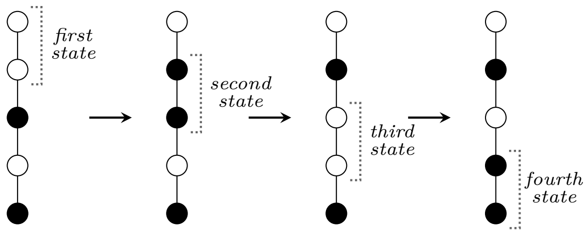

Figure 9 Simplified illustration of a state chain on 4 states (see the full version of the paper for a more detailed illustration). The position of the conflict in the chain shows our current state of the procedure. When propagating the conflict down by one step, the chain proceeds to the next state.

used to accumulate conflicts, a state chain will, on the other hand, always contain exactly one conflict, which we propagate down step by step. The different possible positions of the conflict can then correspond to different states of the procedure, and at a given point in time, our current state in the procedure is determined by the current position of the conflict in the chain (as illustrated in Figure 9).

One such state chain is added to each level of our benevolent construction. The node pairs in the chain that express a state are included in the conditions of the logical gates that control the execution of the recursive procedure on the level, ensuring that certain steps are only available to the benevolent player at certain points in the process. Furthermore, the nodes in the state chain are also connected to enabler nodes, and thus proceeding to the next state is always based on a given condition. Therefore, the benevolent player has no other option than to simultaneously proceed through the steps of the recursive process and the states of the state chain, in the appropriate order. With a couple of auxiliary nodes at the top of the chain, we can also connect the state chain to a pacer node, allowing us to reset the chain and jump back to the first state whenever the last state of the chain is reached.

Given these gadgets, let us now briefly reflect on the states and conditions we need to encode in order to ensure that the player has to follow the recursive sequence. The main idea is to use the logical gates to control the flow of execution within a given level: through the enabler nodes of the gates, we ensure that the switching of the next 2×4 control nodes (i.e., the next step of the control sequence) is only enabled after the previous switching of the base nodes is finished. In practice, this means that, after the base nodes have switched, when the newly added conflicts are propagated down far enough in each of the 2×6 storage chains below, the gates enable the further down-propagation of the appropriate 2×4 conflicts in the storage chains above, which will in turn make the next subset of 2×4 control nodes switchable. That is, the input (observer) nodes of these logical gates are connected to specific nodes of the storage chains below the level, while their output (enabler) nodes are connected to nodes of the storage chains above the level.

However, recall that charging the storage chains below takes 4 steps, while executing the control sequence above consist of 6 steps, so the two processes do not remain in synchrony. Thus in different phases of the procedure, the same set of storage chain nodes below have to enable different subsets of the control nodes above. Because of this, our construction encodes these different phases of the procedure as states in the state chain, and the appropriate state is also included in the condition of the logical gate that enables the next set of control nodes. When a cycle is finished (i.e., the two processes return to their default state at the same time), the state chain is reset and iteration starts again from the first state of the chain.

Furthermore, note that throughout the recursion, execution repeatedly leaves the current level and continues on the level above (or below), so the state chain of each level also has specific states indicating that the execution is currently on a level above (or below).

Altogether, these benevolent-case modifications only add constantly many gadgets (each of constant-size) to each level of the construction. Therefore, the modified construction still has only $O(1)$ nodes in a level, allowing for $\Theta(n)$ levels and thus $2^{\Theta(n)}$ stabilization time. This establishes our lower bound for model SB. By design, the construction only has a few (at most constantly many) switchable nodes at every point in time, and thus even in model CB, it allows for only very limited concurrency for the benevolent player. Specifically, since there are concrete nodes in the construction that switch $2^{\Theta(n)}$ times, the number of steps is still exponential in model CB.

Also, note that even with the gadgets added in the benevolent case, each node of the graph still has a constant degree, and thus our bound is also valid for sparse graphs.

References

1. Ron Aharoni, Eric C Milner, and Karel Prikry. Unfriendly partitions of a graph. *Journal of Combinatorial Theory, Series B*, 50(1):1–10, 1990.
2. Cristina Bazgan, Zsolt Tuza, and Daniel Vanderpooten. On the existence and determination of satisfactory partitions in a graph. In *International Symposium on Algorithms and Computation*, pages 444–453. Springer, 2003.
3. Cristina Bazgan, Zsolt Tuza, and Daniel Vanderpooten. Complexity and approximation of satisfactory partition problems. In *International Computing and Combinatorics Conference*, pages 829–838. Springer, 2005.
4. Cristina Bazgan, Zsolt Tuza, and Daniel Vanderpooten. The satisfactory partition problem. *Discrete applied mathematics*, 154(8):1236–1245, 2006.
5. Cristina Bazgan, Zsolt Tuza, and Daniel Vanderpooten. Satisfactory graph partition, variants, and generalizations. *European Journal of Operational Research*, 206(2):271–280, 2010.
6. Olivier Bodini, Thomas Fernique, and Damien Regnault. Quasicrystallization by stochastic flips. *HAL online archives*, 2009.
7. Olivier Bodini, Thomas Fernique, and Damien Regnault. Stochastic flips on two-letter words. In *2010 Proceedings of the Seventh Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 48–55. SIAM, 2010.
8. Henning Bruhn, Reinhard Diestel, Agelos Georgakopoulos, and Philipp Sprüssel. Every rayless graph has an unfriendly partition. *Combinatorica*, 30(5):521–532, 2010.
9. Zhigang Cao and Xiaoguang Yang. The fashion game: Network extension of matching pennies. *Theoretical Computer Science*, 540:169–181, 2014.
10. Jacques Demongeot, Julio Aracena, Florence Thuderoz, Thierry-Pascal Baum, and Olivier Cohen. Genetic regulation networks: circuits, regulons and attractors. *Comptes Rendus Biologies*, 326(2):171–188, 2003.
11. Silvio Frischknecht, Barbara Keller, and Roger Wattenhofer. Convergence in (social) influence networks. In *International Symposium on Distributed Computing*, pages 433–446. Springer, 2013.

12 Michael U Gerber and Daniel Kobler. Algorithmic approach to the satisfactory graph partitioning problem. *European Journal of Operational Research*, 125(2):283–291, 2000.

13 Michael U Gerber and Daniel Kobler. Classes of graphs that can be partitioned to satisfy all their vertices. *Australasian Journal of Combinatorics*, 29:201–214, 2004.

14 Sandra M Hedetniemi, Stephen T Hedetniemi, KE Kennedy, and Alice A Mcrae. Self-stabilizing algorithms for unfriendly partitions into two disjoint dominating sets. *Parallel Processing Letters*, 23(01):1350001, 2013.

15 Barbara Keller, David Peleg, and Roger Wattenhofer. How Even Tiny Influence Can Have a Big Impact! In *International Conference on Fun with Algorithms*, pages 252–263. Springer, 2014.

16 Jeremy Kun, Brian Powers, and Lev Reyzin. Anti-coordination games and stable graph colorings. In *International Symposium on Algorithmic Game Theory*, pages 122–133. Springer, 2013.

17 Damien Regnault, Nicolas Schabanel, and Éric Thierry. Progresses in the Analysis of Stochastic 2D Cellular Automata: A Study of Asynchronous 2D Minority. In Luděk Kučera and Antonín Kučera, editors, *Mathematical Foundations of Computer Science 2007*, pages 320–332. Springer Berlin Heidelberg, 2007.

18 Damien Regnault, Nicolas Schabanel, and Éric Thierry. On the analysis of “simple” 2d stochastic cellular automata. In *International Conference on Language and Automata Theory and Applications*, pages 452–463. Springer, 2008.

19 Jean-Baptiste Rouquier, Damien Regnault, and Éric Thierry. Stochastic minority on graphs. *Theoretical Computer Science*, 412(30):3947–3963, 2011.

20 Khurram H Shafique and Ronald D Dutton. On satisfactory partitioning of graphs. *Congressus Numerantium*, pages 183–194, 2002.

21 Saharon Shelah and Eric C Milner. Graphs with no unfriendly partitions. *A tribute to Paul Erdös*, pages 373–384, 1990.