Sublinear-Time Cellular Automata

and Connections to Complexity Theory

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Augusto Casagrande Viapiana Modanese

Tag der mündlichen Prüfung: 11. November 2022

Erster Gutachter: Prof. Dr. Jörn Müller-Quade,
Karlsruher Institut für Technologie (KIT)
Zweiter Gutachter: Prof. Dr. Martin Kutrib,
Justus-Liebig-Universität Gießen

Abstract

Distributed computing studies models in which multiple computational units coordinate themselves to work towards some common goal while having only limited resources at their disposal—be it time, space, or communication. The main object of study of this dissertation is the arguably simplest such model there is: (one-dimensional) cellular automata. Our goal is to obtain a better picture of the capabilities and limitations of the model and its variants in the case where the overall processing time is significantly smaller than the size of the input (i.e., *sublinear time*). We carry out our analysis from the perspective of computational complexity theory and also establish connections between cellular automata and other fields such as distributed computing and streaming algorithms.

Sublinear-time cellular automata. A *cellular automaton* (CA) is composed of identical cells arranged on a line. Each cell is essentially a very primitive computational unit (namely a deterministic finite automaton) that may interact with its two neighbors. Computation occurs by the cells updating their state according to the same local transition function all across the automaton. The variants we consider include *shrinking CAs*, which are CAs that are dynamically reconfigurable (to some extent) by means of cell deletion, and also a probabilistic variant in which each cell is equipped with access to a fair coin.

Despite remarkable interest on CAs in the linear- and real-time cases, the setting of sublinear time appears to have been neglected by the community at large. We review the little previous work on the topic there is and further develop some techniques originating from it so as to widen the scope of applications. This effort yields, among others, a time hierarchy theorem for the deterministic model. We also transfer lower bound techniques from complexity theory over to the shrinking CA model as well as develop new ones to analyze the capabilities of probabilistic sublinear-time CAs.

A connection to hardness magnification. One connection to complexity theory we establish along the way is a *hardness magnification* theorem for shrinking CAs. Here hardness magnification refers to a recent line of work that shows how even slightly nontrivial lower bounds might have very surprising consequences for complexity theory. Ours is a recasting of a recent result of McKay, Murray, and Williams (STOC, 2019) for streaming algorithms. As we also show, the result may be equally stated in terms of shrinking CAs, which provably strengthens it.

A connection to sliding-window algorithms. We relate the *distributed* model of cellular automata to the *sequential* model of streaming algorithms. As we show, (certain variants of) CAs can be simulated by streaming algorithms that are subject to certain locality restrictions. Concretely, the current state of the algorithm is exclusively dependent on a (fixed-size) window that contains the last few symbols that have been processed. We dub this restricted form of streaming algorithm a *sliding-window algorithm*, accordingly. We prove that sliding-window algorithms can simulate CAs quite efficiently and, in particular, in such a way that their space complexity is tightly connected to the time complexity of the simulated CA.

Derandomization results. We prove derandomization results for the model of sliding-window algorithms that draw randomness from a binary source. To do so, we rely on the robust machinery of branching programs that forms the standard approach for derandomization of space-bounded computation in complexity theory. As an application, using the aforementioned connection to cellular automata, we derive derandomization results for sublinear-time probabilistic CAs.

Predicting fungal sandpiles. A final problem we tackle and that relates to sublinear-time complexity in relation to cellular automata (albeit not sublinear-time cellular automata *per se*) is the *prediction problem* in sandpile automata. These automata are defined based on two-dimensional CAs and model a deterministic process in which particles (usually thought of as grains of sand) spread across space. The prediction problem asks whether, given the number y of a cell and some initial configuration for the sandpile, the cell with the number y will eventually assume a non-zero state in a set amount of time.

The complexity of the prediction problem (for two-dimensional sandpiles) has been remarkably open for at least two decades. We effectively settle the question for a variant of sandpiles called *fungal sandpiles* recently proposed by Goles et al. (Phys. Lett. A, 2020). Our result is of particular relevance since it provides fresh insights and new techniques that could be highly relevant towards obtaining a solution for the open problem in the general case.

Zusammenfassung

Im Gebiet des verteilten Rechnens werden Modelle untersucht, in denen sich mehrere Berechnungseinheiten koordinieren, um zusammen ein gemeinsames Ziel zu erreichen, wobei sie aber nur über begrenzte Ressourcen verfügen — sei diese Zeit-, Platz- oder Kommunikationskapazitäten. Das Hauptuntersuchungsobjekt dieser Dissertation ist das wohl einfachste solche Modell überhaupt: (eindimensionale) Zellularautomaten. Unser Ziel ist es, einen besseren Überblick über die Fähigkeiten und Einschränkungen des Modells und ihrer Varianten zu erlangen in dem Fall, dass die gesamte Bearbeitungszeit deutlich kleiner als die Größe der Eingabe ist (d. h. *Sublinear-Zeit*). Wir führen unsere Analyse von dem Standpunkt der Komplexitätstheorie und stellen dabei auch Bezüge zwischen Zellularautomaten und anderen Gebieten wie verteiltes Rechnen und Streaming-Algorithmen her.

Sublinear-Zeit Zellularautomaten. Ein *Zellularautomat* (ZA) besteht aus identischen Zellen, die entlang einer Linie aneinandergereiht sind. Jede Zelle ist im Wesentlichen eine sehr primitive Berechnungseinheit (nämlich ein deterministischer endlicher Automat), die mit deren beiden Nachbarn interagieren kann. Die Berechnung entsteht durch die Aktualisierung der Zustände der Zellen gemäß derselben Zustandsüberführungsfunktion, die gleichzeitig überall im Automaten angewendet wird. Die von uns betrachteten Varianten sind unter anderem *schrumpfende ZAs*, die (gewissermaßen) dynamisch rekonfigurierbar sind, sowie eine probabilistische Variante, in der jede Zelle mit Zugriff auf eine faire Münze ausgestattet ist.

Trotz überragendem Interesse an Linear- und Real-Zeit-ZAs scheint der Fall von Sublinear-Zeit im Großen und Ganzen von der wissenschaftlichen Gemeinschaft vernachlässigt worden zu sein. Wir arbeiten die überschaubare Anzahl an Vorarbeiten zu dem Thema auf, die vorhanden ist, und entwickeln die daraus stammenden Techniken weiter, sodass deren Spektrum an Anwendungsmöglichkeiten wesentlich breiter wird. Durch diese Bemühungen entsteht unter anderem ein Zeithierarchiesatz für das deterministische Modell. Außerdem übertragen wir Techniken zum Beweis unterer Schranken aus der Komplexitätstheorie auf das Modell der schrumpfenden ZAs und entwickeln neue Techniken, die auf probabilistische Sublinear-Zeit-ZAs zugeschnitten sind.

Ein Bezug zu Härte-Magnifizierung. Ein Bezug zu Komplexitätstheorie, die wir im Laufe unserer Untersuchungen herstellen, ist ein Satz über *Härte-Magnifizierung* (engl. *hardness magnification*) für schrumpfende ZAs. Hier bezieht sich Härte-Magnifizierung auf eine

Reihe neuerer Arbeiten, die bezeugen, dass selbst geringfügig nicht-triviale untere Schranken sehr beeindruckende Konsequenzen in der Komplexitätstheorie haben können. Unser Satz ist eine Abwandlung eines neuen Ergebnisses von McKay, Murray und Williams (STOC, 2019) für Streaming-Algorithmen. Wie wir zeigen kann die Aussage dabei genauso in Bezug auf schrumpfende ZAs formuliert werden, was sie auch beweisbar verstärkt.

Eine Verbindung zu Sliding-Window Algorithmen. Wir verknüpfen das *verteilte* Zellularautomatenmodell mit dem *sequenziellen* Streaming-Algorithmen-Modell. Wie wir zeigen, können (gewisse Varianten von) ZAs von Streaming-Algorithmen simuliert werden, die bestimmten Lokalitätseinschränkungen unterliegen. Konkret ist der aktuelle Zustand des Algorithmus vollkommen bestimmt durch den Inhalt eines Fensters fester Größe, das wenige letzte Symbole enthält, die vom Algorithmus verarbeitet worden sind. Dementsprechend nennen wir diese eingeschränkte Form eines Streaming-Algorithmus einen *Sliding-Window*-Algorithmus. Wir zeigen, dass Sliding-Window-Algorithmen ZAs sehr effizient simulieren können und insbesondere in einer solchen Art und Weise, dass deren Platzkomplexität eng mit der Zeitkomplexität des simulierten ZA verbunden ist.

Derandomisierungsergebnisse. Wir zeigen Derandomisierungsergebnisse für das Modell von Sliding-Window-Algorithmen, die Zufall aus einer binären Zufallsquelle beziehen. Dazu stützen wir uns auf die robuste Maschinerie von Branching-Programmen, die den gängigen Ansatz zur Derandomisierung von Platz-beschränkten Maschinen in der Komplexitätstheorie darstellen. Als eine Anwendung stellen sich Derandomisierungsergebnisse für probabilistische Sublinear-Zeit-ZAs heraus, die durch die oben genannten Verknüpfung erlangt werden.

Vorhersageproblem für Pilz-Sandhaufen. Ein letztes Problem, das wir behandeln und das auch einen Bezug zu Sublinear-Zeitkomplexität im Rahmen von Zellularautomaten hat (obwohl nicht zu Sublinear-Zeit-Zellularautomaten selber), ist das *Vorhersageproblem* für Sandhaufen-Zellularautomaten. Diese Automaten sind basierend auf zweidimensionalen ZAs definiert und modellieren einen deterministischen Prozess, in dem sich Partikel (in der Regel denkt man an Sandkörnern) durch den Raum verbreiten. Das Vorhersageproblem fragt ob, gegeben eine Zellennummer y und eine initiale Konfiguration für den Sandhaufen, die Zelle mit Nummer y irgendwann vor einer gewissen Zeitschranke einen von Null verschiedenen Zustand erreichen wird.

Die Komplexität dieses mindestens zwei Jahrzehnte alten Vorhersageproblems ist für zweidimensionelle Sandhaufen bemerkenswerterweise nach wie vor offen. Wir lösen diese Frage im Wesentlichen für eine neue Variante von Sandhaufen namens *Pilz-Sandhaufen*, die von Goles u. a. (Phys. Lett. A, 2020) vorgeschlagen worden ist. Unser Ergebnis ist besonders relevant, weil es innovative Erkenntnisse und neue Techniken liefert, die für die Lösung des offenen Problems im allgemeinen Fall von hoher Relevanz sein könnten.

Acknowledgments

This dissertation crowns my 11 years of study in Karlsruhe. Given the long time period, the reader is asked to excuse the fact that some of these acknowledgments extend well beyond what concerns this work alone.

First and foremost, I thank Dr. Thomas Worsch for introducing me to the topic of cellular automata, for his mentorship during a great part of my studies in Karlsruhe, for many, many lively conversations about research, career advice, and several other miscellaneous topics, for being in essence one of the best teachers I have ever met, and (of course!) for a very nice and fashionable cap.

Secondly, I would like to thank Prof. Dr. Jörn Müller-Quade for serving as the first reviewer to this dissertation as well as for the occasional discussion about my research and for the suberb lectures during my bachelor's and master's. Thanks are also due to Prof. Dr. Martin Kutrib of Gießen for accepting the role of second reviewer and for writing many works that directly or indirectly influenced and inspired quite a few topics of this dissertation and the two theses that preceded it.

Next I should mention Prof. Dr. Ralf Reussner and Jun.-Prof. Dr. Franziska Mathis-Ullrich for contributing with quite insightful conversations prior to the submission of this dissertation. I am particularly thankful to Prof. Reussner for very valuable tips concerning the defense (which inspired not only the structure of the presentation but also of Chapter 1 of this dissertation) as well as for accepting the role of examiner. I also thank Prof. Dr. Michael Beigl and Prof. Dr. Jörg Henkel for their roles in the examination board of my defense.

A next round of thanks goes to all my colleagues at KASTEL, even if we did not have all that many day-to-day dealings in common except for the occasional greeting while walking down the aisle to the printer room. My warmest thanks to Frau Manietta for her great and infallible help when dealing with all matters regarding paperwork, which was called upon several times indeed.

I would also like to thank Dr. Ján Pich and Prof. Rahul Santhanam as well as all the others I met at Oxford for their hospitality while I tended to the final details of this dissertation. Yet another round of thanks goes to Prof. Yuichi Yoshida, Hiraki-san, and the others at the NII for all their help in making my stay in Tokyo an absolutely thrilling experience. I look forward to seeing you all again soon.

On a more personal note, I thank Nagatou-sensei for all her help, advice, and mentorship during virtually the entirety of my studies in Karlsruhe. I also thank all my German teachers—Adilsom, Avelino, Christoph, and Stefan—for their invaluable help in making

a life of study in Germany possible. Special mention goes to Adilsom, of course, for suggesting and encouraging me to follow this path as well as to Christoph for assisting me in applying for the B.Sc. back in 2011.

Last but not least, I would like to say thank you to my parents for all their support and care during my studies as well as to a special someone for cheering every success of mine, as small as it may be, and for reassuring me in the times I needed the most. I love you all so much.

Coimbra, November 22nd, 2022

Contents

Part I.

Introduction

1. Introduction

1.1. Background and Motivation

The relevance of distributed—and especially *local*—computational models in the modern era cannot be understated. Oftentimes a large chunk of data is spread across a network of several computers—potentially all across the globe—that do not have the resources to communicate with one another, be it due to constraints in time, space, or even in the number of communication lines. This is in line with the contemporary paradigm shift of data being so large and complex that we cannot hope to process all of it in a single machine.

These trends motivate the study of models of distributed computation that must operate very efficiently but nevertheless in a local fashion. In theoretical computer science, a natural translation for such efficiency may be found in *sublinear-time* models, which (as the name already suggests) must operate in fewer steps than the size of their input. As for locality, this is usually modeled by augmenting the computational model at hand so as to render it aware of the state of its immediate neighbors and capable of interacting with them (e.g., by exchanging messages).

Arguably the simplest model capturing systems as described above are *cellular automata*, in particular their one-dimensional variant. A cellular automaton is composed of a certain number of identical computational units—its *cells*—that are arranged according to a path topology. Each cell is a deterministic finite automaton (DFA), essentially the most basic model of computation there is. The communication lines are laid in-between immediate neighbors according to their positions in the underlying one-dimensional array. Evidently, cellular automata are a model that is compatible with the notions of sublinear-time local computing previously described—but one that has been stripped down to the bare minimum amount of resources.

Although there is great motivation to investigate sublinear-time computation by (one-dimensional) cellular automata and the model is by all means a novel one (as it is but less than a couple decades "younger" than Turing machines), the topic has been seemingly neglected by the community at large. (We review the relevant literature in Section 1.2.1.) This dissertation is an attempt at narrowing this knowledge gap. Although our primary focus is the study of cellular automata, our results are at the intersection of multiple fields, bringing together the worlds of computational complexity theory, distributed computing, sublinear-time algorithms, and streaming algorithms.

The next section (Section 1.2) introduces the models studied in this dissertation. Following that, in Section 1.3 we present our results and in Section 1.4 we discuss a selection of overarching themes and techniques that are central in obtaining them. At the end (Section 1.5) we also give a summary of the contents of Part II for the reader's convenience. To avoid repetition, we refrain from giving a "monolithic" section on related work; rather we mention and discuss it directly next to the relevant passages of the text.

1.2. Models Studied

This dissertation is concerned with the study of several models of computation. This section gives a broad overview of all models considered. Most of these are based on cellular automata and are covered in Section 1.2.1. However, another prominent model featured are *streaming algorithms* and, in particular, their subclass of *sliding-window algorithms*, which are addressed in Section 1.2.2.

1.2.1. Cellular Automata and their Variants

The classical model of (one-dimensional) cellular automata (CAs) is a very basic model of distributed computation. The units of the system are deterministic finite automata (DFAs) and are named *cells*. Cells are connected to other cells in their neighborhood according to a path topology. All cells are identical, though in *bounded CA* a distinct behavior for the bordering cells (i.e., the start and end nodes in the underlying path graph) is possible in practice. Unlike most prominent models in distributed computing (e.g., the celebrated LOCAL model of Linial [72]), cells are oblivious to their identity and their position in the automaton.[1] It is possible to circumvent this limitation by encoding this information in the machine's input, which is indeed an approach we explore in Chapters 2 and 3.

The problem of language recognition has been extensively studied in cellular automata [69, 111]. With the notable exception of [60, 66, 105], however, these efforts have been almost exclusively focused on the linear- or real-time case (to the detriment of the sublinear- or constant-time one).

At this junction one might object by arguing that sublinear-time computations have been studied in the context of picture languages (see, e.g., [28]) since the cellular automata there operate in time that is sublinear in the number of input symbols. However, our perspective is that these do not qualify as a "true" sublinear-time model since there there is enough time for any cell to propagate a message to the entire automaton (something which is crucially not possible in the cellular automata models we consider). In addition, the term

[1] In this regard cellular automata are very similar to the so-called port-numbering model [57]. The difference is that, in cellular automata, the underlying communication graph is always a path and the computational units are equipped with a sense of direction (whereas in the port-numbering model the ports of each unit may be labeled arbitrarily).

"sublinear" evaporates if we count time in function of the side length of the input's support. Indeed, the literature seems to agree with us in referring to these automata as linear- or real-time automata [9, 54–56, 96]. An exception might be the characterization of the picture language class REC^d, which bases on constant-time nondeterministic cellular automata [56]. Nonetheless, this is but a simple generalization of a result from the aforementioned work by Sommerhalder and Westrhenen [105].

ACAs. In Chapter 2, we investigate our first model of CAs as language acceptors. The model considered is that of *ACA*, which is a CA with the acceptance condition of simultaneous unanimity; that is, the automaton accepts if and only if all cells are simultaneously accepting. As discussed in Chapter 2, this kind of acceptance condition is necessary for non-trivial sublinear-time computation.

The ACA model is by all means novel; Rosenfeld already mentions it in his book from 1979, for instance [98]. In the setting of sublinear-time computation, ACAs have been previously studied in [60, 66, 105]. (The works [66, 105] are in fact restricted to the constant-time case.) Besides the ACA acceptance condition of unanimity being very natural, it incidentally coincides with the usual one used for similar problems in the sister field of distributed computing.[2]

In Chapter 2 we also propose a decider version of ACA that we call *DACA*. The acceptance condition of DACA is the same as that of ACA, whereas the condition for rejection is the analogue of the acceptance one: The automaton rejects if and only if all cells are simultaneously rejecting. To the best of our knowledge, ACA and DACA are the first example in CA literature for (natural) models in which the acceptor and decider variants differ in their capabilities.

Shrinking CAs. In Chapter 3 our attention is turned to another CA-based model capable of sublinear-time computation. A *shrinking CA* (SCA) is a CA variant in which cells may spontaneously delete themselves. This is modeled by a distinguished state that may be assumed as a function of the local configuration. In the underlying path topology, the corresponding operation is node deletions. In order to maintain connectedness, at each step we reconfigure the remaining nodes so as to connect them with the nearest neighbors left standing.

The SCA model was originally proposed by Rosenfeld and Wu [99]. For a recent study involving SCA as language acceptors, we refer to the paper of Kutrib, Malcher, and Wendlandt [71]. A generalization of SCAs was also considered by the present author in his bachelor's thesis [79]. (See also the corresponding conference paper [85].)

[2] For concrete examples, we refer the reader to the topics of distributed decision (see [35] for an excellent survey) or distributed proof systems (see, e.g., [36, 40, 68, 87, 88] for recent work in the area).

Probabilistic ACAs. As most deterministic machine models, it is natural to consider a probabilistic version of the ACA model. In Chapter 4 we analyze such a model which we accordingly refer to as *probabilistic ACA* (PACA). The model is highly inspired on the definition of probabilistic Turing machines (see, e.g., [5]) and also resembles a model that was previously studied by Arrighi, Schabanel, and Theyssier [7].

Unlike its deterministic counterpart, a PACA possesses *two* local transition functions. Each cell is given access to a fair coin, which it tosses at each time step. The outcome of this coin toss then determines which local transition function is applied by the cell. Hence it is admissible to view PACAs as a form of non-uniform CA model (in the classical sense of non-uniformity), but where the non-uniformity is dynamic (i.e., it may change from one step to the other) and is controlled by an external randomness source.

An alternative natural form of obtaining a probabilistic ACA model would be lifting the cells from DFAs to probabilistic finite machines (e.g., based on [97]). One reason we opt for not doing so is that models with an explicit binary randomness source are more amenable to a complexity-theoretical analysis. (Indeed, this choice paves the way for the derandomization results of Chapter 5.) Another reason for our choice is that an explicit binary random input is a more accurate representation of randomness sources in modern computers (rather than the aforementioned version based on probabilistic finite machines, which are allowed to follow arbitrary distributions).

As usual for probabilistic models, one may consider both one- and two-sided error versions of the model (in the same spirit as the classical complexity classes RP and BPP, respectively).

Sandpile automata. Chapter 6 covers a relatively distinct model in comparison to the previous ones. Namely we study a question concerning *sandpile automata*, which are cellular automata roughly modeling the behavior of piles of grains of sand in space. The cells of the automaton have states that count the number of grain of cells present in them; if the number exceeds a certain threshold, then the pile *collapses* and the excess sand is transferred to its neighbors. More specifically, in Chapter 6 we consider the recently proposed model of *fungal sandpile automata*, which is a variant of the model in which sand transfer is only possible along the horizontal or vertical axis, alternatingly.

The model of sandpile automata is due to Bak, Tang, and Wiesenfeld [10] and has been intensively studied from various perspectives. For the literature on sandpile automata in relation to the problems we study, we refer to the survey by Formenti and Perrot [39]. Fungal sandpile automata were recently introduced by Goles et al. [53]. The model is related to the (one-dimensional) model of *fungal automata* due to the same authors [1]. As the name suggests, these models are partially motivated by their likeliness in behavior to the biological processes inherent to certain species of fungi. (Again, we refer the reader to the previously cited works [1, 53] for a detailed discussion.) Regardless of these motivations, the model is interesting to consider as a stepping stone towards settling open questions regarding the predictability of (standard) two-dimensional sandpile automata.

1.2.2. Streaming Algorithms

In addition to the aforementioned models of cellular automata, we will also address *streaming algorithms*. These are sequential algorithms that operate on an input stream which is read one symbol at a time and in order. Streaming algorithms are usually subject to space limitations (using much less space than the length of their input stream). One can also consider streaming algorithms that perform multiple passes over their input; in this work we focus on single-pass algorithms.

Streaming algorithms have been the subject of intense study in the past few years, in particular due to their applicability to the processing of massive streams of data. Giving an overview of the field is beyond the scope of this dissertation. For an introduction to the particular case of graph algorithms, one might consult [74].

Sliding-window algorithms. A particular restriction of streaming algorithms that we consider in more detail are *sliding-window algorithms*. In the literature, the term usually refers to a model in which a contiguous *window* of fixed size t is passed over the data stream. The current operation or output of the algorithm depends only on the contents of its window; any preceding symbols are perceived as "outdated" and discarded. Usually the goal is to solve the task at hand while using an optimal amount of space, ideally much smaller than the window size t.

Sliding-window algorithms model computational processes where the data stream is perceived as "very large" (as in streaming algorithms), but only the more recent data is considered relevant or of use. The model as presented above is due to Datar et al. [29]. See also the survey by Braverman [16] for the main line of work in the area.

The kind of sliding-window algorithms that we shall consider in Chapter 5 moderately differs from the standard model just described. In particular, we shall consider a probabilistic variant of the model that is equipped with access to a binary randomness source. The window is then passed *not over the algorithm's data stream but over its randomness source*. sliding-window algorithms. This gives a fundamentally different model than other work on probabilistic (See Chapter 5 for a more in-depth discussion.)

Regarding the relevance of sliding-window algorithms to the main object of study of this dissertation (i.e., sublinear-time cellular automata), it certainly suffices to mention the very recent paper by Pacut et al. [94]. In their work (which, to the best of our knowledge, is the first to do so), Pacut et al. point out a connection between sliding-window algorithms (in [94] called "time-local" algorithms) and one-dimensional local distributed models such as cellular automata. We exploit this connection to obtain derandomization results as discussed in the next section.

1.3. Main Contributions

This section provides a broad overview of the main results presented in Part II. We also briefly discuss the consequences of each result and how it contributes to the preexisting state of the art. Most results are presented only informally or in a condensed manner. For greater details and a more formal treatment, we refer the reader to the respective chapters in Part II.

1.3.1. Sublinear-Time Cellular Automata

The results on sublinear-time cellular automata form the core contribution of this work. We present the results for the deterministic and probabilistic variants separately.

In the following, n always stands for the input length.

1.3.1.1. Deterministic Automata

ACAs. Let $\mathrm{ACA}[t]$ denote the class of languages that can be accepted by an ACA in at most t steps. As mentioned previously, to the best of the present author's knowledge there are only a few works that predate this dissertation in investigating the landscape of the sublinear-time ACA classes. Sommerhalder and Westrhenen [105] proved a characterization of $\mathrm{ACA}[1]$ based on subregular languages. (Kim and McCloskey [66] also proved a similar characterization but for ACAs with a different acceptance condition.) Meanwhile, Ibarra, Palis, and Kim [60] showed that every language in $\mathrm{ACA}[o(\log n)]$ is regular and that there is a non-regular language in $\mathrm{ACA}[\log n]$. Nothing was known for the classes between $O(\log n)$ and $O(n)$ time other than trivial inclusion relations. (Note $\mathrm{ACA}[n]$ coincides with the class of languages that can be accepted by cellular automata in $O(n)$ time *under the standard acceptance condition.*)

The results in Chapter 2 provide a much finer picture of the sublinear-time ACA classes. A first result in this sense is a *time hierarchy* for the classes in-between $\mathrm{ACA}[\log n]$ and $\mathrm{ACA}[n]$:

Theorem 2.13 (informal)**.** *Let t be a "nice" function that is asymptotically in-between $\Omega(\log n)$ and $O(n)$. Then, for any $t' = o(t)$, $\mathrm{ACA}[t'] \subsetneq \mathrm{ACA}[t]$ holds.*

"Nice" functions include, for instance, $t(n) = (\log n)^a$ and $t(n) = n^{1/a}$ for any reasonable (e.g., rational) choice of $a > 1$.

As a result of Theorem 2.13, starting from $\mathrm{ACA}[\log n]$ (if the distance between levels is "large enough") we get an ever-increasing hierarchy of language classes all the way up to the linear-time class $\mathrm{ACA}[n]$. A question that arises is, of course, what kind of new languages are obtained by doing so. (The proof of Theorem 2.13 already gives concrete examples, but we are asking this from a holistic perspective.) In particular, it is natural to

ask whether any of these languages are regular. It turns out that, if any are indeed regular, then we can narrow them down to a very specific superclass of $\mathsf{ACA}[1] = \mathsf{SLT}_\cup$:

Theorem 2.15. *Let* $\mathsf{LT} \subsetneq \mathsf{REG}$ *be the Boolean closure of the constant-time class* $\mathsf{SLT}_\cup$*, and let* REG *denote the regular languages. Then* $\mathsf{ACA}[o(n)] \cap \mathsf{REG} \subsetneq \mathsf{LT}$.

(Note we actually even show the inclusion is strict.) This leaves only a narrow "gap" for the existence of a language $L \in \mathsf{REG}$ that is decidable by an ACA in sublinear but not constant time (if any does indeed exist).

In the work of Ibarra, Palis, and Kim [60] it was also shown that $\mathsf{ACA}[o(\log n)] \subseteq \mathsf{REG}$. Together with their example for a non-regular language in $\mathsf{ACA}[\log n]$, this suggested that there was some fundamental distinction between the classes below and above logarithmic time. We settle this by proving that $\mathsf{ACA}[o(\log n)]$ fully collapses to constant time:

Theorem 2.16. $\mathsf{ACA}[o(\log n)] = \mathsf{ACA}[1]$.

Other results on the sublinear-time ACA classes of note are strict inclusions for $\mathsf{ACA}[o(n)]$ in the parallel computation classes SC and AC. In summary, this all gives a much more fine-grained perspective of the sublinear-time ACA hierarchy.

Yet another topic covered in Chapter 2 are *decider ACAs* (DACA), which we propose as a decider counterpart to ACA (i.e., as machines that must not only recognize words in their target language but also timely reject words that are not). For DACA we prove results similar to the ones above. Let $\mathsf{DACA}[t]$ denote the class of languages decidable by a DACA in time at most $O(t)$. The first main result on DACAs is the following characterization:

Theorem 2.25. $\mathsf{DACA}[1] = \mathsf{LT}$.

Recall that LT is defined as the Boolean closure of $\mathsf{ACA}[1] = \mathsf{SLT}_\cup$; this means that, when it comes to constant-time complexity, DACAs are (rather counterintuitively) *strictly more powerful* than ACAs (despite apparently having to function under stronger constraints). Our second result is the following, which is in the same spirit as the aforementioned result of Ibarra, Palis, and Kim [60]:

Theorem 2.26. $\mathsf{DACA}[o(\sqrt{n})] = \mathsf{DACA}[1]$.

The reader should compare this with Theorem 2.16.

Shrinking CAs. Our results on SCAs are at the intersection of the fields of cellular automata, complexity theory, and streaming algorithms. The main contributions are two-fold:

1. Strengthening previous work by McKay, Murray, and Williams [75] on streaming algorithms, we establish that showing *even slightly nontrivial* lower bounds for SCAs has surprising consequences for outstanding problems in complexity theory.
2. We *unconditionally prove* that SCAs are a weaker model than streaming algorithms (thus justifying the use of the term "strengthening" in the previous item).

We further elaborate on these two points.

Starting with the work of Oliveira and Santhanam [93], quite a few recent works in complexity theory have presented results of the following form: Say some very low (albeit non-trivial) lower bound holds for a computational model $\mathcal{M}$; then a very surprising separation of complexity classes (e.g., $\mathsf{P} \neq \mathsf{NP}$) follows. In the aforementioned work by Oliveira and Santhanam, the authors identified a few results of this type and also coined the term *hardness magnification* for them. As for subsequent works in this same vein, [20–24, 41, 92] may be named.

A problem that prominently features among the results on hardness magnification (and that has also received quite a deal of attention in other contexts; see, e.g., the recent survey by Allender [2]) is the *minimum circuit size problem* MCSP. Having a fixed a parameter $s(n)$, the question is, given the truth table of a Boolean function f, whether there is a Boolean circuit of size at most $s(n)$ that computes f. (We refer to Chapter 3 for the precise definition.) Recall that the *update time* of a streaming algorithm is the maximum amount of time it spends computing between reading one symbol and the next. In the aforementioned work by McKay, Murray, and Williams, the authors show the following hardness magnification result for MCSP in the context of streaming algorithms:

Theorem 3.2 ([75], informal)**.** *If there is no* $\mathsf{poly}(s(n))$*-space streaming algorithm with* $\mathsf{poly}(s(n))$ *update time for (the search version of)* MCSP*, then* $\mathsf{P} \neq \mathsf{NP}$*.*

We strengthen this result by proving that we can essentially replace "streaming algorithms" with "SCAs" in its statement. For a function t, let $\mathsf{SCA}[t]$ denote the class of languages that can be accepted by an SCA in $O(t)$ time.

Theorem 3.3 (informal)**.** *If (an adequately presented version of)* $\mathsf{MCSP} \notin \mathsf{SCA}[\mathsf{poly}(s(n))]$*, then* $\mathsf{P} \neq \mathsf{NP}$*.*

The need for an adequate presentation of MCSP to SCAs is due to limitations in the model that originate from the underlying model of cellular automata. In particular, sublinear-time SCAs are insensitive to the length of long unary substrings in their input, thus ruling out the existence of efficient SCAs for simple problems (e.g., parity) if the input word is presented as-is. Nevertheless, as we argue in Chapter 3, these limitations can be circumvented by presenting the input in a *block word* format, which is a locally verifiable encoding of the original input. In particular, (as alluded to in Section 1.2.1) this allows the

cells to be locally aware of the input length and their absolute position, which is otherwise not possible in general.

The second point above (i.e., showing that Theorem 3.3 is indeed a strengthening of Theorem 3.2) is proven by two results. In the first one, we show how a streaming algorithm can efficiently (in both time and space) simulate an SCA. Recall that the *reporting time* of a streaming algorithm is the time the algorithm spends computing between reading the last symbol in its stream and finally presenting its output.

Theorem 3.4 (informal)**.** *For every constructible function t and any $L \in \mathsf{SCA}[t(n)]$, there is an $O(t(n))$-space streaming algorithm for L with $O(t(n)\log t(n))$ update and $O(t(n)^2 \log t(n))$ reporting time.*

Conversely, there are languages which can be decided efficiently by a streaming algorithm (even if they are presented in a block word format) but which require nearly linear time on an SCA.

Theorem 3.5 (informal)**.** *There is a language L_1 for which (an adequately presented version of) $L_1 \notin \mathsf{SCA}[o(n/\log n)]$ can be accepted by an $O(\log n)$-space streaming algorithm with $\tilde{O}(\log n)$ update time.*

1.3.1.2. Probabilistic Automata

In Chapter 4 we show a number of results on sublinear-time PACAs. As we discuss further below, special attention is given to the constant-time case, which is already very rich and allows us to draw some interesting parallels to the deterministic case.

We first compare one- and two-sided error PACAs with respect to their computational power and effectively separate the two of them all the way up to $o(\sqrt{n})$ time. Two machines are said to be *equivalent* if they accept the same language.

Theorem 4.1. *The following hold:*

1. *If C is a one-sided error PACA with time complexity T, then there is an equivalent two-sided error PACA C' with time complexity $O(T)$.*
2. *There is a language L recognizable by constant-time two-sided error PACA but not by any $o(\sqrt{n})$-time one-sided error PACA.*

It is relevant to note the first item is not a direct consequence of the definitions; rather, we must first show how to implement error reduction by a constant factor, which requires a non-trivial construction.

We also show consequences of derandomizing PACAs with respect to time complexity for open problems in complexity theory. The results obtained are as the following:

Theorem 4.2 (excerpt)**.** *If there is $\varepsilon > 0$ such that every n^ε-time (one- or two-sided error) PACA can be converted into an equivalent $n^{O(1)}$-time deterministic CA, then* $\mathsf{P} = \mathsf{RP}$.

The main relevance we see in this result is in directing further efforts on derandomizing probabilistic cellular automata. Since the case of derandomization for RP is still wide open, this damps hopes for obtaining time-efficient derandomization of PACAs in the near future. Nevertheless, it does not rule out *space-efficient* derandomization, which is indeed a point we explore later in Chapter 5 (see also Section 1.3.2 below). This does not come as a surprise since along the years there has been much more significant progress towards derandomizing RL than RP.

The most prominent results obtained in Chapter 4 are on the constant-time case. For one-sided error PACAs, we prove a full derandomization.

Theorem 4.3. *For any constant-time one-sided error PACA C, there is an equivalent constant-time deterministic ACA.*

In turn, we significantly narrow down the set of languages that can be accepted by two-sided error PACAs in constant time. To this end we first identify a natural class LLT of subregular languages we call the *locally linearly testable* languages. The name is due to the central defining condition for these classes, which characterizes words based on whether the weighted sum of the number of infixes in a word exceeds some predefined (constant) threshold. Let $\mathsf{LLT}_{\cup\cap}$ denote the closure over LLT under union and intersection, and let LTT denote the class of *locally threshold testable* languages, which can be shown to coincide with the Boolean closure of LLT (i.e., its closure under union, intersection, and complement).

Theorem 4.4. *The class of languages that can be accepted by a constant-time two-sided error PACA contains* $\mathsf{LLT}_{\cup\cap}$ *and is strictly contained in* LTT.

1.3.2. Sliding-Window Algorithms

A *pseudorandom generator* (PRG) is a function $G\colon \{0,1\}^d \to \{0,1\}^n$ that "looks random" to a class of distinguishers. Ideally, $d \ll n$ and G is efficiently computable (in our context, using a linear amount of space). One central application of PRGs is in replacing the random input of a machine model $\mathcal{M}$, in which case we say G *fools* $\mathcal{M}$. Concretely, suppose that $\mathcal{M}$ receives a regular input x and a random input r uniformly chosen from $\{0,1\}^n$. Then the PRG G guarantees that, if we choose s uniformly at random from $\{0,1\}^d$, then the probability of $\mathcal{M}$ accepting on the same input x but using $r = G(s)$ as random input instead is approximately the same as that when we pick r uniformly at random from $\{0,1\}^n$.

Suppose now that an execution of $\mathcal{M}$ is efficiently computable (in the same sense as G is). Then we can obtain a *derandomization* result for $\mathcal{M}$ by looping over every possible seed of G and tallying up the results. This is because using G to generate random inputs for

$\mathcal{M}$ gives us an approximation for the probability of $\mathcal{M}$ accepting in the general case (i.e., where we choose the random input of $\mathcal{M}$ completely at random).

Our main contribution in Chapter 5 is the construction of a PRG specifically designed for fooling sliding-window algorithms. As an example, for sliding-window algorithms where the window size is polynomial in its space complexity s, our PRG has a seed length of $O(\log s)$, which is essentially optimal. We refer the reader to Chapter 5 for a formal presentation of the results.

In Chapter 5 we also construct PRGs that fool sliding-window satisfying an additional property we dub *δ-criticality*. In a nutshell, an algorithm A satisfies this property if, for every random input r, every single section of r either has a significant impact on the decision of A (by a probability gap of at least δ) or no impact at all (i.e., the section is non-critical). This investigation is mostly motivated by its application to the derandomization of PACAs, which we discuss next.

Application to PACAs. The greater relevance of the results from Chapter 5 for the broader scope of this dissertation are in that they yield space-efficient derandomizations of the PACA model. This is a direct consequence of the following result:

Theorem 5.25. *Let C be a (one- or two-sided error) PACA with state set Q and $T \in \mathbb{N}_0$. Then there is a $O(T \log|Q|)$-space randomized non-uniform sliding-window algorithm $S = S_T$ of window size $O(T^2)$ such that*

$$\Pr[S(x) = 1] = \Pr[C \text{ accepts } x \text{ in time step } T].$$

The derandomization obtained in the case of one-sided error PACAs is particularly surprising.

Theorem 5.9 (informal)**.** *For any T-time one-sided error PACA C, there is a deterministic algorithm for $L(C)$ with space complexity $O(T + (\log n)^2)$.*

For comparison, the state of the art for derandomizing space-bounded algorithms in general falls short of achieving even quasilinear space (whereas here we obtain *linear* space when $T = \Omega(\log n)^2$). We also remark that Theorem 5.9 can be extended to the case where the acceptance probability of C is inversely polynomial in T (and not simply constant). As always, we refer to Chapter 5 for a more in-depth discussion.

The direct application to two-sided error PACAs appears to be more limited; nevertheless, it does merit mention regarding the case of δ-critical PACAs. A PACA C is said to be *δ-critical* if, for any given input, if a cell of C is such that it does not always accept, then the probability that it does is at most $1 - \delta$.

Theorem 5.11 (informal)**.** *Let $\delta > 0$. For any T-time δ-critical two-sided error PACA C, there is a deterministic algorithm for $L(C)$ with space complexity $\tilde{O}(T \cdot (\log(1/\delta))^2) + O(\log n)$.*

The result of Theorem 5.11 can be shown to hold as well when the gap between the probabilities for C accepting or rejecting is only exponentially small in T (and not just constant).

1.3.3. Fungal Sandpile Automata

Obviously the model of sandpile automata (or its fungal automata variant) does not qualify as a sublinear-time model. Rather our object of study in Chapter 6 are *prediction problems* and—more specifically—whether these are solvable in sublinear time or not.

The standard prediction problem Π in these types of automata is the following: Given a cell index i, a number of time steps t, and a starting configuration c, determine whether the cell corresponding to i is in a non-zero state after t steps of the automaton. In standard sandpile automata, it is known that Π is solvable in sublinear time when the underlying sandpile automaton is one-dimensional [78], whereas it is P-complete (and hence unlikely to be solvable in sublinear time) if the automaton is three-dimensional [89]. In-between the two lies the two-dimensional case, which has been open for quite some time.

Goles et al. [53] recently showed that Π is P-complete in the case of two-dimensional fungal sandpile automata, but only for a restricted variant of the model. In particular, the result of Goles et al. only applies to fungal sandpile automata that operate following a specific sequence of horizontal and vertical transitions. In Chapter 6, we greatly improve on this result by showing P-completeness also in the case where horizontal and vertical transitions are strictly alternating, which is the most general case possible.

More than just being a generalization of the aforementioned result of Goles et al., we argue our result is a major contribution because of *the techniques developed in the proof*. The overall strategy we follow, of course, is the same as that of previous works, namely by showing how to realize arbitrary Boolean circuits in the underlying model. (This is sufficient because evaluating a circuit for a given set of inputs is a famous P-complete problem.) Of prime importance, however, is how this realization of circuits is carried out. We exploit the characteristics of the model in an ingenious way, completely avoiding barriers that were previously shown to hold for the two-dimensional case [43]. These new ideas could prove themselves to be of high relevance for attacking the open case of general two-dimensional sandpile automata. We refer the reader to Chapter 6 for details.

1.4. Common Themes and Techniques

In this section we discuss a few overarching themes and techniques that are central in obtaining the results presented in Sections 1.3.1 and 1.3.2. (We stress that this is not an attempt at giving detailed proof ideas for the results, which is rather the subject of the respective chapters.) Almost all of these are used or present in the context of results that belong to two or more chapters of this dissertation. The single exception is the shattering technique (Section 1.4.5), which only appears in Chapter 5 but merits being addressed

here since it appears to be a very useful tool for the further study of sliding-window algorithms.

1.4.1. Locality and Anonymity

As already mentioned in Section 1.2.1, the cells of a CA can only communicate with those in their vicinity (*locality*) and are also initially unaware of their position or identity in the automaton (*anonymity*).[3] These limitations already pose a certain challenge in the linear-time or real-time setting, but generally it is possible to overcome them by using a set of standard techniques. For instance, to cope with anonymity, one can use signals to pinpoint the location of a particular cell (e.g., to find the "middle" cell in the automaton) or also resort to more elaborate counter constructions (see, e.g., [106, 114]).

In the sublinear-time case the limitations turns out to be much more severe. Although the aforementioned methods remain valid for a variety of purposes, they fall flat in overcoming locality and anonymity. For instance, since in t steps a cell can only receive signals that were started from t cells away, a sublinear-time bound immediately prevents every cell from "seeing" more than just a very small portion of the input.[4] Meanwhile, the aspect of anonymity implies this restricted view must be a local one and (except for the border cells and those in their vicinity) completely detached from the cell's relative position in the automaton.

These observations are present in previous work on sublinear-time ACAs [60, 66, 105] in one form or the other. Indeed, locality is the driving aspect in the characterization from [105]; in Chapter 2 we extend the relevant result there from the case of constant to that of sublinear time (Lemmas 2.9 and 2.11). Meanwhile, in the work by Ibarra, Palis, and Kim [60] we find a partial solution to the issue of anonymity, which we discuss afterwards in Section 1.4.2.

It is interesting to note that shrinking CAs are still subject to the locality limitation to a considerable extent. Admittedly, unlike the other models from Section 1.2.1, one can overcome the distance between two cells by deleting everything in-between them. The price to pay, however, is that any information stored between the two cells is irrecoverably lost. In a sense, this is similar to the space restrictions that a streaming algorithm is subject to (i.e., having to choose what information to store and what to throw away). As Theorem 3.5 shows, in shrinking cellular automata this restriction has more extreme consequences.

The issue of locality is also present in sliding-window algorithms as a consequence of the algorithm's window being but a local view of the input. From this perspective one

[3] An exception to the anonymity restriction are the two border cells and the ones in their vicinity. This explains the occasional need to handle prefixes and suffixes of a word separately (in the results of Chapters 2 and 4, for instance).

[4] The corresponding phenomenon in physics is called *causality* and is also tightly connected to the workings of cellular automata in other contexts; see, e.g., [6, 34].

can say the algorithm is *local* with respect to the current contents of the window. As for anonymity, there does not appear to be any equivalent in sliding-window algorithms. In fact, there should not be one at all since there is nothing that prevents the algorithm from maintaining a counter that indicates its current position in the input stream.

1.4.2. Locally Verifiable Padding

In [60], Ibarra, Palis, and Kim show the existence of a non-regular language L that can be accepted by an ACA in $O(\log n)$ time. The words of L are of the following form:

$$w_0 \# w_1 \# w_2 \# \cdots \# w_{2^k-1}$$

where k is a non-negative integer and w_i is the binary representation of i of length k.[5] Let us refer to each w_i as a *block* and k as the *block length*. The words of L exhibit two key properties:

1. The structure of a word can be verified *in parallel* in a number of steps that is linear in the block length. This is because each block only has to check if the contents of its two neighbors are consistent with its own; that is, a block containing w_i must only verify that the preceding block contains w_{i-1} and the subsequent one w_{i+1}. Additionally, the leftmost (resp., rightmost) block must verify that it contains w_0 (resp., w_{2^k-1}). A block is aware that it is the left- or rightmost block since it "sees" the word's borders.
2. The word structure *forces* a specific number of blocks. Namely, for L as defined above, the number of blocks must be a power of two.

With these two observations it is straightforward to see that this word format can be used to implement a *locally verifiable form of padding*. Among others, this is used to obtain several results in Chapter 2 and it also forms the basis for the adapted presentation of inputs developed in Chapter 3. Note the format admits generalizations; for instance:

1. Replacing $2^k - 1$ with $g(k)$ for any function g that is efficiently computable (for some adequate notion thereof). This allows us to control the word length as we desire: The resulting words must be $\Theta(g(k)|w_{g(k)}|)$ long (parameterized on k).
2. Padding each block with any desirable (but globally fixed) number of (properly padded) trailing zeroes $b(k)$ for any efficiently computable function b. This forces each block to be $\Theta(b(k))$ long.

These generalizations are used to prove, among others, Theorems 2.13 and 4.2.

[5] To be more accurate, Ibarra, Palis, and Kim actually let the w_i be *minimal* binary representations (and also sort the w_i in reverse order); however, using *fixed-length* representations instead has several advantages. Besides them being more simple to work with (e.g., it is simpler to state the resulting word's length), they also give us a local encoding of the parameter k, which is very useful to have in some cases (e.g., padding, as discussed later in the text).

1.4.3. The Inclusion-Exclusion Principle

The *inclusion-exclusion principle* is a basic identity in combinatorics. For a positive integer n, let $[n]$ denote the set of the first n positive integers. The principle states that, given any events $E_1, \ldots, E_n$, we can rewrite the probability that any of the E_i occurs in terms of the probabilities of the E_i occurring simultaneously:[6]

$$\Pr\left[\bigcup_{i=1}^{n} E_i\right] = \sum_{j=1}^{n} \Pr[E_j] - \sum_{\substack{J \subseteq [n] \\ |J|=2}} \Pr\left[\bigcap_{i \in J} E_i\right] + \cdots + (-1)^{n+1} \Pr\left[\bigcap_{i=1}^{n} E_i\right].$$

Suppose we wish to analyze the probability that a T-time PACA C accepts some fixed input x. Recall that C accepts if and only if it reaches a configuration in at most T steps in which every cell is accepting. This means that we may not know in which step the PACA accepts (as a function of its input x and randomness r), and indeed it might even accept in multiple time steps (prior to its T-th step).[7]

To deal with this uncertainty, one first measure is to simply fix a time step $t < T$ and exclusively consider whether C accepts in time step t or not. The parameter t can be set non-uniformly, which leads to a clean construction. Now identify E_t with the event that C accepts x in step t. Then we immediately notice that the principle allows us to determine the probability that C accepts x (in the sense of the PACA definition) in terms of the probabilities of the E_t occurring simultaneously. Moreover, it turns out that the latter quantities are often easier to work with. This observation is a key ingredient in the proofs of Theorems 4.4 and 5.11.

One downside of the approach above is that the number of terms in the right-hand side of the equality grows exponentially in T. This leads to problems if our goal is to algorithmically determine (or estimate) the probability of C accepting. Therefore (although the same approach would still be valid) in Theorem 5.10 we use an alternative, simpler analysis with the added benefit that it leads to an improved result.

1.4.4. Simulation Techniques

As often the case in theoretical computer science, we develop simulation techniques for certain machines to simulate others in the same or in less powerful classes. Here we present two such constructions that are used in this dissertation in more than a single context.

[6] The principle is more commonly stated in terms of set cardinality. Here we present the same identity in terms of probabilities to better relate it to the subsequent discussion.

[7] It is of interest to note that, when the PACA does accept in multiple time steps, say in steps t_1 and t_2, we cannot even be sure whether the events of C accepting in steps t_1 and t_2 (respectively) are correlated or not. For instance, it may be that the cells of C accept in t_2 if and only if they accepted in t_1, but it is also possible that their behavior in t_2 depends exclusively on coin tosses after t_1.

1.4.4.1. Simulating Multiple Automata in Parallel

One very standard form of simulation in CA theory is that of a CA simulating multiple copies of other CAs. Here we will address the case in which we wish to simulate a *constant* number of other CAs.

We start from the underlying construction, which is certainly not new and is best attributed to folklore. Let the CAs $C_1, \dots, C_n$ be given (for some constant n). The task is to construct a CA S that simulates the C_i in parallel in the sense that, for each i, the configurations that C_i assumes over time can be observed in the time-state diagram of S. (We keep the discussion considerably abstract for now.) To do so, we let the state set of S contain n many components, where we associate the i-th component with C_i; indeed, the value of the i-th component is equal to the state that the respective cell would have in C_i. To perform the actual simulation, we then advance each C_i according to some global update scheme (using the values of the i-th components in the neighborhood of each cell to compute the cell's next value). We note that, in our constructions, we only need to update one C_i at a time, but it is also natural to consider other schemes (e.g., updating all the C_i simultaneously).

The appropriate choice of an update scheme depends on the context in which the simulation technique is used. In the deterministic setting of ACA, we use a simple round-robin scheme to show closure under union in Proposition 2.7 and Theorem 2.25. The same approach applies to PACA in Proposition 4.21. More interestingly, the same strategy extends to error reduction on one-sided error PACA (Proposition 4.12).

To obtain closure under intersection, a more involved update scheme is needed. In particular, the scheme involves *rewinding* copies so as to explore every possible combination of time steps. For example, for $n = 2$, we need S to visit step t_1 of C_1 and step t_2 of C_2 simultaneously at least once for every combination of values of t_1 and t_2 (up to some constant). This is the approach we take in Proposition 4.21. It also applies to the setting of ACA, of course, but said result was already shown in [105].

An even more complex update scheme is needed for error reduction for two-sided error PACA (Proposition 4.13). In that case we need to consider not only every possible combination (as in the aforementioned case of closure under union) but also every possible majority over the C_i. We refer to the respective discussion in Chapter 4 for the details.

1.4.4.2. Simulation by Streaming Algorithms

The second simulation we describe is the simulation of (variants of) CAs by streaming algorithms.

For a moment, let us step back and suppose we are in the classical setting of CAs that operate in an unbounded amount of time. The folklore strategy for simulating CAs by Turing machines in this setting is to have the Turing machine maintain on its tape a copy of the configuration of the CA. For each step of the CA, the Turing machine head visits

one cell at a time and updates its state. Once it has updated every cell, it returns to the first one and the process begins anew.

This strategy is tailored to the case where the time-space diagram is arbitrarily deep but not very wide. In sublinear-time CAs, however, we have the exact opposite: The time-space diagram is *very wide* but also *very shallow*. This suggests there might be a more efficient simulation by Turing machines, which is indeed the case.

In the classical simulation, we explore the time-space diagram row by row, that is, horizontally. When the dimensions are reversed, however, it should make more sense to explore it vertically—if we could. The problem is that, unlike the case of rows, we cannot compute a column of the time-space diagram from its previous one. Nevertheless, one can compute a *diagonal* from the previous one, which is indeed the approach we follow.

The efficiency of the construction comes from the fact that the shallowness of the time-space diagram implies the diagonals we need to maintain are also not very long. In fact, to simulate a T-time CA (with the standard neighborhood), we need only maintain at most two diagonals of size at most T. This means there is a connection between the efficiency in time of the CA and the space required to simulate it. Such a connection is to be expected, of course, as the literature that connects space-efficient to parallel machines is quite vast.

This kind of simulation is used in two contexts in this dissertation. In the first one, we use it to show that streaming algorithms can simulate shrinking CAs efficiently (Theorem 3.4). The additional challenge to overcome there is to show that the simulation is also compatible with the setting of shrinking CAs, in which cells might vanish without any previous notice. The second context is that of sliding-window algorithms in connection to PACAs (Theorem 5.25) and which forms the core of our approach to show derandomization results for PACAs (i.e., Theorems 5.9 and 5.11).

1.4.5. Shattering and Interleaving

The last techniques we discuss are *shattering* and *interleaving*. Unlike the others, which apply to cellular automata, these are targeted at sliding-window algorithms. In particular, they are one of the key ingredients in our construction of pseudorandom generators in Chapter 5 (i.e., Theorems 5.5 and 5.7).

Recall the current state of a sliding-window algorithm A depends exclusively on the contents of its window of size t. This means that, if we fix any substring y of length t in the input to A, then necessarily the state of A after reading y is uniquely determined by y alone.

Let us formalize this idea. Assume that $x = x_1yx_2$ is an input to A where $|y| = t$, and let $A(z)$ denote the state of A after having read input z. (For now, we assume the algorithm A is deterministic.) Then the sliding-window property implies that $A(x_1y) = A(zy)$ holds for *any* string z. Since the state $A(xy)$ is uniquely determined, we may *shatter* the algorithm A into two pieces: one that processes x_1y (i.e., working as A on input x_1y) and another that processes x_2 (i.e., working as A on input x_2 *from the initial state* $A(xy)$).

The above gives us shattering in the context of deterministic algorithms; however, the interesting application is to *randomized* sliding-window algorithms. Consider thus the case where A is randomized and obtains its randomness from an auxiliary binary input string r. In addition, assume the input stream x to A is fixed.[8] Then we apply the same observation as above, fixing t many bits in r to *shatter* A into two algorithms A_1 and A_2. In particular, we may then analyze the acceptance probabilities of the two parts A_1 and A_2 independently from one another (as they are, indeed, two distinct algorithms, even if they originate from the same algorithm A).

The point is that we may perform this operation not only in one position but all across r. Suppose that A expects a random string r of length nt. Then we *fix every other substring of t bits* by *interleaving* fixed and "free" strings as follows:

$$r = x_1 y_1 x_2 y_2 \cdots x_n y_n$$

where the y_i are fixed and the x_i are free in the sense that they originate from a binary randomness source. By doing so, we have shattered A into n many algorithms $A_1, \ldots, A_n$, each expecting a random string of size t. Since the A_i operate independently from one another—and also using much less random bits than the algorithm A—, we may expect to generate pseudorandom strings for them much more easily (than if we had tried to fool A as a whole).

Having to fix half of the bits in r may sound rather excessive; however, we stress this is not an issue at all since we may generate $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ using independent seeds. That is, we can first fix x to an arbitrary string and, using the shattering argument, replace y with a pseudorandom string. Having done so, we then fix y and replace x with a pseudorandom string (using the same shattering argument) that is generated independently from y. The resulting r is then completely pseudorandom. We refer the reader to Chapter 5 for the details.

1.5. Organization

Part II of this dissertation is subdivided into five chapters, each of which is taken verbatim (up to minor changes in notation and references in order to improve cohesion) from a stand-alone contribution to international conferences or journals. Chapters 2 and 3 are already published work whereas the most recent ones, that is, Chapters 4 and 5 are currently under peer review or, in the case of Chapter 6, are pending publication. If applicable, the corresponding published or preprint versions are given at the beginning of the chapter. Each chapter is self-contained and may be read independently of the others.

[8] This is a very standard approach taken in derandomization. Having fixed the ordinary input, we will produce a *pseudorandom* set of values R such that, if we run A with a random input r uniformly chosen from R, then the algorithm's behavior is "approximately the same" as if r is drawn at random from the uniform distribution over all possible random inputs to A.

All chapters are the fruit of independent work by the author of this dissertation. The sole exception is Chapter 6, which is co-authored with Thomas Worsch. The key aspects of the construction there are all due to the present author.

In Section 1.3 we already described the contents of the subsequent chapters, which we now repeat in a succinct manner for the reader's convenience. The first three chapters are directly concerned with the sublinear-time CA models previously mentioned: Chapter 2 addresses deterministic ACAs, Chapter 3 shrinking CAs (SCAs), and Chapter 4 probabilistic ACAs (PACAs). Subsequently, in Chapter 5 we state and prove our results on sliding-window algorithms as well as the low-space derandomization results for PACAs. Finally, Chapter 6 covers the result on the prediction problem for fungal sandpile automata.

Part II.

Contents

2. Sublinear-Time Recognition and Decision by One-Dimensional Cellular Automata

Published versions: [81, 83]

Abstract

After an apparent hiatus of roughly 30 years, we revisit a seemingly neglected subject in the theory of (one-dimensional) cellular automata: sublinear-time computation. The model considered is that of ACAs, which are language acceptors whose acceptance condition depends on the states of all cells in the automaton. We prove a time hierarchy theorem for sublinear-time ACA classes, analyze their intersection with the regular languages, and, finally, establish strict inclusions in the parallel computation classes SC and (uniform) AC. As an addendum, we introduce and investigate the concept of a decider ACA (DACA) as a candidate for a decider counterpart to (acceptor) ACAs. We show the class of languages decidable in constant time by DACAs equals the locally testable languages, and we also determine $\Omega(\sqrt{n})$ as the (tight) time complexity threshold for DACAs up to which no advantage compared to constant time is possible.

2.1. Introduction

While there have been several works on linear- and real-time language recognition by cellular automata over the years (see, e.g., [69, 111] for an overview), interest in the sublinear-time case has been scanty at best. We can only speculate that this has been due to a certain obstinacy concerning what is now the established acceptance condition for cellular automata, namely that the first cell determines the automaton's response, despite alternatives being long known [98]. Under this condition, only a constant-size prefix can ever influence the automaton's decision, which effectively dooms sublinear time to be but a trivial case just as it is for (classical) Turing machines, for example. Nevertheless, at least in the realm of Turing machines, this shortcoming was readily circumvented by adding a random access mechanism to the model, thus sparking rich theories on parallel computation [26, 103], probabilistically checkable proofs [108], and property testing [37, 101].

In the case of cellular automata, the adaptation needed is an alternate (and by all means novel) acceptance condition, covered in Section 2.2. Interestingly, in the resulting model,

called ACA, parallelism and local behavior seem to be more marked features, taking priority over cell communication and synchronization algorithms (which are the dominant themes in the linear- and real-time constructions). As mentioned above, the body of theory on sublinear-time ACAs is very small and, to the best of our knowledge, resumes itself to [60, 66, 105]. Ibarra, Palis, and Kim [60] show sublinear-time ACAs are capable of recognizing non-regular languages and also determine a threshold (namely $\Omega(\log n)$) up to which no advantage compared to constant time is possible. Meanwhile, Kim and McCloskey [66] and Sommerhalder and van Westrhenen [105] analyze the constant-time case subject to different acceptance conditions and characterize it based on the locally testable languages, a subclass of the regular languages.

Indeed, as covered in Section 2.3, the defining property of the locally testable languages, that is, that words which locally appear to be the same are equivalent with respect to membership in the language at hand, effectively translates into an inherent property of acceptance by sublinear-time ACAs. In Section 2.4, we prove a time hierarchy theorem for sublinear-time ACAs as well as further relate the language classes they define to the regular languages and the parallel computation classes SC and (uniform) AC. In the same section, we also obtain an improvement on a result of [60]. Finally, in Section 2.5 we consider a plausible model of ACAs as language deciders, that is, machines which must not only accept words in the target language but also explicitly reject those which do not. Section 2.6 concludes the chapter.

2.2. Definitions

We assume the reader is familiar with the theory of formal languages and cellular automata as well as with computational complexity theory (see, e.g., standard references [5, 30]). This section reviews basic concepts and introduces ACAs.

The set of integers is denoted by $\mathbb{Z}$ and that of (strictly) positive integers by $\mathbb{N}_+$. Furthermore, $\mathbb{N}_0 = \mathbb{N}_+ \cup \{0\}$. For a function $f : A \to B$ and $A' \subseteq A$, $f|A'$ indicates the restriction of f to A'. The set of functions $f : A \to B$ is denoted by B^A. For a word $w \in \Sigma^*$ over an alphabet Σ, $w(i)$ is the i-th symbol of w (starting with the 0-th symbol), and $|w|_x$ is the number of occurrences of $x \in \Sigma$ in w. For $k \in \mathbb{N}_0$, $p_k(w)$, $s_k(w)$, and $I_k(w)$ are the prefix, suffix and set of infixes of length k of w, respectively, where $p_{k'}(w) = s_{k'}(w) = w$ and $I_{k'}(w) = \{w\}$ for $k' \geq |w|$. Also, $\Sigma^{\leq k}$ is the set of words $w \in \Sigma^*$ for which $|w| \leq k$. Unless otherwise noted, n stands for the input length.

2.2.1. (Strictly) Locally Testable Languages

The class REG of regular languages is defined in terms of (deterministic) automata with finite memory and which read their input in a single direction (i.e., from left to right), one symbol at a time; once all symbols have been read, the machine outputs a single bit representing its decision. In contrast, a *scanner* is a memoryless machine which reads a

span of $k \in \mathbb{N}_+$ symbols at a time of an input provided with start and end markers (so it can handle prefixes and suffixes separately); the scanner validates every such substring it reads using the same predicate, and it accepts if and only if all these validations are successful. The languages accepted by these machines are the strictly locally testable languages.[1]

Definition 2.1 (strictly locally testable)**.** Let Σ be an alphabet. A language $L \subseteq \Sigma^*$ is *strictly locally testable* if there is some $k \in \mathbb{N}_+$ and sets $\pi, \sigma \subseteq \Sigma^{\leq k}$ and $\mu \subseteq \Sigma^k$ such that, for every word $w \in \Sigma^*$, $w \in L$ if and only if $p_k(w) \in \pi$, $I_k(w) \subseteq \mu$, and $s_k(w) \in \sigma$. The class of strictly locally testable languages is denoted by SLT.

A more general notion of locality is provided by the locally testable languages. Intuitively, L is locally testable if a word w being in L or not is entirely dependent on a property of the substrings of w of some constant length $k \in \mathbb{N}_+$ (that depends only on L, not on w). Thus, if any two words have the same set of substrings of length k, then they are equivalent with respect to being in L:

Definition 2.2 (locally testable)**.** Let Σ be an alphabet. A language $L \subseteq \Sigma^*$ is *locally testable* if there is some $k \in \mathbb{N}_+$ such that, for every $w_1, w_2 \in \Sigma^*$ with $p_k(w_1) = p_k(w_2)$, $I_k(w_1) = I_k(w_2)$, and $s_k(w_1) = s_k(w_2)$ we have that $w_1 \in L$ if and only if $w_2 \in L$. The class of locally testable languages is denoted by LT.

The class LT is the *Boolean closure* of SLT, that is, its closure under union, intersection, and complement [77]. In particular, the inclusion $\mathsf{SLT} \subsetneq \mathsf{LT}$ is proper [76].

2.2.2. Cellular Automata

We are strictly interested in one-dimensional cellular automata with the standard neighborhood. For $r \in \mathbb{N}_0$, let

$$N_r(z) = \{z' \in \mathbb{Z} \mid |z - z'| \leq r\}$$

denote the *extended neighborhood of radius r* of the cell $z \in \mathbb{Z}$.

Definition 2.3 (cellular automaton)**.** A *cellular automaton* (*CA*) C is a triple (Q, δ, Σ) where Q is a finite, non-empty set of *states*, $\delta \colon Q^3 \to Q$ is the *local transition function*, and $\Sigma \subseteq Q$ is the *input alphabet*. An element of Q^3 (resp., $Q^{\mathbb{Z}}$) is called a *local* (resp., *global*) *configuration* of C. The function δ induces the *global transition function* $\Delta \colon Q^{\mathbb{Z}} \to Q^{\mathbb{Z}}$ on the configuration space $Q^{\mathbb{Z}}$ by

$$\Delta(c)(z) = \delta(c(z-1), c(z), c(z+1)),$$

where $z \in \mathbb{Z}$ is a cell and $c \in Q^{\mathbb{Z}}$.

[1] The term "(locally) testable in the strict sense" ((L)TSS) is also common [66, 76, 77].

Our interest in CAs is as machines which receive an input and process it until a final state is reached. The input is provided from left to right, with one cell for each input symbol. The surrounding cells are inactive and remain so for the entirety of the computation (i.e., the CA is bounded). It is customary for CAs to have a distinguished cell, usually cell zero, which communicates the machine's output. As mentioned in the introduction, this convention is inadequate for computation in sublinear time; instead, we require the finality condition to depend on the entire (global) configuration (modulo inactive cells):

Definition 2.4 (CA computation)**.** There is a distinguished state $q \in Q \setminus \Sigma$, called the *inactive state*, which, for every $z_1, z_2, z_3 \in Q$, satisfies $\delta(z_1, z_2, z_3) = q$ if and only if $z_2 = q$. A cell not in state q is said to be *active*. For an input $w \in \Sigma^*$, the *initial configuration* $c_0 = c_0(w) \in Q^{\mathbb{Z}}$ of C for w is $c_0(i) = w(i)$ for $i \in \{0, \dots, |w| - 1\}$ and $c_0(i) = q$ otherwise. For $F \subseteq Q \setminus \{q\}$, a configuration $c \in Q^{\mathbb{Z}}$ is *F-final* (for w) if there is a (minimal) $\tau \in \mathbb{N}_0$ such that $c = \Delta^\tau(c_0)$ and c contains only states in $F \cup \{q\}$. In this context, the sequence $c_0, \dots, \Delta^\tau(c_0) = c$ is the *trace* of w, and τ is the *time complexity* of C (with respect to F and w).

Because we effectively consider only bounded CAs, the computation of w involves exactly $|w|$ active cells. The surrounding inactive cells are needed only as markers for the start and end of w. As a side effect, the initial configuration $c_0 = c_0(\varepsilon)$ for the empty word ε is stationary (i.e., $\Delta(c_0) = c_0$) regardless of the choice of δ. Since this is the case only for ε, we disregard it for the rest of the chapter; that is, we assume ε is not contained in any of the languages considered.

Finally, we relate final configurations and computation results. We adopt an acceptance condition as in [98, 105] and obtain a so-called ACA; here, the "A" of "ACA" refers to the property that all (active) cells are relevant for acceptance.

Definition 2.5 (ACA)**.** An *ACA* is a CA C with a non-empty subset $A \subseteq Q \setminus \{q\}$ of *accept states*. For $w \in \Sigma^+$, if C reaches an A-final configuration, we say C *accepts* w. We write $L(C)$ for the set of words accepted by C. For a function $t\colon \mathbb{N}_+ \to \mathbb{N}_0$, $\mathrm{ACA}[t]$ denotes the class of languages that can be accepted by an ACA with time complexity at most t; that is, $L \in \mathrm{ACA}[t]$ if and only if there is an ACA C with $L = L(C)$ and such that, for every $w \in L$, the time complexity of C with respect to A and w is $\leq t(|w|)$.

The inclusion $\mathrm{ACA}[t_1] \subseteq \mathrm{ACA}[t_2]$ is immediate for functions $t_1, t_2\colon \mathbb{N}_+ \to \mathbb{N}_0$ with $t_1(n) \leq t_2(n)$ for every $n \in \mathbb{N}_+$. Because Definition 2.5 allows multiple accept states, it is possible for each state z (that is not an accept state) to have a corresponding accept state z_A. In the rest of this chapter, when we say a cell becomes (or marks itself as) accepting (without explicitly mentioning its state), we intend to say it changes from such a state z to z_A.

Figure 2.1 illustrates the computation of an ACA with input alphabet $\Sigma = \{0, 1\}$ and which accepts $\{01\}^+$ with time complexity equal to one (step). The local transition function is such that

$$\delta(0, 1, 0) = \delta(1, 0, 1) = \delta(q, 0, 1) = \delta(0, 1, q) = a,$$

a being the (only) accept state, and $\delta(z_1, z_2, z_3) = z_2$ for $z_2 \neq a$ and arbitrary z_1 and z_3.

Figure 2.1.: Computation of an ACA which recognizes $L = \{01\}^+$. The input words are $010101 \in L$ and $001010 \notin L$, respectively.

2.3. First Observations

This section recalls results on sublinear-time ACA computation (i.e., $\mathsf{ACA}[t]$ where $t(n) = o(n)$) from [60, 66, 105] and provides some additional remarks. We start with the constant-time case (i.e., $\mathsf{ACA}[O(1)]$). Here, the connection between scanners and ACAs is apparent: If an ACA accepts an input w in time $\tau = \tau(w)$, then w can be verified by a scanner with an input span of $2\tau + 1$ symbols and using the predicate induced by the local transition function of the ACA (i.e., the predicate is true if and only if the symbols read correspond to $N_\tau(z)$ for some cell z in the initial configuration and z is accepting after τ steps).

Constant-time ACA computation has been studied in [66, 105]. Although in [66] we find a characterization based on a hierarchy over SLT, the acceptance condition there differs slightly from that in Definition 2.5; in particular, the automata there run for a number of steps which is fixed for each automaton, and the outcome is evaluated (only) in the final step. In contrast, in [105] we find the following, where $\mathsf{SLT}_\cup$ denotes the closure of SLT under union:

Theorem 2.6 ([105])**.** $\mathsf{ACA}[O(1)] = \mathsf{SLT}_\cup$.

Thus, $\mathsf{ACA}[O(1)]$ is closed under union. In fact, more generally:

Proposition 2.7. *For any $t\colon \mathbb{N}_+ \to \mathbb{N}_0$, $\mathsf{ACA}[O(t)]$ is closed under union.*

Proof. Let L_1 and L_2 be languages accepted by the ACAs C_1 and C_2, respectively, in $O(t)$ time. Furthermore, let Q_i (resp., Q_i^a) denote the set of states (resp., accept states) of C_i. We construct an ACA C which accepts $L_1 \cup L_2$ as follows: C simulates C_1 and C_2 in interleaved fashion, alternating between one step of C_1 and one step of C_2, and accepts whenever either C_i does. To this end, each cell maintains components $q_1 \in Q_1$, $q_2 \in Q_2$, and $r \in \{1, 2\}$, where r indicates which of the two simulations is to be updated next; that is, at each step, a cell next updates q_r according to the local transition function of C_r and the q_r states of its neighbors. At the start of the computation, all cells simultaneously initialize $r = 1$. The set of accept states of C is

$$Q_1 \times Q_2^a \times \{1\} \cup Q_1^a \times Q_2 \times \{2\}.$$

Thus, if C_i accepts in exactly τ steps, then C will accept in exactly $2\tau + i - 1$ steps and vice-versa. It follows that $L(C) = L_1 \cup L_2$ and the time complexity of C is in $O(2t) = O(t)$. □

The class ACA$[O(1)]$ is closed under intersection [105]. It is an open question whether ACA$[O(t)]$ is also closed under intersection for every $t(n) = o(n)$. In particular, note that taking the construction of the proof above and setting, for example, $Q_1^a \times Q_2^a \times \{1\}$ as the set of accept states is insufficient (as it is not guaranteed that C_1 and C_2 accept *at the same time*).

Moving beyond constant time, in [60] we find the following:

Theorem 2.8 ([60])**.** *For $t(n) = o(\log n)$,* ACA$[t] \subseteq$ REG*.*

In [60] we also find an example for a non-regular language in ACA$[O(\log n)]$ which is essentially a variation of the language

$$\mathsf{BIN} = \{\mathrm{bin}_k(0)\#\,\mathrm{bin}_k(1)\#\cdots\#\,\mathrm{bin}_k(2^k-1) \mid k \in \mathbb{N}_+\}$$

where $\mathrm{bin}_k(m)$ is the k-digit binary representation of $m \in \{0,\ldots,2^k-1\}$.

To illustrate the ideas involved, we present an example related to BIN (though it results in a different time complexity) and which is also useful in later discussions in Section 2.5. Let $w_k(i) = 0^i 10^{k-i-1}$ and consider the language

$$\mathsf{IDMAT} = \{w_k(0)\#w_k(1)\#\cdots\#w_k(k-1) \mid k \in \mathbb{N}_+\}$$

of all identity matrices in line-for-line representations, where the lines are separated by # symbols.[2]

We now describe an ACA for IDMAT; the construction closely follows the aforementioned one for BIN found in [60] (and the difference in complexity is only due to the different number and size of blocks in the words of IDMAT and BIN). Denote each group of cells initially containing a (maximally long) $\{0,1\}^+$ substring of $w \in$ IDMAT by a *block*. Each block of size b propagates its contents to the neighboring blocks (in separate registers); using a textbook CA technique, this requires exactly $2b$ steps. Once the strings align, a block initially containing $w_k(i)$ verifies it has received $w_k(i-1)$ and $w_k(i+1)$ from its left and right neighbor blocks (if either exists), respectively. The cells of a block and its delimiters become accepting if and only if the comparisons are successful and there is a single # between the block and its neighbors. This process takes linear time in b; since any $w \in$ IDMAT has $O(\sqrt{|w|})$ many blocks, each with $b = O(\sqrt{|w|})$ cells, it follows that IDMAT $\in$ ACA$[O(\sqrt{n})]$.

To show the above construction is time-optimal, we use the following observation, which is also central in proving several other results in this chapter:

Lemma 2.9. *Let C be an ACA, and let w be an input which C accepts in exactly $\tau = \tau(w)$ steps. Then, for every input w' such that $p_{2\tau}(w) = p_{2\tau}(w')$, $I_{2\tau+1}(w') \subseteq I_{2\tau+1}(w)$, and $s_{2\tau}(w) = s_{2\tau}(w')$, C accepts w' in at most τ steps.*

[2] Alternatively, one can also think of IDMAT as a (natural) problem on graphs presented in the adjacency matrix representation.

The lemma is intended to be used with $\tau < |w|/2$ since otherwise we have $w = w'$ and the statement is trivial. We can apply the lemma, for instance, to show that

$$\mathsf{SOMEONE} = \left\{w \in \{0,1\}^+ \mid |w|_1 \geq 1\right\}$$

is not in $\mathsf{ACA}[t]$ for any $t(n) = o(n)$ (e.g., set $w = 0^k 10^k$ and $w' = 0^{2k+1}$ for large $k \in \mathbb{N}_+$). It follows that $\mathsf{REG} \not\subseteq \mathsf{ACA}[t]$ for any $t(n) = o(n)$.

Proof. Let A be the set of accept states of C, and let c_0 and c'_0 denote the initial configurations for w and w', respectively. We prove that $c'_\tau = \Delta^\tau(c'_0)$ is A-final.

Let $i \in \mathbb{Z}$. If all of $c'_0|N_\tau(i)$ is inactive, then cell i is also inactive in c'_τ (i.e., $c'_\tau(i) = q$). If $c'_0|N_\tau(i)$ contains both inactive and active states, then $i < \tau$ or $i \geq |w'| - \tau$, in which case $p_{2\tau}(w) = p_{2\tau}(w')$ and $s_{2\tau}(w) = s_{2\tau}(w')$ imply $c'_0|N_\tau(i) = c_0|N_\tau(i)$. Finally, if $c'_0|N_\tau(i)$ is purely active, $c'_0|N_\tau(i)$ (seen as a word over the input alphabet of C) is in $I_{2\tau+1}(w') \subseteq I_{2\tau+1}(w)$; as a result, there is $j \in \mathbb{Z}$ with $\tau \leq j < |w| - \tau$ and such that $c'_0|N_\tau(i) = c_0|N_\tau(j)$. In both the previous cases, since c_τ is A-final, it follows that $c'_\tau(i) \in A$, thus implying c'_τ is A-final. □

Since the complement of SOMEONE (respective to $\{0,1\}^+$) is $\{0\}^+$ and $\{0\}^+ \in \mathsf{ACA}[O(1)]$ (e.g., simply set 0 as the ACA's accept state), $\mathsf{ACA}[t]$ is not closed under complement for any $t(n) = o(n)$. Also, SOMEONE is a regular language and $\mathsf{BIN} \in \mathsf{ACA}[O(\log n)]$ is not, so we have:

Proposition 2.10. *For any function $t\colon \mathbb{N}_+ \to \mathbb{N}_0$ with $t(n) = \Omega(\log n)$ and $t(n) = o(n)$, $\mathsf{ACA}[t]$ and REG are incomparable.*

If the inclusion of infixes in Lemma 2.9 is strengthened to an equality, one may apply it in both directions and obtain the following stronger statement:

Lemma 2.11. *Let C be an ACA with time complexity bounded by $t\colon \mathbb{N}_+ \to \mathbb{N}_0$ (i.e., C accepts any input of length n in at most $t(n)$ steps). Then, for any two inputs w and w' with $p_{2\mu}(w) = p_{2\mu}(w')$, $I_{2\mu+1}(w) = I_{2\mu+1}(w')$, and $s_{2\mu}(w) = s_{2\mu}(w')$ where $\mu = \max\{t(|w|), t(|w'|)\}$, we have that $w \in L(C)$ if and only if $w' \in L(C)$.*

Proof. The key idea is that, for any $k \in \mathbb{N}_0$, $p_k(w) = p_k(w')$ implies $p_{k'}(w) = p_{k'}(w')$ for every $k' \leq k$, and the same is true for $s_k(w) = s_k(w')$ and $I_k(w) = I_k(w')$. Hence, if $w \in L(C)$ and C accepts w in exactly $\tau = \tau(w) \leq t(|w|)$ steps, then by Lemma 2.9 we have that C accepts w' in at most $\tau \leq \mu$ many steps. Conversely, if $w' \in L(C)$ and C accepts w' in exactly $\tau = \tau(w') \leq t(|w'|)$ steps, then using Lemma 2.9 again (with w in place of w' and vice-versa) we obtain that C accepts w in at most $\tau \leq \mu$ many steps. □

Finally, we can show our ACA for IDMAT is time-optimal (asymptotically speaking):

Proposition 2.12. *For any $t(n) = o(\sqrt{n})$, $\mathsf{IDMAT} \notin \mathsf{ACA}[t]$.*

Proof. Let $n \in \mathbb{N}_+$ be such that $t(n) < \sqrt{n}/8$, and let $w \in \mathsf{IDMAT}$ be given with $|w| = n$. In particular, $w \in \mathsf{IDMAT}$ implies there is $k \in \mathbb{N}_+$ such that

$$w = w_k(0)\# \cdots \# w_k(k-1)$$

and $|w| = k^2 + k - 1$; without restriction, we also assume k is even. Let w' be the word obtained from w by replacing its $(k/2-1)$-th block with $w_k(k/2)$. Then we have $p_{2t(n)}(w) = p_{2t(n)}(w')$, $s_{2t(n)}(w) = s_{2t(n)}(w')$, and $I_{2t(n)+1}(w) \subseteq I_{2t(n)+1}(w')$, where the latter follows from $w_k(k/2) = 0^{k/2}10^{k/2-1}$, $|w_k(i)| = k$, and $t(n) < (k+1)/8$. Thus, by Lemma 2.9, if an ACA with time complexity bounded by t accepts w, so does it accept the word $w' \notin \mathsf{IDMAT}$. □

2.4. Main Results

In this section, we present various results regarding $\mathsf{ACA}[t]$ where $t(n) = o(n)$. First, we obtain a time hierarchy theorem; that is, (under plausible conditions) $\mathsf{ACA}[t'] \subsetneq \mathsf{ACA}[t]$ for $t'(n) = o(t(n))$. Next, we show $\mathsf{ACA}[t] \cap \mathsf{REG}$ is (strictly) contained in LT and also present an improvement to Theorem 2.8. Finally, we study inclusion relations between $\mathsf{ACA}[t]$ and the SC and (uniform) AC hierarchies. Save for the material covered so far, all three subsections stand out independently from one another.

2.4.1. Time Hierarchy

For functions $f, t\colon \mathbb{N}_+ \to \mathbb{N}_0$, we say f is *time-constructible by CAs in $t(n)$ time* if there is a CA C which, on input 1^n, reaches a configuration containing the value $f(n)$ (binary-encoded) in at most $t(n)$ steps.[3] Note that, since CAs can simulate (one-tape) Turing machines in real-time, any function constructible by Turing machines (in the corresponding sense) is also constructible by CAs.

Theorem 2.13. *Let $f, g\colon \mathbb{N}_+ \to \mathbb{N}_0$ be functions with $f(n) = \omega(n)$, $f(n) \leq 2^n$, $g(n) = 2^{n-\lfloor \log f(n) \rfloor}$, and let f and g be time-constructible (by CAs) in $f(n)$ time. Furthermore, let $t\colon \mathbb{N}_+ \to \mathbb{N}_0$ be such that*

$$3f(k) \leq t(f(k) \cdot g(k)) \leq cf(k)$$

for some constant $c \geq 3$ and all but finitely many $k \in \mathbb{N}_+$. Then, for every $t'(n) = o(t(n))$, $\mathsf{ACA}[t'] \subsetneq \mathsf{ACA}[t]$.

[3] Just as is the case for Turing machines, there is not a single definition for time-constructibility by CAs (see, e.g., [63] for an alternative). Here, we opt for a plausible variant which has the benefit of simplifying the ensuing line of argument.

Given $a > 1$, this can be used, for instance, with any time-constructible $f \in \Theta(n^a)$ (resp., $f \in \Theta(2^{n/a})$, in which case $a = 1$ is also possible) and $t \in \Theta(\log n)^a$ (resp., $t \in \Theta(n^{1/a})$). The proof idea is to construct a language L similar to BIN (see Section 2.3) in which every $w \in L$ has length exponential in the size of its blocks while the distance between any two blocks is $\Theta(t(|w|))$. Due to Lemma 2.9, the latter implies L is not recognizable in $o(t(|w|))$ time.

Proof. For simplicity, let $f(n) > n$. Consider $L = \{w_k \mid k \in \mathbb{N}_+\}$ where

$$w_k = \mathrm{bin}_k(0)\#^{f(k)-k}\,\mathrm{bin}_k(1)\#^{f(k)-k}\cdots\mathrm{bin}_k(g(k)-1)\#^{f(k)-k}$$

and note $|w_k| = f(k) \cdot g(k)$. Because $t(|w_k|) = O(f(k))$ and $f(k) = \omega(k)$, given any $t'(n) = o(t(n))$, setting $w = w_k$, $w' = 0^k\#^{|w_k|-k}$, and $\tau = t'(|w_k|)$ and applying Lemma 2.9 for sufficiently large k yields $L \notin \mathsf{ACA}[t']$.

By assumption, it suffices to show $w = w_k \in L$ is accepted by an ACA C in at most $3f(k) \leq t(|w|)$ steps for sufficiently large $k \in \mathbb{N}_+$. The cells of C perform two procedures P_1 and P_2 simultaneously: The procedure P_1 is as in the ACA for BIN (see Section 2.3) and ensures that the blocks of w have the same length, that the respective binary encodings are valid, and that the last value is correct (i.e., equal to $g(k) - 1$). In P_2, each block computes $f(k)$ as a function of its block length k. Subsequently, the value $f(k)$ is decreased using a real-time counter (see, e.g., [63] for a construction). Every time the counter is decremented, a signal starts from the block's leftmost cell and is propagated to the right. This allows every group of cells of the form bs with $b \in \{0,1\}^+$ and $s \in \{\#\}^+$ to assert there are precisely $f(k)$ symbols in total (i.e., $|bs| = f(k)$). A cell is accepting if and only if it is accepting both in P_1 and P_2. The proof is complete by noticing either procedure takes a maximum of $3f(k)$ steps (again, for sufficiently large k). □

2.4.2. Intersection with the Regular Languages

In light of Proposition 2.10, we now consider the intersection $\mathsf{ACA}[t] \cap \mathsf{REG}$ for $t(n) = o(n)$ (in the same spirit as a conjecture by Straubing [107]). For this section, we assume the reader is familiar with the theory of syntactic semigroups (see, e.g., [33] for an in-depth treatment).

Given a language L, let $\mathrm{SS}(L)$ denote the syntactic semigroup of L. It is well-known that $\mathrm{SS}(L)$ is finite if and only if L is regular. A semigroup S is a *semilattice* if $x^2 = x$ and $xy = yx$ for every $x, y \in S$. Additionally, S is *locally semilattice* if eSe is a semilattice for every *idempotent* $e \in S$, that is, $e^2 = e$. We use the following characterization of locally testable languages:

Theorem 2.14 ([17, 76])**.** *We have $L \in \mathsf{LT}$ if and only if $\mathrm{SS}(L)$ is finite and locally semilattice.*

In conjunction with Lemma 2.9, this yields the following, where the strict inclusion is due to $\mathsf{SOMEONE} \notin \mathsf{ACA}[t]$ (since $\mathsf{SOMEONE} \in \mathsf{LT}$; see Section 2.3):

Theorem 2.15. *For every $t(n) = o(n)$, $\mathsf{ACA}[t] \cap \mathsf{REG} \subsetneq \mathsf{LT}$.*

Proof. Let $L \in \mathsf{ACA}[t]$ be a language over the alphabet Σ and, in addition, let $L \in \mathsf{REG}$, that is, $S = \mathrm{SS}(L)$ is finite. By Theorem 2.14, it suffices to show S is locally semilattice. To that end, let $e \in S$ be idempotent, and let $x, y \in S$.

To show $(exe)(eye) = (eye)(exe)$, let $a, b \in \Sigma^*$ and consider the words $u = a(exe)(eye)b$ and $v = a(eye)(exe)b$. For $m \in \mathbb{N}_+$, let

$$u'_m = a(e^m x e^m)(e^m y e^m)b,$$

and let $r \in \mathbb{N}_+$ be such that $r > \max\{|x|, |y|, |a|, |b|\}$ and also

$$t\left(\left|u'_{2r+1}\right|\right) < \frac{\left|u'_{2r+1}\right|}{16|e|} < r.$$

Since e is idempotent, u and $u' = u'_{2r+1}$ belong to the same class in S; that is, $u' \in L$ if and only if $u \in L$. The same is true for v and

$$v' = a(e^{2r+1} y e^{2r+1})(e^{2r+1} x e^{2r+1})b.$$

Furthermore, $p_{2r}(u') = p_{2r}(v')$, $I_{2r+1}(u') = I_{2r+1}(v')$, and $s_{2r}(u') = s_{2r}(v')$ hold. Since $L \in \mathsf{ACA}[t]$, Lemma 2.11 applies.

The proof of $(exe)(exe) = exe$ is analogous. Simply consider the words $a(e^m x e^m)b$ and $a(e^m x e^m)(e^m x e^m)b$ for sufficiently large $m \in \mathbb{N}_+$ and use, again, Lemma 2.11 and the fact that e is idempotent. □

For $t(n) = o(\log n)$, Theorems 2.8 and 2.15 imply that $\mathsf{ACA}[t] \subsetneq \mathsf{LT}$. It turns out we can tighten this bound to $\mathsf{ACA}[O(1)] = \mathsf{SLT}_\cup$, which is a proper subset of LT:

Theorem 2.16. *For every $t(n) = o(\log n)$, $\mathsf{ACA}[t] = \mathsf{ACA}[O(1)]$.*

Essentially, we show that an ACA C with $o(\log n)$ time complexity actually has $O(1)$ time complexity. The key observation is that, if a (long enough) word $w \in L(C)$ is accepted in $o(\log n)$ time, then it must be locally identical (in the sense of Lemma 2.9) to some strictly shorter $w' \in L(C)$. Concretely, this is established by a careful analysis of the De Bruijn graph that corresponds to the neighborhoods of the cells of C when given input w.[4] Since $|w'| < |w|$, by induction we obtain that w is locally identical to some word in $L(C)$ whose length is bounded by a constant n_0. As there are only finitely many such words, it immediately follows that C has $O(1)$ time complexity.

[4] The application of De Bruijn graphs to cellular automata has a long history; the reader is referred to [104, 110] for various examples.

Proof. As stated above, we prove every ACA C with time complexity at most $t(n) = o(\log n)$ actually has $O(1)$ time complexity. Let Q be the state set of C and assume $|Q| \geq 2$. Furthermore, let $n_0 \in \mathbb{N}_+$ be such that $t(n) < (\log_{|Q|} n)/9$ for $n \geq n_0$. Setting $k(n) = 2t(n) + 1$ and assuming $t(n) \geq 1$, we then have

$$|Q|^{3k(n)} \leq |Q|^{9t(n)} < n,$$

which shall be needed later in the proof.

Our key goal is to establish the following: For any word $w \in L$ of length $|w| \geq n_0$, either w is accepted in at most $\max_{n' \leq n_0} t(n')$ steps, or there is a word $w' \in L$ of length $|w'| \leq n_0$ and $\tau > t(|w'|)$ for which $p_{2\tau}(w) = p_{2\tau}(w')$, $I_{2\tau+1}(w) = I_{2\tau+1}(w')$, and $s_{2\tau}(w) = s_{2\tau}(w')$. Assuming this holds, using that $p_r(w) = p_r(w')$ (resp., $I_r(w) = I_r(w')$; resp., $s_r(w) = s_r(w')$) implies $p_{r'}(w) = p_{r'}(w')$ (resp., $I_{r'}(w) = I_{r'}(w')$; resp., $s_{r'}(w) = s_{r'}(w')$) for any $r' \leq r$, we apply Lemma 2.9 and obtain that C must accept w in at most $t(|w'|)$ steps. Since the set of all such w' is finite (and $\max_{n' \leq n_0} t(n')$ is constant), this implies that C has $O(1)$ time complexity.

Now let w be as above, and let C accept w in exactly $\tau = \tau(w) \leq t(|w|)$ steps where $\tau > t(n')$ for every $n' \leq n_0$. We prove the claim by induction on $|w|$. The base case $|w| = n_0$ is trivial, so let $n > n_0$ and assume the claim holds for every word in L of length strictly less than n. Consider the De Bruijn graph G over the words in $|Q|^\kappa$ where $\kappa = 2\tau+1$. To w then corresponds a path P in G which starts at the leftmost infix (of length κ) in w, visits every subsequent one in order of appearance in w, and ends at the rightmost one.

Let G' be the induced subgraph of G containing exactly the nodes visited by P. We then observe the following holds: For every such P and G', there is a path P' in G' with the same starting and ending points as P and that visits every node of G' at least once while having length at most $m^2 \leq |Q|^{2\kappa}$, where m is the number of nodes in G'. To see this, number the nodes of G' from 1 to m according to the order in which they are first visited by P. Then there is a path in G' from i to $i+1$ for every $i \in \{1, \ldots, m-1\}$, and a shortest such path has length at most m. Piecing these paths together along with a last (shortest) path from m to the ending point of P, we obtain a path of length at most m^2 with the purported property.

Finally, to the path P' corresponds a word w' of length $|w'| \leq \kappa + |Q|^{2\kappa} < |Q|^{3\kappa}$ for which, by construction of P' and G', $p_{\kappa-1}(w') = p_{\kappa-1}(w)$, $I_\kappa(w') = I_\kappa(w)$, and $s_{\kappa-1}(w') = s_{\kappa-1}(w')$. Since $\kappa \leq k(|w|)$ and $|Q|^{3k(|w|)} < |w|$, we have $|w'| < |w|$. Then either $|w'| \leq n_0$ already holds, or we can apply the induction hypothesis. The claim follows in either case. □

2.4.3. Relation to Parallel Complexity Classes

In this section, we relate ACA$[t]$ to other classes which characterize parallel computation, namely the SC and (uniform) AC hierarchies. In this context, SC^k is the class of problems decidable by deterministic Turing machines in $O(\log n)^k$ space and polynomial time, whereas AC^k is that decidable by Boolean circuits with polynomial size, $O(\log n)^k$ depth, and gates with unbounded fan-in. The class SC (resp., AC) is the union of all SC^k (resp.,

AC^k) for $k \in \mathbb{N}_0$. Here, we consider only uniform versions of AC; when relevant, we state the respective uniformity condition. Although $\mathsf{SC}^1 = \mathsf{L} \subseteq \mathsf{AC}^1$ is known, it is unclear whether any other containment holds between SC and AC.

One should not expect to include SC or AC in $\mathsf{ACA}[t]$ for any $t(n) = o(n)$. Conceptually speaking, whereas the models of SC and AC are capable of random access to their input, ACAs are inherently local (as evinced by Lemmas 2.9 and 2.11). Explicit counterexamples may be found among the unary languages: For any fixed $m \in \mathbb{N}_+$ and $w_1, w_2 \in \{1\}^+$ with $|w_1|, |w_2| \geq m$, trivially $p_{m-1}(w_1) = p_{m-1}(w_2)$, $I_m(w_1) = I_m(w_2)$, and $s_{m-1}(w_1) = s_{m-1}(w_2)$ hold. Hence, by Lemma 2.9, if an ACA C accepts $w \in \{1\}^+$ in $t(n) = o(n)$ time and $|w|$ is large (e.g., $|w| > 4t(|w|)$), then C accepts any $w' \in \{1\}^+$ with $|w'| \geq |w|$. Thus, extending a result from [105]:

Proposition 2.17. *If $t(n) = o(n)$ and $L \in \mathsf{ACA}[t]$ is a unary language (i.e., $L \subseteq \Sigma^+$ and $|\Sigma| = 1$), then L is either finite or co-finite.*

In light of the above, the rest of this section is concerned with the converse type of inclusion (i.e., of $\mathsf{ACA}[t]$ in the SC or AC hierarchies). For $f, s, t \colon \mathbb{N}_+ \to \mathbb{N}_0$ with $f(n) \leq s(n)$, we say f is *constructible (by a Turing machine) in $s(n)$ space and $t(n)$ time* if there is a Turing machine T which, on input 1^n, outputs $f(n)$ in binary using at most $s(n)$ space and $t(n)$ time. Also, recall a Turing machine can simulate τ steps of a CA with m (active) cells in $O(m)$ space and $O(\tau m)$ time.

Proposition 2.18. *Let C be an ACA with time complexity bounded by $t(n) = o(n)$, $t(n) \geq \log n$, and let t be constructible in $t(n)$ space and $\mathsf{poly}(n)$ time. Then there is a Turing machine which decides $L(C)$ in $O(t(n))$ space and $\mathsf{poly}(n)$ time.*

Proof. We construct a machine T with the purported property. Given an input w, T first determines the input length $|w|$ and computes the value $t(|w|)$ in time bounded by a polynomial $p \colon \mathbb{N}_+ \to \mathbb{N}_0$, thus requiring $O(t(|w|) + \log|w|) = O(t(|w|))$ space and $O(|w| + p(|w|))$ time. The rest of the computation of T takes place in stages, where stage i corresponds to step i of C. The machine T maintains a counter for i, incrementing it after each stage, and rejects whenever $i > t(|w|)$ holds. In stage i, T iterates over every (active) cell $z \in \{0, \ldots, |w| - 1\}$ of C and, by iteratively applying the local transition function of C on $N_i(z)$, determines the state of z in step i of C; since $|N_i(z)| = 2i + 1$, this requires $O(i)$ space and $O(i^2)$ time for each z. If all states computed in the same stage are accept states, then T accepts. Since T runs for at most $t(|w|)$ stages, it uses $O(t(|w|))$ space and $O(|w| \cdot t(|w|)^3 + p(|w|))$ time in total. □

Thus, for polylogarithmic t (where the strict inclusion is due to Proposition 2.17):

Corollary 2.19. *For $k \in \mathbb{N}_+$, $\mathsf{ACA}[O(\log n)^k] \subsetneq \mathsf{SC}^k$.*

Moving on to the AC classes, we employ some notions from descriptive complexity theory (see, e.g., [62] for an introduction). Let $\mathsf{FO}_{\mathsf{L}}[t]$ be the class of languages describable by first-order formulas with numeric relations in L (i.e., logarithmic space) and quantifier block iterations bounded by $t\colon \mathbb{N}_+ \to \mathbb{N}_0$. For C, Q, δ, and Δ as in Definition 2.3, we extend Δ so $\Delta(c)$ is also defined for $c \in Q^{2\tau+1}$ with $\tau \in \mathbb{N}_+$ by

$$\Delta(c)(i) = \delta(c(i), c(i+1), c(i+2))$$

where $i \in \{0, \ldots, 2\tau - 1\}$; in particular, $\Delta(c) \in Q^{2\tau-1}$. Additionally, let $\mathsf{DELTA}_C(c, s)$ be the relation which is true if and only if $c \in Q^{2\tau+1}$ for some $\tau \in \mathbb{N}_0$, $s \in Q$, and $\Delta^\tau(c) = s$ (where Δ^0 is the identity). Note DELTA_C is computable by a Turing machine in $O(\tau)$ space and $O(\tau^2)$ time.

Theorem 2.20. *Let $t\colon \mathbb{N}_+ \to \mathbb{N}_0$ with $t(n) \geq \log n$ be constructible in logarithmic space (and arbitrary time). For any ACA C whose time complexity is bounded by t,*

$$L(C) \in \mathsf{FO}_{\mathsf{L}}\left[O\left(\frac{t(n)}{\log n}\right)\right].$$

In the following proof, we let "$\doteq$" denote the equality relation inside a formula.

Proof. Let Q be the state set of C and let $A \subseteq Q$ be the set of accept states of C; without restriction, we may assume $|Q| \geq 2$. In addition, let w be an input for C and $r = \log_{|Q|}|w|$. Assume we have a predicate $\mathsf{STATE}_{C,w}(z, s, t')$ which is true if and only if cell z of C, when given input w, is in state s after t' steps. Then we may express whether C accepts w or not by the following formula:

$$\varphi_{C,w} = (\exists t' \leq t(|w|))\ (\forall z)\ (\exists s \in A)\ \ \mathsf{STATE}_{C,w}(z, s, t').$$

Note $\varphi_{C,w}$ is true if and only if the initial configuration $c_0 = c_0(w)$ of w is such that an A-final configuration is reached in at most $t' \leq t(|w|)$ steps of C (i.e., C accepts w in (at most) t' steps). In turn, $\mathsf{STATE}_{C,w}$ may be expressed as follows:

$$\mathsf{STATE}_{C,w}(z, s, t') = [B]^{\lfloor t'/r \rfloor}\ (\exists r' \doteq t' \bmod r)\ \ \mathsf{DELTA}_C(c_0|N_{r'}(z), s)$$

where B is a quantifier block (iterated $\lfloor t'/r \rfloor$ times) in which the state s of z is traced back r steps to a former subconfiguration c (of size $2r + 1$) such that the state of every cell z' in c is consistent with the computation of C:

$$B = \left(\exists c \in Q^{2r+1}\ \mathsf{DELTA}_C(c, s)\right)\ (\forall z'\ |z - z'| \leq r)\ (\exists s \doteq c(z' - z + r))\ (\exists z \doteq z')\,.$$

Note all numeric predicates in $\varphi_{C,w}$ are computable in logarithmic space, including DELTA_C. Since $w \in L(C)$ if and only if $w \models \varphi_{C,w}$ holds, the claim follows. □

Since $\mathsf{FO}_{\mathsf{L}}[O(\log n)^k]$ equals L-uniform AC^k [62], by Proposition 2.17 we have:

Corollary 2.21. *For $k \in \mathbb{N}_+$, $\mathsf{ACA}[O(\log n)^k] \subsetneq \mathsf{L}$-uniform AC^{k-1}.*

Because $\mathsf{SC}^1 \not\subseteq \mathsf{AC}^0$ (regardless of non-uniformity) [42], this is an improvement on Corollary 2.19 for at least $k = 1$. Nevertheless, note the usual uniformity condition for AC^0 is not L- but the more restrictive DLOGTIME-uniformity [115], and there is good evidence that these two versions of AC^0 are distinct [18]. Using methods from [12], Corollary 2.21 may be rephrased for AC^0 in terms of $\mathsf{TIME}[\mathrm{polylog}(n)]$- or even $\mathsf{TIME}[(\log n)^2]$-uniformity (since DELTA_C is computable by a Turing machine in quadratic time), but the DLOGTIME-uniformity case remains unclear.

2.5. Decider ACA

So far, we have considered ACAs strictly as language acceptors. As such, their time complexity for inputs not in the target language (i.e., those which are not accepted) is entirely disregarded. In this section, we investigate ACAs as *deciders*, that is, as machines which must also (explicitly) reject invalid inputs. We analyze the case in which these decider ACAs must reject under the same condition as acceptance (i.e., all cells are simultaneously in a final rejecting state):

Definition 2.22 (DACA)**.** A *decider ACA* (*DACA*) is an ACA C which, in addition to its set A of accept states, has a non-empty subset $R \subseteq Q \setminus \{q\}$ of *reject states* that is disjoint from A (i.e., $A \cap R = \emptyset$). Every input $w \in \Sigma^+$ of C must lead to an A- or an R-final configuration (or both). We say C *accepts* w if it leads to an A-final configuration c_A *and none of the configurations prior to* c_A *are* R*-final.* Similarly, C *rejects* w if it leads to an R-final configuration c_R and none of the configurations prior to c_R are A-final. The time complexity of C (with respect to w) is the number of steps elapsed until C reaches an R- or A-final configuration (for the first time). The class $\mathsf{DACA}[t]$ is the DACA analogue of $\mathsf{ACA}[t]$.

In contrast to Definition 2.5, here we must be careful so that the accept and reject results do not overlap (i.e., a word cannot be both accepted and rejected). We opt for interpreting the first (chronologically speaking) of the final configurations as the machine's response. Since the outcome of the computation is then irrelevant regardless of any subsequent configurations (whether they are final or not), this is equivalent to requiring, for instance, that the DACA must halt once a final configuration is reached.

One peculiar consequence of Definition 2.22 is the relation between languages which can be recognized by acceptor ACAs and DACAs (i.e., the classes $\mathsf{ACA}[t]$ and $\mathsf{DACA}[t]$). As it turns out, the situation is quite different from what is usually expected of restricting an acceptor model to a decider one, that is, that deciders yield a (possibly strictly) more restricted class of machines. In fact, one can show $\mathsf{DACA}[t] \not\subseteq \mathsf{ACA}[t]$ holds for $t(n) = o(n)$ since $\mathsf{SOMEONE} \notin \mathsf{ACA}[o(n)]$ (see discussion after Lemma 2.9) but surprisingly $\mathsf{SOMEONE} \in \mathsf{DACA}[O(1)]$. For example, the local transition function δ of the DACA

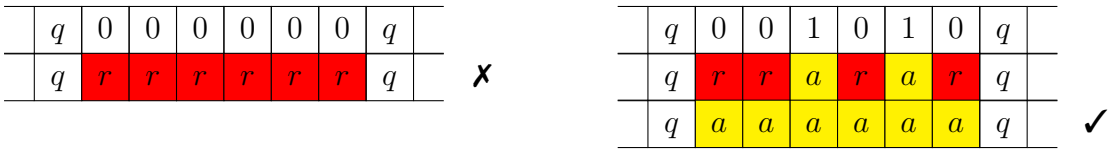


Figure 2.2.: Computation of a DACA C which decides SOMEONE. The inputs words are $000000 \in L(C)$ and $001010 \notin L(C)$, respectively.

can be chosen as $\delta(z_1, 0, z_2) = r$ and $\delta(z_1, z, z_2) = a$ for $z \in \{1, a, r\}$, where z_1 and z_2 are arbitrary states, and a and r are the (only) accept and reject states, respectively; see Figure 2.2. Choosing the same δ for an (acceptor) ACA does *not* yield an ACA for SOMEONE since then all words of the form 0^+ are accepted in the second step (as they are not rejected in the first one). We stress this rather counterintuitive phenomenon occurs only in the case of sublinear time (as $\mathsf{ACA}[t] = \mathsf{CA}[t] = \mathsf{DACA}[t]$ for $t(n) = \Omega(n)$).

Similar to (acceptor) ACAs (Lemma 2.9), sublinear-time DACAs operate locally:

Lemma 2.23. *Let C be a DACA and let $w \in \{0,1\}^+$ be a word which C decides in exactly $\tau = \tau(w)$ steps. Then, for every word $w' \in \{0,1\}^+$ with $p_{2\tau}(w) = p_{2\tau}(w')$, $I_{2\tau+1}(w') = I_{2\tau+1}(w)$, and $s_{2\tau}(w) = s_{2\tau}(w')$, C decides w' in exactly τ steps, and $w \in L(C)$ holds if and only if $w' \in L(C)$.*

Proof. Let A and R be the set of accept and reject states of C, respectively, and let c_0 and c'_0 denote the initial configurations for w and w', respectively. Given $w \in L(C)$ (resp., $w \notin L(C)$), it suffices to prove that, on input w', the following holds:

1. $c'_\tau = \Delta^\tau(c'_0)$ is A-final (resp., R-final).
2. $c'_{\tau'} = \Delta^{\tau'}(c'_0)$ is neither A- nor R-final for $\tau' < \tau$.

The proof of (1) is essentially the same as that of Lemma 2.9. For (2), it suffices to prove that, in every such $c'_{\tau'}$, there is a cell z_1 which is not accepting as well as a cell z_2 which is not rejecting. Since the trace of C for w is such that only the last configuration c_τ can be A- or R-final, there is z_1 which is not accepting as well as z_2 which is not rejecting in $\Delta^{\tau'}(c_0)$. An argument as in the proof of Lemma 2.9 yields this is also the case for $c'_{\tau'}$. □

One might be tempted to relax the requirements above to $I_{2\tau+1}(w') \subseteq I_{2\tau+1}(w)$ (as in Lemma 2.9). We stress, however, the equality $I_{2\tau+1}(w) = I_{2\tau+1}(w')$ is crucial for the existence of z_1 and z_2 in the proof. Indeed, otherwise it might be the case that C takes strictly less than τ steps to decide w' and, hence, $w \in L(C)$ may not be equivalent to $w' \in L(C)$ (despite c'_τ being A- or R-final).

We note that, in addition to Lemmas 2.9 and 2.11, the results from Section 2.4 are extendable to decider ACAs; however, a more systematic treatment is left as a topic for future work. The remainder of this section is concerned with characterizing $\mathsf{DACA}[O(1)]$ computation (as a parallel to Theorem 2.6) as well as establishing the time threshold for DACAs to decide languages other than those in $\mathsf{DACA}[O(1)]$ (as Theorem 2.16 and the result $\mathsf{BIN} \in \mathsf{ACA}[O(\log n)]$ do for acceptor ACAs).

2.5.1. The Constant-Time Case

We note that, for any DACA C, swapping the accept and reject states yields a DACA with the same time complexity and which decides the complement of $L(C)$. Hence, in contrast to ACAs (see discussion following Lemma 2.9):

Proposition 2.24. *For any $t\colon \mathbb{N}_+ \to \mathbb{N}_0$, $\mathsf{DACA}[t]$ is closed under complement.*

Using this, we can prove the following, which characterizes constant-time DACA computation as a parallel to Theorem 2.6:

Theorem 2.25. $\mathsf{DACA}[O(1)] = \mathsf{LT}$.

Hence, we obtain the rather surprising inclusion $\mathsf{ACA}[O(1)] \subsetneq \mathsf{DACA}[O(1)]$; that is, for constant time, DACAs constitute a *strictly more* powerful model than their acceptor counterparts.

Proof. The inclusion $\mathsf{DACA}[O(1)] \subseteq \mathsf{LT}$ is easily obtained by using the locally testable property (see Definition 2.2) together with Lemma 2.23. Conversely, we obtain $\mathsf{LT} \subseteq \mathsf{DACA}[O(1)]$ by proving the following:

1. $\mathsf{ACA}[O(1)] \subseteq \mathsf{DACA}[O(1)]$.
2. $\mathsf{DACA}[O(1)]$ is closed under union.

Since $\mathsf{DACA}[O(1)]$ is also closed under complement (by Proposition 2.24 above), it is then closed under intersection as well. Using that LT is the Boolean closure of $\mathsf{SLT} \subseteq \mathsf{ACA}[O(1)]$ (see Theorem 2.6) completes the proof.

For the first claim above, let C be an ACA with time complexity bounded by $t \in \mathbb{N}_0$. Since C must accept any input w in at most t steps, if $t+1$ steps elapse without C accepting, then necessarily $w \notin L(C)$. Hence, since t is constant, C can be transformed into a DACA C' with $L(C) = L(C')$ by having all cells unconditionally become rejecting in step $t+1$.

To prove the second claim, let C_1 and C_2 be DACAs with time complexity bounded by constants $t_1, t_2 \in \mathbb{N}_0$, respectively. We construct a DACA C for $L(C_1) \cup L(C_2)$ as follows: C saves its input w in a second register and then simulates C_1 on w for t_1 steps, accepting if C_1 does. (Note C does *not* yet reject, even if C_1 does so.) Following that, if it has not yet accepted, C simulates C_2 for t_2 steps and accepts if C_2 does. Finally, if $t_1 + t_2 + 1$ steps elapse and C has not accepted, it rejects unconditionally. Hence, C accepts $w \in L(C_1)$ in at most t_1 steps, $w \in L(C_2)$ in at most $t_1 + t_2$ steps, and rejects any $w \notin L(C_1) \cup L(C_2)$ in (exactly) $t_1 + t_2 + 1$ steps. □

2.5.2. Beyond Constant Time

Theorem 2.16 establishes a logarithmic time threshold for (acceptor) ACAs to recognize languages not in ACA$[O(1)]$. We now turn to obtaining a similar result for DACAs. As it turns out, in this case the bound is considerably larger:

Theorem 2.26. *For any* $t(n) = o(\sqrt{n})$, DACA$[t]$ = DACA$[O(1)]$.

One immediate implication is that DACA$[t]$ and ACA$[t]$ are *incomparable* if $t(n) = o(\sqrt{n})$ and $t(n) = \omega(1)$ (since, e.g., BIN $\in$ ACA$[O(\log n)]$; see Section 2.3). The proof idea is that any DACA whose time complexity is not constant admits an infinite sequence of words with increasing time complexity; however, the time complexity of each such word can be traced back to a critical set of cells which prevent the automaton from either accepting or rejecting. By contracting the words while keeping the extended neighborhoods of these cells intact, we obtain a new infinite sequence of words which the DACA necessarily takes $\Omega(\sqrt{n})$ time to decide:

Proof. Let C be a DACA with time complexity bounded by t and assume $t(n) = \omega(1)$. We show $t(n) = \Omega(\sqrt{n})$. Since $t(n) = \omega(1)$, for every $i \in \mathbb{N}_0$ there is a w_i such that C takes strictly more than i steps to decide w_i. In particular, when C receives w_i as input, there are cells x^i_j and y^i_j for $j \in \{0, \dots, i\}$ such that x^i_j (resp., y^i_j) is not accepting (resp., not rejecting) in step j. Let J_i be the set of all $z \in \{0, \dots, |w_i| - 1\}$ for which

$$\min\left\{\left|z - x^i_j\right|, \left|z - y^i_j\right|\right\} \leq j,$$

that is, $z \in N_j(x^i_j) \cup N_j(y^i_j)$ for some j. Consider the restriction w'_i of w_i to the symbols having index in J_i, that is, $w'_i(k) = w_i(j_k)$ for $J_i = \{j_0, \dots, j_{m-1}\}$ and $j_0 < \cdots < j_{m-1}$, and notice w'_i has the same property as w_i (i.e., C takes strictly more than i steps to decide w_i). Since $|w'_i| = |J_i| \leq 2(i+1)^2$, C has $\Omega(\sqrt{n})$ time complexity on the (infinite) set $\{w'_i \mid i \in \mathbb{N}_0\}$. □

The language IDMAT from Section 2.3 now turns out to be useful in showing the bound from Theorem 2.26 is optimal:

Proposition 2.27. IDMAT $\in$ DACA$[O(\sqrt{n})]$.

We already have IDMAT $\in$ ACA$[O(\sqrt{n})]$ (see Section 2.3) and most ideas from the ACA construction can be adapted to a DACA. The non-trivial part, however, is ensuring the DACA also rejects every $w \notin$ IDMAT in $O(\sqrt{|w|})$ time. In particular, in such strings the # delimiters may be an arbitrary number of cells apart or even absent altogether; hence, naively comparing every pair of blocks is not an option. Rather, we additionally check the existence of a particular set of substrings of increasing length and which must be present if the input is in IDMAT. Every $O(1)$ steps the existence of a different substring is verified; the result is that the input length must be at least quadratic in the length of the last substring tested (and the input is timely rejected if it does not contain any one of the required substrings).

Proof. We construct a DACA A which, given an input w, decides whether $w \in \mathsf{IDMAT}$ holds or not in $O(\sqrt{|w|})$ time.

As a warm-up, first consider the case in which every block B of w has the same length $b = O(\sqrt{|w|})$ and that every neighboring pair of blocks is separated by a single #. The leftmost cell in B creates a special marker symbol m. During this first procedure, every cell which does not contain such an m is rejecting. At each step, if m is on a cell containing a 0 or it determines the string it has read so far does not match the regular expression 0^*10^*, then it marks the cell as rejecting; otherwise, it does nothing (i.e., the cell remains not rejecting). The signal m propagates itself to the right with speed 1, the result being that A does not reject in $i > 0$ steps if and only if for every $p \in \{1, 01, \ldots, 0^{i-1}1\}$ there is a block in w for which p is a prefix. It follows that $|w| \geq i(i+1)/2$ and, in particular, if b steps have elapsed, then $|w| > b^2/2$. Thus, if A rejects a word during this procedure, then it does so in $O(\sqrt{|w|})$ time. Once m encounters #, it triggers a block comparison procedure as in the ACA A' which accepts IDMAT. This requires $O(b)$ time. If a violation is detected, B becomes rejecting and maintains that state. Finally, if a number of steps have elapsed such that A' would already have accepted (which by construction of A' can be determined in $O(b)$ time as a function of b), B becomes rejecting and maintains that result, even if it contains cells which had been previously marked as accepting. Thus, A accepts if and only if A' does (and rejects otherwise).

For the general case in which the block lengths vary, we let the two procedures run in parallel, with the cells of A switching between the two back and forth. More precisely, the computation of A is subdivided into rounds, with each round consisting of two phases, both taking constant time each; the time complexity of A, then, is directly proportional to the number of rounds elapsed until a final configuration is reached. The two phases correspond to the two aforementioned procedures: Phase one (P_1) checks that the blocks satisfy the regular expression 0^*10^* as well as ensures the presence of the 0^*1 prefixes, and phase two (P_2) checks the blocks are of the same length and have valid contents. P_1 advances its procedure one step at a time, while P_2 advances two steps (as in the ACA construction; see Section 2.3). In addition, we separate the two so as to not interfere with each other; namely, if a cell is accepting (resp., rejecting) in phase P_i, then it is not necessarily so in P_j (where $i \neq j$); rather, it is only accepting or rejecting in P_j if the procedure corresponding to P_j requires it to be so. If $w \in \mathsf{IDMAT}$, the two phases end simultaneously after $3b \in O(\sqrt{|w|})$ steps, and A accepts. Conversely, if A rejects, then it also does so in at most $3b' \in O(\sqrt{|w|})$ steps where $b' \in \mathbb{N}_+$ is maximal such that $\#1, \#01, \ldots, \#0^{b'-1}1$ are all substrings of $w \notin \mathsf{IDMAT}$. □

2.6. Conclusion and Open Problems

Following the definition of ACAs in Section 2.2, Section 2.3 reviewed existing results on ACA$[t]$ for sublinear t (i.e., $t(n) = o(n)$); we also observed that sublinear-time ACAs operate in an inherently local manner (Lemmas 2.9 and 2.11). In Section 2.4, we proved a

time hierarchy theorem (Theorem 2.13), narrowed down the languages in $\mathsf{ACA}[t] \cap \mathsf{REG}$ (Theorem 2.15), improved Theorem 2.8 to $\mathsf{ACA}[o(\log n)] = \mathsf{ACA}[O(1)]$ (Theorem 2.16), and, finally, obtained (strict) inclusions in the parallel computation classes SC and AC (Corollaries 2.19 and 2.21, respectively). The existence of a hierarchy theorem for ACAs is of interest because obtaining an equivalent result for NC and AC is an open problem in computational complexity theory. Also of note is that the proof of Theorem 2.13 does not rely on diagonalization (the prevalent technique for most computational models) but, rather, on a quintessential property of sublinear-time ACA computation (i.e., locality as in the sense of Lemma 2.9).

In Section 2.5 we considered DACAs, which are a plausible variant of ACAs as language deciders (as opposed to simply acceptors). The respective constant-time class is LT (Theorem 2.25), which surprisingly is a (strict) superset of $\mathsf{ACA}[O(1)] = \mathsf{SLT}_{\cup}$. Meanwhile, $\Omega(\sqrt{n})$ is the time complexity threshold for deciding languages other than those in LT (Theorem 2.26 and Proposition 2.27).

As for future work, the primary concern is extending the results of Section 2.4 to DACAs. $\mathsf{DACA}[O(1)] = \mathsf{LT}$ is closed under union and intersection and we saw that $\mathsf{DACA}[t]$ is closed under complement for any $t(n) = o(n)$; a further question would be whether $\mathsf{DACA}[t]$ is also closed under union and intersection. Finally, we have $\mathsf{ACA}[O(1)] \subsetneq \mathsf{DACA}[O(1)]$, $\mathsf{ACA}[O(n)] = \mathsf{CA}[O(n)] = \mathsf{DACA}[O(n)]$, and that $\mathsf{ACA}[t]$ and $\mathsf{DACA}[t]$ are incomparable if $t(n) = o(\sqrt{n})$ and $t(n) = \omega(1)$; it remains open what the relation between the two classes is for t such that $t(n) = \Omega(\sqrt{n})$ and $t(n) = o(n)$.

3. Lower Bounds and Hardness Magnification for Sublinear-Time Shrinking Cellular Automata

Published version: [82]

Abstract

The minimum circuit size problem (MCSP) is a string compression problem with a parameter s in which, given the truth table of a Boolean function over inputs of length n, one must answer whether it can be computed by a Boolean circuit of size at most $s(n) \geq n$. Recently, McKay, Murray, and Williams (STOC, 2019) proved a hardness magnification result for MCSP involving (one-pass) streaming algorithms: For any reasonable s, if there is no $\mathrm{poly}(s(n))$-space streaming algorithm with $\mathrm{poly}(s(n))$ update time for $\mathsf{MCSP}[s]$, then $\mathsf{P} \neq \mathsf{NP}$. We prove an analogous result for the (provably) strictly less capable model of shrinking cellular automata (SCAs), which are cellular automata whose cells can spontaneously delete themselves. We show every language accepted by an SCA can also be accepted by a streaming algorithm of similar complexity, and we identify two different aspects in which SCAs are more restricted than streaming algorithms. We also show there is a language which cannot be accepted by any SCA in $o(n/\log n)$ time, even though it admits an $O(\log n)$-space streaming algorithm with $O(\log n)$ update time.

3.1. Introduction

The ongoing quest for lower bounds in complexity theory has been an arduous but by no means unfruitful one. Recent developments have brought to light a phenomenon dubbed *hardness magnification* [20, 21, 25, 75, 92, 93], giving several examples of natural problems for which even slightly non-trivial lower bounds are as hard to prove as major complexity class separations such as $\mathsf{P} \neq \mathsf{NP}$. Among these, the preeminent example appears to be the *minimum circuit size problem*:

Definition 3.1 (MCSP)**.** For a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, let $\mathrm{tt}(f)$ denote the truth table representation of f (as a binary string in $\{0,1\}^+$ of length $|\mathrm{tt}(f)| = 2^n$). For $s\colon \mathbb{N}_+ \to \mathbb{N}_+$, the *minimum circuit size problem* $\mathsf{MCSP}[s]$ is the problem where, given such a truth table $\mathrm{tt}(f)$, one must answer whether there is a Boolean circuit C on inputs

of length n and size at most $s(n)$ that computes f, that is, $C(x) = f(x)$ for every input $x \in \{0,1\}^n$.

It is a well-known fact that there is a constant $K > 0$ such that, for any function f on n variables as above, there is a circuit of size at most $K2^n/n$ that computes f; hence, $\mathsf{MCSP}[s]$ is only non-trivial for $s(n) < K2^n/n$. Furthermore, $\mathsf{MCSP}[s] \in \mathsf{NP}$ for any constructible s and, since every circuit of size at most $s(n)$ can be described by a binary string of $O(s(n)\log s(n))$ length, if $2^{O(s(n)\log s(n))} \leq \mathrm{poly}(2^n)$ (e.g., $s(n) = O(n/\log n)$), by enumerating all possibilities we have $\mathsf{MCSP}[s] \in \mathsf{P}$. (Of course, such a bound is hardly useful since $s(n) = O(n/\log n)$ implies the circuit is degenerate and can only read a strict subset of its inputs.) For large enough $s(n) < K2^n/n$ (e.g., $s(n) \geq n$), it is unclear whether $\mathsf{MCSP}[s]$ is NP-complete (under polynomial-time many-one reductions); see also [64, 90]. Still, we remark there has been some recent progress regarding NP-completeness under *randomized* many-one reductions for *certain variants* of MCSP [61].

Oliveira and Santhanam [93] and Oliveira, Pich, and Santhanam [92] recently analyzed hardness magnification in the average-case as well as in the worst-case approximation (i.e., gap) settings of MCSP for various (uniform and non-uniform) computational models. Meanwhile, McKay, Murray, and Williams [75] showed similar results hold in the standard (i.e., exact or gapless) worst-case setting and proved the following magnification result for (single-pass) *streaming algorithms* (see Definition 3.6), which is a very restricted uniform model; indeed, as mentioned in [75], even string equality (i.e., the problem of recognizing $\{ww \mid w \in \{0,1\}^+\}$) cannot be solved by streaming algorithms (with limited space).

Theorem 3.2 ([75])**.** *Let $s\colon \mathbb{N}_+ \to \mathbb{N}_+$ be time constructible and $s(n) \geq n$. If there is no $\mathrm{poly}(s(n))$-space streaming algorithm with $\mathrm{poly}(s(n))$ update time for (the search version of) $\mathsf{MCSP}[s]$, then $\mathsf{P} \neq \mathsf{NP}$.*

In this chapter, we present the following hardness magnification result for a (uniform) computational model which is provably *even more restricted* than streaming algorithms: *shrinking cellular automata* (SCAs). Here, Block_b refers to a slightly modified presentation of $\mathsf{MCSP}[s]$ that is only needed due to certain limitations of the model (see further discussion as well as Section 3.3.1).

Theorem 3.3. *For a certain $m \in \mathrm{poly}(s(n))$, if $\mathsf{Block}_b(\mathsf{MCSP}[s]) \notin \mathsf{SCA}[nf(m)]$ for every $f \in \mathrm{poly}(m)$ and $b = O(f)$, then $\mathsf{P} \neq \mathsf{NP}$.*

Furthermore, we show every language accepted by a sublinear-time SCA can also be accepted by a streaming algorithm of comparable complexity:

Theorem 3.4. *Let $t\colon \mathbb{N}_+ \to \mathbb{N}_+$ be computable by an $O(t)$-space random access machine (as in Definition 3.6) in $O(t\log t)$ time. Then, if $L \in \mathsf{SCA}[t]$, there is an $O(t)$-space streaming algorithm for L with $O(t\log t)$ update and $O(t^2\log t)$ reporting time.*

Finally, we identify and prove *two distinct limitations* of SCAs compared to streaming algorithms (under sublinear-time constraints):

1. They are insensitive to the length of long unary substrings in their input (Lemma 3.17), which means (standard versions of) fundamental problems such as parity, modulo, majority, and threshold cannot be solved in sublinear time (Proposition 3.18 and Corollary 3.20).
2. Only a limited amount of information can be transferred between cells which are far apart (in the sense of one-way communication complexity; see Lemma 3.23).

Both limitations are inherited from the underlying model of cellular automata. The first can be avoided by presenting the input in a special format (the previously mentioned Block_n) that is efficiently verifiable by SCAs, which we motivate and adopt as part of the model (see the discussion below). The second is more dramatic and results in lower bounds even for languages presented in this format:

Theorem 3.5. *There is a language L_1 for which $\mathsf{Block}_n(L_1) \notin \mathsf{SCA}[o(N/\log N)]$ (N being the instance length) can be accepted by an $O(\log N)$-space streaming algorithm with $\tilde{O}(\log N)$ update time.*

From the above, it follows that any proof of $\mathsf{P} \neq \mathsf{NP}$ based on a lower bound for solving $\mathsf{MCSP}[s]$ with streaming algorithms and Theorem 3.2 must implicitly contain a proof of a lower bound for solving $\mathsf{MCSP}[s]$ with SCAs. From a more "optimistic" perspective (with an eventual proof of $\mathsf{P} \neq \mathsf{NP}$ in mind), although not as widely studied as streaming algorithms, SCAs are thus at least as good as a "target" for proving lower bounds against and, in fact, should be an easier one if we are able to exploit their aforementioned limitations. Refer to Section 3.6 for further discussion on this, where we take into account a recently proposed barrier [20] to existing techniques and which also applies to our proof of Theorem 3.5.

From the perspective of cellular automata theory, our results further knowledge in sublinear-time cellular automata models, a topic seemingly neglected by the community at large (as pointed out in Chapter 2). Although this is certainly not the first result in which complexity-theoretical results for cellular automata and their variants have consequences for classical models (see, e.g., [70, 95] for results in this sense), to the best of our knowledge said results address only *necessary* conditions for separating classical complexity classes. Hence, our result is also novel in providing an implication in the other direction, that is, a *sufficient* condition for said separations based on lower bounds for cellular automata models.

3.1.1. The Model

(One-dimensional) cellular automata (CAs) are a parallel computational model composed of identical *cells* arranged in an array. Each cell operates as a deterministic finite automaton

(DFA) that is connected with its left and right neighbors and operates according to the same local rule. In classical CAs, the cell structure is immutable; *shrinking* CAs relax the model in that regard by allowing cells to spontaneously vanish (with their contents being irrecoverably lost). The array structure is conserved by reconnecting every cell with deleted neighbors to the nearest non-deleted ones in either direction.

SCAs were introduced by Rosenfeld, Wu, and Dubitzki in 1983 [100], but it was not until recent years that the model received greater attention by the CA community [71, 85]. SCAs are a natural and robust model of parallel computation which, unlike classical CAs, admit (non-trivial) sublinear-time computations.

We give a brief intuition as to how shrinking augments the classical CA model in a significant way. Intuitively speaking, any two cells in a CA can only communicate by signals, which necessarily requires time proportional to the distance between them. Assuming the entire input is relevant towards acceptance, this imposes a linear lower bound on the time complexity of the CA. In SCAs, however, this distance can be shortened as the computation evolves, thus rendering acceptance in sublinear time possible. As a matter of fact, the more cells are deleted, the faster distant cells can communicate and the computation can evolve. This results in a trade-off between space (i.e., cells containing information) and time (i.e., amount of cells deleted).

Comparison with Related Models. Unlike other parallel models such as random access machines, SCAs are *incapable of random access* to their input. In a similar sense, SCAs are constrained by the *distance* between cells, which is an aspect usually disregarded in circuits and related models except perhaps for VLSI complexity [19, 112], for instance. In contrast to VLSI circuits, however, in SCAs distance is a fluid aspect, changing dynamically as the computation evolves. Also of note is that SCAs are a *local* computational model in a quite literal sense of locality that is coupled with the above concept of distance (instead of more abstract notions such as that from [116], for example).

These limitations hold not only for SCAs but also for standard CAs. Nevertheless, SCAs are more powerful than other CA models capable of sublinear-time computation such as ACAs (see Chapter 2 as well as [60]), which are CAs with their acceptance behavior such that the CA accepts if and only if all cells simultaneously accept. This is because SCAs can *efficiently aggregate results* computed in parallel (by combining them using some efficiently computable function); in ACAs any such form of aggregation is fairly limited as the underlying cell structure is static.

Block Words. As mentioned above, there is an input format which allows us to circumvent the first of the limitations of SCAs compared to streaming algorithms and which is essential in order to obtain a more serious computational model. In this format, the input is subdivided into *blocks* of the same size and which are separated by delimiters and numbered in ascending order from left to right. Words with this structure are dubbed *block words* accordingly, and a set of such words is a *block language*. There is a natural presentation of

any (ordinary) word as a block word (by mapping every symbol to its own block), which means there is a block language version to any (ordinary) language. (See Section 3.3.1.)

The concept of block words seems to arise naturally in the context of sublinear-time (both shrinking and standard) CAs (as in Chapter 2 or also [60]). The syntax of block words is very efficiently verifiable (more precisely, in time linear in the block length) by a CA (without need of shrinking). In addition, the translation of a language to its block version (and its inverse) is a very simple map; one may frame it, for instance, as an AC^0 reduction. Hence, the difference between a language and its block version is solely in presentation.

Block words coupled with CAs form a computational paradigm that appears to be substantially diverse from linear- and real-time CA computation (see Chapter 2 for examples). Often we shall describe operations on a block (rather than on a cell) level and, by making use of block numbering, two blocks with distinct numbers may operate differently even though their contents are the same; this would be impossible at a cell level due to the locality of CA rules. In combination with shrinking, certain block languages admit merging groups of blocks in parallel; this gives rise to a form of reduction we call *blockwise reductions* and which we employ in a manner akin to downward self-reducibility as in [3].

An additional technicality which arises is that the number of cells in a block is fixed at the start of the computation; this means a block cannot "allocate extra space" (beyond a constant multiple of the block length). This is the same limitation as that of linear bounded automata (LBAs) compared to Turing machines with unbounded space, for example. We cope with this limitation by increasing the block length in the problem instances as needed, that is, by padding each block so that enough space is available from the outset.[1] This is still in line with the considerations above; for instance, the resulting language is still AC^0 reducible to the original one (and vice-versa).

3.1.2. Techniques

We give a broad overview of the proof ideas behind our results.

Theorem 3.3 is a direct corollary of Theorem 3.24, proven is Section 3.5. The proof closely follows [75] (see the discussion in Section 3.5 for a comparison) and, as mentioned above, bases on a scheme similar to self-reducibility as in [3].

The lower bounds in Section 3.3.2 are established using Lemma 3.17, which is a generic technical limitation of sublinear-time models based on CAs (the first of the two aforementioned limitations of SCAs with respect to streaming algorithms) and which we also show to hold for SCAs.

[1] An alternative solution is allowing the CA to "expand" by dynamically creating new cells between existing ones; however, this may result in a computational model which is dramatically more powerful than standard CAs [80, 85].

One of the main technical highlights is the proof of Theorem 3.4, where we give a streaming algorithm to simulate an SCA with limited space. Our general approach bases on dynamic programming and is able to cope with the unpredictability of when, which, or even how many cells are deleted during the simulation. The space efficiency is achieved by keeping track of only as much information as needed as to determine the state of the SCA's decision cell step for step.

A second technical contribution is the application of *one-way* communication complexity to obtain lower bounds for SCAs, which yields Theorem 3.5. Essentially, we split the input in some position i of our choice (which may even be non-uniformly dependent on the input length) and have A be given as input the symbols preceding i while B is given the rest, where A and B are (non-uniform) algorithms with unbounded computational resources. We show that, in this setting, A can determine the state of the SCA's decision cell with only $O(1)$ information from B for every step of the SCA. Thus, an SCA with time complexity t for a language L yields a protocol with $O(t)$ one-way communication complexity for the above problem. Applying this in the contrapositive, Theorem 3.5 then follows from the existence of a language L_1 (in some contexts referred to as the indexing or memory access problem) that has nearly linear one-way communication complexity despite admitting an efficient streaming algorithm.

3.1.3. Organization

The rest of the chapter is organized as follows: Section 3.2 presents the basic definitions. In Section 3.3 we introduce block words and related concepts and discuss the aforementioned limitations of sublinear-time SCAs. Following that, in Section 3.4 we address the proof of Theorem 3.4 and in Section 3.5 that of Theorem 3.3. Finally, Section 3.6 concludes the chapter.

3.2. Preliminaries

We denote the set of integers by $\mathbb{Z}$, that of positive integers by $\mathbb{N}_+$, and $\mathbb{N}_+ \cup \{0\}$ by $\mathbb{N}_0$. For $a, b \in \mathbb{N}_0$,

$$[a, b] = \{x \in \mathbb{N}_0 \mid a \leq x \leq b\}.$$

For sets A and B, B^A is the set of functions $A \to B$. Unless otherwise noted, the base of the logarithm is two.

We assume the reader is familiar with cellular automata as well as with the fundamentals of computational complexity theory (see, e.g., standard references [5, 30, 50]). Words are indexed starting with index zero. For a finite, non-empty set Σ, Σ^* denotes the set of words over Σ, and Σ^+ the set $\Sigma^* \setminus \{\varepsilon\}$. For $w \in \Sigma^*$, we write $w(i)$ for the i-th symbol of w (and, in general, w_i stands for another word altogether, *not* the i-th symbol of w). For $a, b \in \mathbb{N}_0$, $w[a, b]$ is the subword $w(a)w(a+1)\cdots w(b-1)w(b)$ of w (where $w[a, b] = \varepsilon$ for $a > b$). $|w|_a$ is the number of occurrences of $a \in \Sigma$ in w. $\mathrm{bin}_n(x)$ stands for the

binary representation of $x \in \mathbb{N}_0$, $x < 2^n$, of length $n \in \mathbb{N}_+$ (padded with leading zeros). $\mathrm{poly}(n)$ is the class of functions polynomial in $n \in \mathbb{N}_0$. REG denotes the class of regular languages, and $\mathsf{TISP}[t, s]$ (resp., $\mathsf{TIME}[t]$) that of problems decidable by a Turing machine (with one tape and one read-write head) in $O(t)$ time and $O(s)$ space (resp., unbounded space). Without restriction, we assume the empty word ε is not a member of any of the languages considered.

An *ω-word* is a map $\mathbb{N}_0 \to \Sigma$, and a *$\omega\omega$-word* is a map $\mathbb{Z} \to \Sigma$. We write $\Sigma^\omega = \Sigma^{\mathbb{N}_0}$ for the set of ω-words over Σ. For $x \in \Sigma$, x^ω denotes the (unique) ω-word with $x^\omega(i) = x$ for every $i \in \mathbb{N}_0$. To each $\omega\omega$-word w corresponds a unique pair (w_-, w_+) of ω-words $w_-, w_+ \in \Sigma^\omega$ with $w_+(i) = w(i)$ for $i \geq 0$ and $w_-(i) = w(-i-1)$ for $i < 0$. (Partial) ω-word homomorphisms are extendable to (partial) *$\omega\omega$-word homomorphisms* as follows: Let $f\colon \Sigma^\omega \to \Sigma^\omega$ be an ω-word homomorphism; then there is a unique $f_{\omega\omega}\colon \Sigma^{\mathbb{Z}} \to \Sigma^{\mathbb{Z}}$ such that, for every $w \in \Sigma^{\mathbb{Z}}$, $w' = f_{\omega\omega}(w)$ is the $\omega\omega$-word with $w'_+ = f(w_+)$ and $w'_- = f(w_-)$.

For a circuit C, $|C|$ denotes the *size* of C, that is, the total number of gates in C. It is well-known that any Boolean circuit C can be described by a binary string of $O(|C| \log |C|)$ length.

Definition 3.6 (Streaming algorithm)**.** Let $s, u, r\colon \mathbb{N}_+ \to \mathbb{N}_+$ be functions. An *s-space streaming algorithm* A is a random access machine which, on input w, works in $O(s(|w|))$ space and, on every step, can either perform an operation on a constant number of bits in memory or read the next symbol of w. A has *u update time* if, for every w, the number of operations it performs between reading $w(i)$ and $w(i+1)$ is at most $u(|w|)$. A has *r reporting time* if it performs at most $r(|w|)$ operations after having read $w(|w|-1)$ (until it terminates).

Our interest lies in s-space streaming algorithms that, for an input w, have $\mathrm{poly}(s(|w|))$ update and reporting time for sublinear s (i.e., $s(|w|) = o(|w|)$).

3.2.1. Cellular Automata

We consider only CAs with the standard neighborhood. The symbols of an input w are provided from left to right in the cells 0 to $|w|-1$ and are surrounded by inactive cells, which conserve their state during the entire computation (i.e., the CA is bounded). Acceptance is signaled by cell zero (i.e., the leftmost input cell).

Definition 3.7 (Cellular automaton)**.** A *cellular automaton* (*CA*) C is a tuple $(Q, \delta, \Sigma, q, A)$ where:

- Q is a non-empty and finite set of *states*.
- $\delta\colon Q^3 \to Q$ is the *local transition function*.
- $\Sigma \subsetneq Q$ is the *input alphabet* of C.

- $q \in Q \setminus \Sigma$ is the *inactive state*, that is, $\delta(q_1, q, q_2) = q$ for every $q_1, q_2 \in Q$.
- $A \subseteq Q \setminus \{q\}$ is the set of *accepting states* of C.

A cell which is not in the inactive state is said to be *active*. The elements of $Q^{\mathbb{Z}}$ are the (*global*) *configurations* of C. δ induces the *global transition function* $\Delta\colon Q^{\mathbb{Z}} \to Q^{\mathbb{Z}}$ of C by

$$\Delta(c)(i) = \delta(c(i-1), c(i), c(i+1))$$

for every cell $i \in \mathbb{Z}$ and configuration $c \in Q^{\mathbb{Z}}$.

C *accepts* an input $w \in \Sigma^+$ if cell zero is eventually in an accepting state, that is, there is $t \in \mathbb{N}_0$ such that $(\Delta^t(c_0))(0) \in A$, where $c_0 = c_0(w)$ is the *initial configuration* (for w): $c_0(i) = w(i)$ for $i \in [0, |w| - 1]$, and $c_0(i) = q$ otherwise. For a minimal such t, we say C accepts w with *time complexity* t. $L(A) \subseteq \Sigma^+$ denotes the set of words accepted by C. For $t\colon \mathbb{N}_+ \to \mathbb{N}_0$, $\mathsf{CA}[t]$ is the class of languages accepted by CAs with time complexity $O(t(n))$, n being the input length.

For convenience, we extend Δ in the obvious manner (i.e., as a map induced by δ) so it is also defined for every (finite) word $w \in Q^*$. For $|w| \leq 2$, we set $\Delta(w) = \varepsilon$; for longer words, $|\Delta(w)| = |w| - 2$ holds.

We state some remarks concerning the classes $\mathsf{CA}[t]$: $\mathsf{CA}[\mathsf{poly}] = \mathsf{TISP}[\mathsf{poly}, n]$ (i.e., the class of polynomial-time LBAs), and $\mathsf{CA}[t] = \mathsf{CA}[1] \subsetneq \mathsf{REG}$ for every sublinear t. Furthermore, $\mathsf{CA}[t] \subseteq \mathsf{TISP}[t^2, n]$ (where $t^2(n) = (t(n))^2$) and $\mathsf{TISP}[t, n] \subseteq \mathsf{CA}[t]$.

Definition 3.8 (Shrinking CA)**.** A *shrinking CA* (*SCA*) S is a CA with a *delete state* $\otimes \in Q \setminus (\Sigma \cup \{q\})$. The global transition function Δ_S of S is given by applying the standard CA global transition function Δ (as in Definition 3.7) followed by removing all cells in the state $\otimes$; that is, $\Delta_S = \Phi \circ \Delta$, where $\Phi\colon Q^{\mathbb{Z}} \to Q^{\mathbb{Z}}$ is the (partial) $\omega\omega$-word homomorphism resulting from the extension to $Q^{\mathbb{Z}}$ of the map $\varphi\colon Q \to Q$ with $\varphi(\otimes) = \varepsilon$ and $\varphi(x) = x$ for $x \in Q \setminus \{\otimes\}$. For $t\colon \mathbb{N}_+ \to \mathbb{N}_0$, $\mathsf{SCA}[t]$ is the class of languages accepted by SCAs with time complexity $O(t(n))$, where n denotes the input length.

Note that Φ is only partial since, for instance, any $\omega\omega$-word in $\otimes^\omega \Sigma^* \otimes^\omega$ has no proper image (as it is not mapped to a $\omega\omega$-word). Hence, Δ_S is also only a partial function (on $Q^{\mathbb{Z}}$); nevertheless, Φ is total on the set of $\omega\omega$-words in which $\otimes$ occurs only finitely often and, in particular, Δ_S is total on the set of configurations arising from initial configurations for finite input words (which is the setting we are interested in).

The acceptance condition of SCAs is the same as in Definition 3.7 (i.e., acceptance is dictated by cell zero). Unlike in standard CAs, the index of one same cell can differ from one configuration to the next; that is, a cell index does not uniquely determine a cell on its own (rather, only in conjunction with a time step). This is a consequence of applying Φ, which contracts the global configuration towards cell zero. More precisely, for a configuration $c \in Q^{\mathbb{Z}}$, the cell with index $i \geq 0$ in $\Delta(c)$ corresponds to that with index $i + d_i$ in c, where d_i is the number of cells with index $\leq i$ in c that were deleted in the

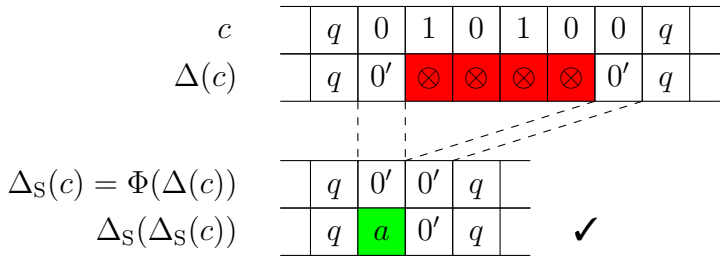


Figure 3.1.: Computation of an SCA that recognizes $L = \{w \in \{0,1\}^+ \mid w(0) = w(|w|-1)\}$ in $O(1)$ time. Here, the input word is $010100 \in L$.

transition to $\Delta(c)$. This also implies the cell with index zero in $\Delta(c)$ is the same as that in c with minimal positive index that was not deleted in the transition to $\Delta(c)$; thus, in any time step, cell zero is the leftmost active cell (unless all cells are inactive; in fact, cell zero is inactive if and only if all other cells are inactive). Granted, what indices a cell has is of little importance when one is interested only in the configurations of an SCA and their evolution; nevertheless, they are relevant when simulating an SCA with another machine model (as we do in Sections 3.3.3 and 3.4).

Naturally, $\mathsf{CA}[t] \subseteq \mathsf{SCA}[t]$ for every function t, and $\mathsf{SCA}[\mathrm{poly}] = \mathsf{CA}[\mathrm{poly}]$. For sublinear t, $\mathsf{SCA}[t]$ contains non-regular languages if, for instance, $t(n) = \Omega(\log n)$ (see below); hence, the inclusion of $\mathsf{CA}[t]$ in $\mathsf{SCA}[t]$ in strict. In fact, this is the case even if we consider only regular languages. One simple example is

$$L = \{w \in \{0,1\}^+ \mid w(0) = w(|w|-1)\},$$

which is in $\mathsf{SCA}[1]$ and regular but not in $\mathsf{CA}[o(n)] = \mathsf{CA}[O(1)]$. One obtains an SCA for L by having all cells whose both neighbors are active delete themselves in the first step; the two remaining cells then compare their states, and cell zero accepts if and only if this comparison succeeds or if the input has length 1 (which it can notice immediately since it is only for such words that it has two inactive neighbors). Formally, the local transition function δ is such that, for $z_1, z_3 \in \{0,1,q\}$ and $z_2 \in \{0,1\}$, $\delta(z_1,z_2,z_3) = \otimes$ if both z_1 and z_3 are in $\{0,1\}$, $\delta(z_1,z_2,z_3) = z_2'$ if $z_1 = q$ or $z_3 = q$, and $\delta(q,z_2',z_2') = \delta(q,z_2',q) = a$; in all other cases, δ simply conserves the cell's state. See Figure 3.1 for an example.

Using a textbook technique to simulate a (bounded) CA with an LBA (simply skipping deleted cells), we have:

Proposition 3.9. *For every function $t\colon \mathbb{N}_+ \to \mathbb{N}_+$ computable by an LBA in $O(n \cdot t(n))$ time,* $\mathsf{SCA}[t(n)] \subseteq \mathsf{TISP}[n \cdot t(n), n]$.

The inclusion is actually proper (see Corollary 3.19). Using the well-known result that $\mathsf{TIME}[o(n \log n)] = \mathsf{REG}$ [67], it follows that at least a logarithmic time bound is needed for SCAs to recognize languages which are not regular:

Corollary 3.10. $\mathsf{SCA}[o(\log n)] \subsetneq \mathsf{REG}$.

This bound is tight: It is relatively easy to show that any language accepted by ACAs in $t(n)$ time can also be accepted by an SCA in $t(n) + O(1)$ time. Since there is a non-regular language recognizable by ACAs [60] in $O(\log n)$ time, the same language is recognizable by an SCA in $O(\log n)$ time.

For any finite, non-empty set Σ, we say a function $f\colon \Sigma^+ \to \Sigma^+$ is *computable in place* by an (S)CA if there is an (S)CA S which, given $x \in \Sigma^+$ as input (surrounded by inactive cells), produces $f(x)$. Additionally, $g\colon \mathbb{N}_+ \to \mathbb{N}_+$ is *constructible in place* by an (S)CA if $g(n) \leq 2^n$ and there is an (S)CA S which, given $n \in \mathbb{N}_0$ in unary, produces $\mathrm{bin}_n(g(n) - 1)$ (i.e., $g(n) - 1$ in binary). Note the set of functions computable or constructible in place by an (S)CA in at most $t(n)$ time, where n is the input length and $t\colon \mathbb{N}_+ \to \mathbb{N}_+$ is some function, includes (but is not limited to) all functions computable by an LBA in at most $t(n)$ time.

3.3. Capabilities and Limitations of Sublinear-Time SCAs

3.3.1. Block Languages

Let Σ be a finite, non-empty set. For $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $x, y \in \Sigma^+$, $\binom{x}{y}$ denotes the (unique) word in $(\Sigma_\varepsilon \times \Sigma_\varepsilon)^+$ of length $\max\{|x|, |y|\}$ for which $\binom{x}{y}(i) = (x(i), y(i))$, where $x(i) = y(j) = \varepsilon$ for $i \geq |x|$ and $j \geq |y|$.

Definition 3.11 (Block word)**.** Let $n, m, b \in \mathbb{N}_+$ be such that $b \geq n$ and $m \leq 2^n$. A word w is said to be an (n, m, b)-*block word* (over Σ) if it is of the form

$$w = w_0 \# w_1 \# \cdots \# w_{m-1}$$

with

$$w_i = \begin{pmatrix} \mathrm{bin}_n(x_i) \\ y_i \end{pmatrix},$$

where $x_0 \geq 0$, $x_{i+1} = x_i + 1$ for every i, $x_{m-1} < 2^n$, and $y_i \in \Sigma^b$. In this context, w_i is the *i-th block* of w.

Hence, every (n, m, b)-block word w has m many blocks of length b, and its total length is $|w| = (b + 1)m - 1 \in \Theta(bm)$. For example,

$$w = \begin{pmatrix} 01 \\ 0100 \end{pmatrix} \# \begin{pmatrix} 10 \\ 1100 \end{pmatrix} \# \begin{pmatrix} 11 \\ 1000 \end{pmatrix}$$

is a $(2, 3, 4)$-block word with $x_0 = 1$, $y_0 = 0100$, $y_1 = 1100$, and $y_2 = 1000$. n is implicitly encoded by the entries in the upper track (i.e., the x_i) and we shall see m and b as parameters depending on n (see Definition 3.12 below), so the structure of each block can be verified locally (i.e., by inspecting the immediate neighborhood of every block). Note the block

numbering starts with an arbitrary x_0; this is intended so that, for $m' < m$, an (n, m, b)-block word admits (n, m', b)-block words as infixes (which would not be the case if we required, say, $x_0 = 0$).

When referring to block words, we use N for the block word length $|w|$ and reserve n for indexing block words of different block length, overall length, or total number of blocks (or any combinations thereof). With m and b as parameters depending on n, we obtain sets of block words:

Definition 3.12 (Block language)**.** Let $m, b\colon \mathbb{N}_+ \to \mathbb{N}_+$ be non-decreasing and constructible in place by a CA in $O(m(n) + b(n))$ time. Furthermore, let $b(n) \geq n$ and $m(n) \leq 2^n$. Then, $\mathfrak{B}_b^m$ denotes the set of all $(n, m(n), b(n))$-block words for $n \in \mathbb{N}_+$, and every subset $L \subseteq \mathfrak{B}_b^m$ is an $((n, m, b)$-*)block language* (over Σ).

An SCA can *verify* its input is a valid block word in $O(b(n))$ time, that is, locally check that the structure and contents of the blocks are consistent (i.e., as in Definition 3.11). This can be realized using standard CA techniques without need of shrinking (see Chapter 2 as well as [60] for constructions). Recall Definition 3.8 does not require an SCA S to explicitly reject inputs not in $L(S)$, that is, the time complexity of S on an input w is only defined for $w \in L(S)$. As a result, when $L(S)$ is a block language, the time spent verifying that w is a block word is only relevant if $w \in L(S)$ and, in particular, if w is a (valid) block word. Provided the state of every cell in S eventually impacts its decision to accept (which is the case for all constructions we describe), it suffices to have a cell mark itself with an error flag whenever a violation in w is detected (even if other cells continue their operation as normal); since every cell is relevant towards acceptance, this eventually prevents S from accepting (and, since $w \notin L(S)$, it is irrelevant how long it takes for this to occur). Thus, for the rest of this chapter, when describing an SCA for a block language, we implicitly require that the SCA checks its input is a valid block word beforehand.

As stated in the introduction, our interest in block words is as a special input format. There is a natural bijection between any language and a block version of it, namely by mapping each word z to a block word w in which each block w_i contains a symbol $z(i)$ of z (padded up to the block length b) and the blocks are numbered from 0 to $|z| - 1$:

Definition 3.13 (Block version of a language)**.** Let $L \subseteq \Sigma^+$ be a language and b as in Definition 3.12. The *block version* $\mathsf{Block}_b(L)$ of L (with blocks of length b) is the block language for which, for every $z \in \Sigma^+$, $z \in L$ holds if and only if we have $w \in \mathsf{Block}_b(L)$ where w is the $(n, m, b(n))$-block word (as in Definition 3.11) with $m = |z|$, $n = \lceil \log m \rceil$, $x_0 = 0$, and $y_i = z(i)0^{b(n)-1}$ for every $i \in [0, m - 1]$.

Note that, for any such language L, $\mathsf{Block}_b(L) \notin \mathsf{REG}$ for any b (since $b(n) \geq n$ is not constant); hence, $\mathsf{Block}_b(L) \in \mathsf{SCA}[t]$ only for $t(n) = \Omega(\log n)$ (and constructible b). For $b(n) = n$, $\mathsf{Block}_n(L)$ is the block version with minimal padding.

For any two finite, non-empty sets Σ_1 and Σ_2, say a function $f\colon \Sigma_1^+ \to \Sigma_2^+$ is *non-stretching* if $|f(x)| \leq |x|$ for every $x \in \Sigma_1^+$. We now define k-blockwise maps, which are maps that

operate on block words by grouping $k(n)$ many blocks together and mapping each such group (in a non-stretching manner) to a single block of length at most $(b(n)+1)k(n)-1$.

Definition 3.14 (Blockwise map)**.** Let $k\colon \mathbb{N}_+ \to \mathbb{N}_+$, $k(n) \geq 2$, be a non-decreasing function and constructible in place by a CA in $O(k(n))$ time. A map $g\colon \mathfrak{B}_b^{km} \to \mathfrak{B}_b^m$ is a *k-blockwise map* if there is a non-stretching $g'\colon \mathfrak{B}_b^k \to \Sigma^+$ such that, for every $w \in \mathfrak{B}_b^{km}$ (as in Definition 3.11) and $w'_i = w_{ik}\# \cdots \# w_{(i+1)k-1}$:

$$g(w) = \begin{pmatrix} \mathrm{bin}_n(x_0) \\ g'(w'_0) \end{pmatrix} \# \cdots \# \begin{pmatrix} \mathrm{bin}_n(x_{m-1}) \\ g'(w'_{m-1}) \end{pmatrix}.$$

Using blockwise maps, we obtain a very natural form of reduction operating on block words and which is highly compatible with sublinear-time SCAs as a computational model. The reduction divides an (n, km, b)-block word in m many groups of k many contiguous blocks and, as a k-blockwise map, maps each such group to a single block (of length b):

Definition 3.15 (Blockwise reducible)**.** For block languages L and L', L is *(k-)blockwise reducible* to L' if there is a computable k-blockwise map $g\colon \mathfrak{B}_b^{km} \to \mathfrak{B}_b^m$ such that, for every $w \in \mathfrak{B}_b^{km}$, we have $w \in L$ if and only if $g(w) \in L'$.

Since every application of the reduction reduces the instance length by a factor of approximately k, logarithmically many applications suffice to produce a trivial instance (i.e., an instance consisting of a single block). This gives us the following computational paradigm of chaining blockwise reductions together:

Lemma 3.16. *Let $k, r\colon \mathbb{N}_+ \to \mathbb{N}_0$ be functions, and let $L \subseteq \mathfrak{B}_b^{k^r}$ be such that there is a series $L = L_0, L_1, \ldots, L_{r(n)}$ of languages with $L_i \subseteq \mathfrak{B}_b^{k^{r-i}}$ and such that L_i is $k(n)$-blockwise reducible to L_{i+1} via the (same) blockwise reduction g. Furthermore, let g' be as in Definition 3.14, and let $t_{g'}\colon \mathbb{N}_+ \to \mathbb{N}_+$ be non-decreasing and such that, for every $w' \in \mathfrak{B}_b^r$, $g'(w')$ is computable in place by an SCA in $O(t_{g'}(|w'|))$ time. Finally, let $L_{r(n)} \in \mathsf{SCA}[t]$ for some function $t\colon \mathbb{N}_+ \to \mathbb{N}_+$. Then,*

$$L \in \mathsf{SCA}\left[r(n)t_{g'}(O(k(n)b(n))) + O(b(n)) + t(b(n))\right].$$

Proof. We consider the SCA S which, given $w \in \mathfrak{B}_b^{k^r}$, repeatedly applies the reduction g, where each application of g is computed by applying g' on each group of relevant blocks (i.e., the w'_i from Definition 3.14) in parallel.

One detail to note is that this results in the same procedure P being applied to different groups of blocks in parallel, but it may be so that P requires more time for one group of blocks than for the other. Thus, we allow the entire process to be carried out asynchronously but require that, for each group of blocks, the respective results be present before each execution of P is started. (One way of realizing this, for instance, is having the first block in the group send a signal across the whole group to ensure all inputs are

available and, when it arrives at the last block in the group, another signal is sent to trigger the start of P.)

Using that $t_{g'}$ is non-decreasing and that g' is non-stretching, the time needed for each execution of P is $t_{g'}(|w_i'|) \in t_{g'}(O(k(n)b(n)))$ (which is not impacted by the considerations above) and, since there are $r(n)$ reductions in total, we have $r(n) \cdot t_{g'}(O(k(n)b(n)))$ time in total. Once a single block is left, the cells in this block synchronize themselves and then behave as in the SCA S' for $L_{r(n)}$ guaranteed by the assumption; using a standard synchronization algorithm, this requires $O(b(n))$ for the synchronization, plus $t(b(n))$ time for emulating S'. □

3.3.2. Block Languages and Parallel Computation

In this section, we prove the first limitation of SCAs discussed in the introduction (i.e., Lemma 3.17) and which renders them unable of accepting the languages PAR, MOD_q, MAJ, and THR_k (defined next) in sublinear time. Nevertheless, as is shown in Proposition 3.21, the *block versions* of these languages can be accepted quite efficiently. This motivates the block word presentation for inputs; that is, this first limitation concerns only the *presentation* of instances (and, hence, is not a *computational* limitation of SCAs).

Let $q > 2$ and let $k\colon \mathbb{N}_+ \to \mathbb{N}_+$ be constructible in place by a CA in at most $t_k(n)$ time for some $t_k\colon \mathbb{N}_+ \to \mathbb{N}_+$. Additionally, let PAR (resp., MOD_q; resp., MAJ; resp., THR_k) be the language consisting of every word $w \in \{0,1\}^+$ for which $|w|_1$ is even (resp., $|w|_1 = 0 \pmod q$; resp., $|w|_1 \geq |w|_0$; resp., $|w|_1 \geq k(|w|)$).

The following is a simple limitation of sublinear-time CA models such as ACAs (see also [105]) which we show also to hold for SCAs.

Lemma 3.17. *Let S be an SCA with input alphabet Σ, and let $x \in \Sigma$ be such that there is a minimal $t \in \mathbb{N}_+$ for which $\Delta_S^t(y) = \varepsilon$, where $y = x^{2t+1}$ (i.e., the symbol x concatenated $2t+1$ times with itself). Then, for every $z_1, z_2 \in \Sigma^+$, $w = z_1yz_2 \in L(S)$ holds if and only if for every $i \in \mathbb{N}_0$ we have $w_i = z_1yx^iz_2 \in L(S)$.*

Proof. Given w and i as above, we show $w_i \in L(S)$; the converse is trivial. Since w and w_i both have z_1y as prefix and $\Delta_S^{t'}(y) \neq \varepsilon$ for $t' < t$, if S accepts w in t' steps, then it also accepts w_i in t' steps. Thus, assume S accepts w in $t' \geq t$ steps, in which case it suffices to show $\Delta_S^t(w) = \Delta_S^t(w_i)$. To this end, let α_j for $j \in [0,t]$ be such that $\alpha_0 = x$ and $\alpha_{j+1} = \delta(\alpha_j, \alpha_j, \alpha_j)$. Hence, $\Delta(\alpha_j^{k+2}) = \alpha_{j+1}^k$ holds for every $k \in \mathbb{N}_+$ (and $j < t$) and, by an inductive argument as well as by the assumption on y (i.e., $\alpha_t = \otimes$),

$$\Delta_S^t\big(yx^i\big) = \Delta_S^t\big(\alpha_0^{2t+i+1}\big) = \varepsilon.$$

Using this along with $|y| \geq t$ and $y \in \{x\}^+$, we have

$$\Delta_S^t\big(q^tz_1yx^i\big) = \Delta_S^t\big(q^tz_1y\big)$$

and

$$\Delta_S^t(yx^iz_2q^t) = \Delta_S^t(x^iyz_2q^t) = \Delta_S^t(yz_2q^t);$$

hence, $\Delta_S^t(w) = \Delta_S^t(w_i)$ follows. □

An implication of Lemma 3.17 is that every unary language $U \in \mathsf{SCA}[o(n)]$ is either finite or cofinite. As $\mathsf{PAR} \cap \{1\}^+$ is neither finite nor cofinite, we can prove:

Proposition 3.18. $\mathsf{PAR} \notin \mathsf{SCA}[o(n)]$ *(where n is the input length).*

Proof. Let S be an SCA with $L(S) = \mathsf{PAR}$. We show S must have $\Omega(n)$ time complexity on inputs from the infinite set

$$U = \{1^{2m} \mid m \in \mathbb{N}_+\} \subset \mathsf{PAR}.$$

If $\Delta_S^t(1^{2t+1}) = \varepsilon$ for some $t \in \mathbb{N}_0$, then, by Lemma 3.17, $L(S) \cap \{1\}^+$ is either finite or cofinite, which contradicts $L(S) = \mathsf{PAR}$. Hence, $\Delta_S^t(1^{2t+1}) \neq \varepsilon$ for every $t \in \mathbb{N}_0$. In this case, the trace of cell zero on input $w = 11^{2t+1}1$ in the first t steps is the same as that on input $w' = 11^{2t+1}11$. Since $w \in \mathsf{PAR}$ if and only if $w' \notin \mathsf{PAR}$, it follows that S has $\Omega(t) = \Omega(n)$ time complexity on U. □

Corollary 3.19. $\mathsf{REG} \not\subseteq \mathsf{SCA}[o(n)]$.

The argument above generalizes to MOD_q, MAJ, and THR_k with $k \in \omega(1)$. For MOD_q, consider $U = \{1^{qm} \mid m \in \mathbb{N}_+\}$. For MAJ and THR_k, set $U = \{0^m1^m \mid m \in \mathbb{N}_+\}$ and $U = \{0^{n-k(n)}1^{k(n)} \mid n \in \mathbb{N}_+\}$, respectively; in this case, U is not unary, but the argument easily extends to the unary suffixes of the words in U.

Corollary 3.20. $\mathsf{MOD}_q, \mathsf{MAJ} \notin \mathsf{SCA}[o(n)]$. *Also,* $\mathsf{THR}_k \in \mathsf{SCA}[o(n)]$ *if and only if* $k = O(1)$.

The *block versions* of these languages, however, are not subject to the limitation above:

Proposition 3.21. *For* $L \in \{\mathsf{PAR}, \mathsf{MOD}_q, \mathsf{MAJ}\}$, $\mathsf{Block}_n(L) \in \mathsf{SCA}[(\log N)^2]$, *where* $N = N(n)$ *is the input length. Also,* $\mathsf{Block}_n(\mathsf{THR}_k) \in \mathsf{SCA}[(\log N)^2 + t_k(n)]$.

Proof. Given $L \in \{\mathsf{PAR}, \mathsf{MOD}_q, \mathsf{MAJ}, \mathsf{THR}_k\}$, we construct an SCA S for $L' = \mathsf{Block}_n(L)$ with the purported time complexity. Let $w \in \mathfrak{B}_n^m$ be an input of S. For simplicity, we assume that, for every such w, $m = m(n) = 2^n$ is a power of two; the argument extends to the general case in a simple manner. Hence, we have $N = |w| = nm$ and $n = \log m \in \Theta(\log N)$.

Let $L_0 \subset \mathfrak{B}_n^m$ be the language containing every such block word $w \in \mathfrak{B}_n^m$ for which, for y_i as in Definition 3.11 and $y = \sum_{i=0}^{m-1} y_i$, we have $f_L(y) = f_{L,n}(y) = 0$, where $f_{\mathsf{PAR}}(y) = y \bmod 2$, $f_{\mathsf{MOD}_q}(y) = y \bmod q$, $f_{\mathsf{MAJ}}(y) = 0$ if and only if $y \geq 2^{n-1}$, and $f_{\mathsf{THR}_k}(y) = 0$ if and only if

$y \geq k(n)$. Thus, (under the previous assumption) we have $L_0 = L'$ (and, in the general case, $L_0 = L' \cap \mathfrak{B}_n^{2^n}$).

Then, L_0 is 2-blockwise reducible to a language $L_1 \subseteq \mathfrak{B}_n^{m/2}$ by mapping every $(n, 2, n)$-block word of the form

$$\begin{pmatrix} \mathrm{bin}_n(2x) \\ y_{2x} \end{pmatrix} \# \begin{pmatrix} \mathrm{bin}_n(2x+1) \\ y_{2x+1} \end{pmatrix}$$

with $x \in [0, 2^{n-1} - 1]$ to

$$\begin{pmatrix} \mathrm{bin}_n(x) \\ y_{2x} + y_{2x+1} \end{pmatrix}.$$

To do so, it suffices to compute $\mathrm{bin}_n(x)$ from $\mathrm{bin}_n(2x)$ and add the y_{2x} and y_{2x+1} values in the lower track; using basic CA arithmetic and cell communication techniques, this is realizable in $O(n)$ time. Repeating this procedure, we obtain a chain of languages $L_0, \ldots, L_n$ such that L_i is 2-blockwise reducible to L_{i+1} in $O(n)$ time. By Lemma 3.16, $L' \in \mathsf{SCA}[n^2 + t(n)]$ follows, where $t\colon \mathbb{N}_+ \to \mathbb{N}_0$ is such that $L_n \in \mathsf{SCA}[t]$. For $L \in \{\mathsf{PAR}, \mathsf{MOD}_q, \mathsf{MAJ}\}$, checking the above condition on $f_L(y)$ can be done in $t(n) = O(n)$ time; as for $L = \mathsf{THR}_k$, we must also compute k, so we have $t(n) = O(n + t_k(n))$.

The general case follows from adapting the above reductions so that words with an odd number of blocks are also accounted for (e.g., by ignoring the last block of w and applying the reduction on the first $m - 1$ blocks). □

3.3.3. An Optimal SCA Lower Bound for a Block Language

Corollary 3.19 already states SCAs are strictly less capable than streaming algorithms. However, the argument bases exclusively on long unary subwords in the input (i.e., Lemma 3.17) and, therefore, does not apply to block languages. Hence Theorem 3.5, which shows SCAs are more limited than streaming algorithms *even considering only block languages*:

Theorem 3.5. *There is a language L_1 for which $\mathsf{Block}_n(L_1) \notin \mathsf{SCA}[o(N/\log N)]$ (N being the instance length) can be accepted by an $O(\log N)$-space streaming algorithm with $\tilde{O}(\log N)$ update time.*

Let L_1 be the language of words $w \in \{0,1\}^+$ such that $|w| = 2^n$ is a power of two and, for $i = w(0)w(1)\cdots w(n-1)$ (seen as an n-bit binary integer), $w(i) = 1$. It is not hard to show that its block version $\mathsf{Block}_n(L_1)$ can be accepted by an $O(\log m)$-space streaming algorithm with $\tilde{O}(\log m)$ update time.

The $O(N/\log N)$ upper bound for $\mathsf{Block}_n(L_1)$ is optimal since there is an $O(N/\log N)$ time SCA for it: Shrink every block to its respective bit (i.e., the y_i from Definition 3.11), reducing the input to a word w' of $O(N/\log N)$ length; while doing so, mark the bit corresponding to the n-th block. Then shift the contents of the first n bits as a counter that decrements itself every new cell it visits and, when it reaches zero, signals acceptance if the cell it is

currently at contains a 1. Using counter techniques as in [106, 114], this requires $O(|w'|)$ time.

The proof of Theorem 3.5 bases on communication complexity. The basic setting is a game with two players A and B (both with unlimited computational resources) which receive inputs w_A and w_B, respectively, and must produce an answer to the problem at hand while exchanging a limited amount of bits. We are interested in the case where the concatenation $w = w_A w_B$ of the inputs of A and B is an input to an SCA and A must output whether the SCA accepts w. More importantly, we analyze the case where *only B is allowed to send messages*, that is, the case of *one-way* communication.[2]

Definition 3.22 (One-way communication complexity)**.** Let $m, f : \mathbb{N}_+ \to \mathbb{N}_+$ be functions with $0 < m(N) \leq N$. A language $L \subseteq \Sigma^+$ is said to have *(m-)one-way communication complexity f* if there are families of algorithms (with unlimited computational resources) $(A_N)_{N\in\mathbb{N}_+}$ and $(B_N)_{N\in\mathbb{N}_+}$ such that the following holds for every $w \in \Sigma^*$ of length $|w| = N$, where $w_A = w[0, m(N) - 1]$ and $w_B = w[m(N), N - 1]$:

1. $|B_N(w_B)| \leq f(N)$.
2. $A_N(w_A, B(w_B)) = 1$ (i.e., accept) if and only if $w \in L$.

$\mathfrak{C}^m_{\mathrm{ow}}(L)$ indicates the (pointwise) minimum over all such functions f.

Note that A_N and B_N are nonuniform, so the length N of the (complete) input w is known implicitly by both algorithms.

Lemma 3.23. *For any computable $t : \mathbb{N}_+ \to \mathbb{N}_+$ and m as in Definition 3.22, if $L \in \mathrm{SCA}[t]$, then $\mathfrak{C}^m_{\mathrm{ow}}(L)(N) = O(t(N))$.*

The proof idea is to have A and B simulate the SCA for L simultaneously, with A maintaining the first half c_A of the SCA configuration and B the second half c_B. (Hence, A is aware of the leftmost active state in the SCA and can detect whether the SCA accepts or not.) The main difficulty is guaranteeing that A and B can determine the states of the cells on the right (resp., left) end of c_A (resp., c_B) despite the respective local configurations "overstepping the boundary" between c_A and c_B. Hence, for each step in the simulation, B communicates the states of the two leftmost cells in c_B; with this, A can compute the states of all cells of c_A in the next configuration as well as that of the leftmost cell α of c_B, which is added to c_A. (See Figure 3.2 for an illustration.) This last technicality is needed due to one-way communication, which renders it impossible for B to determine the next state of α (since its left neighbor is in c_A and B cannot receive messages from A). As the simulation requires at most $t(N)$ steps and B sends $O(1)$ information at each step, this yields the purported $O(t(N))$ upper bound.

[2] One-way communication complexity can also been defined as the maximum over *both* communication directions (i.e., B to A and A to B; see [31] for an example in the setting of CAs). Since our goal is to prove a *lower bound* on communication complexity, it suffices to consider a single (arbitrary) direction (in this case B to A).

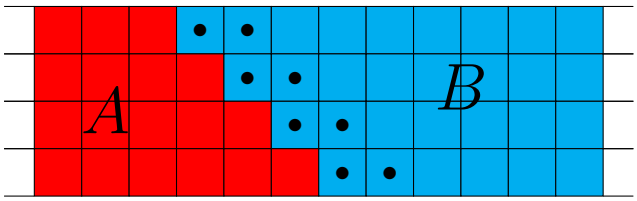


Figure 3.2.: Simulating an SCA with low one-way communication complexity. (For simplicity, in this example the SCA does not shrink.) B communicates the states of the cells marked with "•". The colors indicate which states are computed by each player.

The attentive reader may have noticed this discussion does not address the fact that the SCA may shrink; indeed, we shall also prove that shrinking does not interfere with this strategy.

Proof. Let S be an SCA for L with time complexity $O(t)$. Furthermore, let Q be the state set of S and $q \in Q$ its inactive state. We construct algorithms A_N and B_N as in Definition 3.22 and such that $|B_N(w_B)| \leq 2\log(|Q|)t(N)$.

Fix $N \in \mathbb{N}_+$ and an input $w \in \Sigma^N$. For $w_B^0 = w_B q^{2t(N)+2}$ and $w_B^{i+1} = \Delta_S(w_B^i)$ for $i \in \mathbb{N}_0$, B_N computes and outputs the concatenation

$$B_N(w_B) = w_B^0(0)w_B^0(1)w_B^1(0)w_B^1(1)\cdots w_B^{t(N)}(0)w_B^{t(N)}(1).$$

Similarly, let $w_A^0 = q^{2t(N)+2}w_A$ and $w_A^{i+1} = \Delta_S(w_A^i w_B^i(0)w_B^i(1))$ for $i \in \mathbb{N}_0$. A computes $t(N)$ and w_A^i for $i \in [0, t(N)]$ and accepts if there is any j such that $w_A^i(j)$ is an accept state of S and $w_A^i(j') = q$ for all $j' < j$; otherwise, A rejects.

To prove the correctness of A, we show by induction on $i \in \mathbb{N}_0$:

$$w_A^i w_B^i = \Delta_S^i(q^{2t(n)+2}wq^{2t(n)+2}).$$

Hence, the $w_A^i(j)$ of above corresponds to the state of cell zero in step i of S, and it follows that A accepts if and only if S does. The induction basis is trivial. For the induction step, let $w' = \Delta_S(w_A^i w_B^i)$. Using the induction hypothesis, it suffices to prove $w_A^{i+1}w_B^{i+1} = w'$. Note first that, due to the definition of w_A^{i+1} and w_B^{i+1}, we have

$$w' = \Delta_S(w_A^i)\alpha\beta\Delta_S(w_B^i),$$

where $\alpha, \beta \in Q \cup \{\varepsilon\}$. Let $\alpha_1 = w_A^i(|w_A^i|-2)$, $\alpha_2 = w_A^i(|w_A^i|-1)$, and $\alpha_3 = w_B^i(0)$ and notice $\alpha = \delta(\alpha_1, \alpha_2, \alpha_3)$; the same is true for β and $\beta_1 = \alpha_2$, $\beta_2 = \alpha_3$, and $\beta_3 = w_B^i(1)$. Hence, we have $w_A^{i+1} = \Delta_S(w_A^i)\alpha\beta$, and the claim follows. □

We are now in position to prove Theorem 3.5.

Proof of Theorem 3.5. We prove that, for our language L_1 of before and $m(n) = n(n+1)$ (i.e., A_N receives the first n input blocks), $\mathfrak{C}_{\text{ow}}^m(\text{Block}_n(L_1))(N) \geq 2^n - n$. Since the input length is $N \in \Theta(n \cdot 2^n)$, the claim then follows from the contrapositive of Lemma 3.23.

The proof is by a counting argument. Let A_N and B_N be as in Definition 3.22, and let $Y = \{0,1\}^{2^n-n}$. The basic idea is that, for the same input w_A, if B_N is given different inputs w_B and w'_B but $B_N(w_B) = B_N(w'_B)$, then $w = w_A w_B$ is accepted if and only if $w' = w_A w'_B$ is accepted. Hence, for any $y, y' \in Y$ with $y \neq y'$, we must have $B_N(w_B) \neq B_N(w'_B)$, where $w_B, w'_B \in \mathfrak{B}_n^{2^n-n}$ are the block word versions of y and y', respectively; this is because, letting $j \in [0, 2^n - n]$ be such that $y(j) \neq y'(j)$ and $z = \mathrm{bin}_n(n + j)$, precisely one of the words zy and zy' is in L_1 (and the other not). Finally, note there is a bijection between Y and the set Y' of block words in $\mathfrak{B}_n^{2^n-n}$ whose block numbering starts with $n + 1$ (i.e., $x_0 = n+1$, where x_0 is as in Definition 3.11) and with block entries of the form $a0^{n-1}$ where $a \in \{0,1\}$ (i.e., Y' is essentially the block version of Y as in Definition 3.13 but where we set $x_0 = n + 1$ instead of $x_0 = 0$). We conclude

$$\mathfrak{C}^m_{\mathrm{ow}}(\mathsf{Block}_n(L_1))(N) \geq |Y'| = |Y| = 2^n - n,$$

from which the claim follows. □

3.4. Simulation of an SCA by a Streaming Algorithm

In this section, we recall and prove:

Theorem 3.4. *Let $t : \mathbb{N}_+ \to \mathbb{N}_+$ be computable by an $O(t)$-space random access machine (as in Definition 3.6) in $O(t \log t)$ time. Then, if $L \in \mathsf{SCA}[t]$, there is an $O(t)$-space streaming algorithm for L with $O(t \log t)$ update and $O(t^2 \log t)$ reporting time.*

Before we state the proof, we first introduce some notation. Having fixed an input w, let $c_t(i)$ denote the state of cell i in step t on input w. Note that here we explicitly allow $c_t(i)$ to be the state $\otimes$ and also disregard any changes in indices caused by cell deletion; that is, $c_t(i)$ refers to the *same cell i* as in the initial configuration c_0 (of Definition 3.7; see also the discussion following Definition 3.8). For a finite, non-empty $I = [a, b] \subseteq \mathbb{Z}$ and $t \in \mathbb{N}_0$, let

$$\mathrm{nndcl}_t(I) = \max\{i \mid i < a, c_t(i) \neq \otimes\}$$

denote the nearest non-deleted cell to the left of I; similarly,

$$\mathrm{nndcr}_t(I) = \min\{i \mid i > b, c_t(i) \neq \otimes\}$$

is the nearest such cell to the right of I.

Proof. Let S be an $O(t)$-time SCA for L. Using S, we construct a streaming algorithm A (Algorithm 1) for L and prove it has the purported complexities.

Algorithm 1: Streaming algorithm A

Compute $t(|w|)$;
Initialize lists `leftIndex`, `centerIndex`, `leftState`, and `centerState`;
`leftIndex`$[0] \leftarrow -1$;
`leftState`$[0] \leftarrow q$;
`centerIndex`$[0] \leftarrow 0$;
`centerState`$[0] \leftarrow w(0)$;
`next` $\leftarrow 1$;
$j_0 \leftarrow 0$;
for $\tau \leftarrow 0, \ldots, t(|w|) - 1$ **do**
A $j \leftarrow j_0$;
B **if** *next* $< |w|$ **then**
C `rightIndex` $\leftarrow$ `next`;
  `rightState` $\leftarrow w($`next`$)$;
  `next` $\leftarrow$ `next` $+ 1$;
  else
D `rightIndex` $\leftarrow |w|$;
  `rightState` $\leftarrow q$;
  $j_0 \leftarrow j_0 + 1$;
  end
  while $j \leq \tau$ **do**
E `newRightIndex` $\leftarrow$ `centerIndex`$[j]$;
  `newRightState` $\leftarrow \delta($`leftState`$[j]$, `centerState`$[j]$, `rightState`$)$;
  `leftIndex`$[j] \leftarrow$ `centerIndex`$[j]$;
  `leftState`$[j] \leftarrow$ `centerState`$[j]$;
  `centerIndex`$[j] \leftarrow$ `rightIndex`;
  `centerState`$[j] \leftarrow$ `rightState`;
  `rightIndex` $\leftarrow$ `newRightIndex`;
  `rightState` $\leftarrow$ `newRightState`;
F **if** *rightState* $= \otimes$ **then goto** A;
  $j \leftarrow j + 1$;
  end
  `leftIndex`$[\tau + 1] \leftarrow -1$;
  `leftState`$[\tau + 1] \leftarrow q$;
  `centerIndex`$[\tau + 1] \leftarrow$ `rightIndex`;
  `centerState`$[\tau + 1] \leftarrow$ `rightState`;
G **if** *centerState*$[\tau + 1] = a$ **then accept**;
end
reject;

Construction. Let w be an input to A. To decide L, A computes the states of the cells of S in the time steps up to $t(|w|)$. In particular, A sequentially determines the state of the leftmost active cell in each of these time steps (starting from the initial configuration) and accepts if and only if at least one of these states is accepting. To compute these states efficiently, we use an approach based on dynamic programming, reusing space as the computation evolves.

A maintains lists `leftIndex`, `leftState`, `centerIndex`, and `centerState` and which are indexed by every step j starting with step zero and up to the current step τ. The lists `leftIndex` and `centerIndex` store cell indices while `leftState` and `centerState` store the states of the respective cells, that is, $\texttt{leftState}[j] = c_j(\texttt{leftIndex}[j])$ and $\texttt{centerState}[j] = c_j(\texttt{centerIndex}[j])$.

Recall the state $c_{j+1}(y)$ of a cell y in step $j+1$ is determined exclusively by the previous state $c_j(y)$ of y as well as the states $c_j(x)$ and $c_j(z)$ of the left and right neighbors x and z (respectively) of y in the previous step j (i.e., $x = \mathrm{nndcl}_j(y)$ and $z = \mathrm{nndcr}_j(y)$). In the variables maintained by A, x and $c_j(x)$ correspond to $\texttt{leftIndex}[j]$ and $\texttt{leftState}[j]$, respectively, and y and $c_j(y)$ to $\texttt{centerIndex}[j]$ and $\texttt{centerState}[j]$, respectively. z and $c_j(z)$ are not stored in lists but, rather, in the variables `rightIndex` and `rightState` (and are determined dynamically). The cell indices computed (i.e., the contents of the lists `leftIndex` and `centerIndex` and the variables `rightIndex` and `newRightIndex`) are not actually used by A to compute states and are inessential to the algorithm itself; we use them only to simplify the proof of correctness below (and, hence, do not count them towards the space complexity of A).

In each iteration of the **for** loop, A determines $c_{\tau+1}(z_0^\tau)$, where z_0^τ is the leftmost active cell of S in step τ, and stores it $\texttt{centerState}[\tau+1]$. `next` is the index of the next symbol of w to be read (or $|w|$ once every symbol has been read), and j_0 is the minimal time step containing a cell whose state must be known to determine $c_{\tau+1}(z_0^t)$ and remains 0 as long as $\texttt{next} < |w|$. Hence, the termination of A is guaranteed by the finiteness of w, that is, `next` can only be increased a finite number of times and, once all symbols of w have been read (i.e., the condition in line B no longer holds), by the increment of j_0 in line D.

In each iteration of the **while** loop, the algorithm starts from a local configuration in step j of a cell $y = \texttt{centerIndex}[j]$ with left neighbor $x = \texttt{leftIndex}[j] = \mathrm{nndcl}_j(y)$ and right neighbor $z = \texttt{rightIndex}[j] = \mathrm{nndcl}_j(y)$. It then computes the next state $c_{j+1}(y)$ of y and sets y as the new left cell and z as the new center cell for step j. As long as it is not deleted (i.e., $c_{j+1}(y) \neq \otimes$), y then becomes the right cell for step $j+1$. In fact, this is the only place (line F) in the algorithm where we need to take into consideration that S is a shrinking (and not just a regular) CA. The strategy we follow here is to continue computing states of cells to the right of the current center cell (i.e., $y = \texttt{centerIndex}[j]$) until the first cell to its right which has not deleted itself (i.e., $\mathrm{nndcr}_j(y)$) is found. With this non-deleted cell we can then proceed with the computation of the state of $\texttt{centerIndex}[j+1]$ in step $j+1$. Hence, if y has deleted itself, to compute the state of the next cell to its right we must either read the next symbol of w or, if there are no symbols left, use quiescent cell number $|w|$ as right neighbor in step j_0, computing states up until we are at step j again (hence the **goto** instruction).

Correctness. The following invariants hold for both loops in A:

1. $\texttt{centerIndex}[\tau] = \min\{z \in \mathbb{N}_0 \mid c_\tau(z) \neq \otimes\}$, that is, $\texttt{centerIndex}[\tau]$ is the leftmost active cell of S in step j.
2. If $j \leq \tau$, then:
   - $\texttt{rightIndex} = \mathrm{nndcr}_j(\texttt{centerIndex}[j])$.
   - $\texttt{rightState} = c_j(\texttt{rightIndex})$.
3. For every $j' \in [j_0, \tau]$:
   - $\texttt{leftIndex}[j'] = \mathrm{nndcl}_{j'}(\texttt{centerIndex}[j'])$.
   - $\texttt{leftState}[j'] = c_{j'}(\texttt{leftIndex}[j'])$.
   - $\texttt{centerState}[j'] = c_{j'}(\texttt{centerIndex}[j'])$.

These can be shown together with the observation that, following the assignment of newRightIndex and newRightState in line E, we have

$$\texttt{newRightState} = c_{j+1}\,(\texttt{newRightIndex})$$

and, in case $\texttt{newRightState} \neq \otimes$ and $j < \tau$, then also

$$\texttt{newRightIndex} = \mathrm{nndcr}_j\,(\texttt{centerIndex}[j+1])\,.$$

Using the above, it follows that after the execution of the **while** loop we have $j = \tau + 1$, $\texttt{rightState} \neq \otimes$, and $\texttt{rightState} = c_{\tau+1}(\texttt{rightIndex})$. Since then

$$\texttt{rightIndex} = \texttt{centerIndex}[j-1] = \texttt{centerIndex}[\tau],$$

we obtain

$$\texttt{rightIndex} = \min\{z \in \mathbb{N}_0 \mid c_{\tau+1}(z) \neq \otimes\}\,.$$

Hence, as

$$\texttt{centerState}[\tau+1] = \texttt{rightState} = c_{\tau+1}(\texttt{rightIndex})$$

holds in line G, if A then accepts, so does S accept w in step τ. Conversely, if A rejects, then S does not accept w in any step $\tau \leq t(|w|)$.

Complexity. The space complexity of A is dominated by the leftState and centerState lists, both of which have $O(t(|w|))$ many entries of $O(1)$ size. As mentioned above, we ignore the space used by the lists leftIndex and centerIndex and the variables rightIndex and newRightIndex since they are inessential (i.e., if we remove them as well as all instructions in which they appear, the algorithm obtained is equivalent to A).

As for the update time, note each list access or arithmetic operation costs $O(\log t(|w|))$ time (since $t(|w|)$ upper bounds all numeric variables). Every execution of the **while** loop body requires then $O(\log t(|w|))$ time and, since, there are at most $O(t(|w|))$ executions between

any two subsequent reads (i.e., line C), this gives us the purported $O(t(|w|)\log t(|w|))$ update time.

Finally, for the reporting time of A, as soon as $i = |w|$ holds after execution of line C (i.e., A has completed reading its input) we have that the **while** loop body is executed at most $\tau - j + 1$ times before line C is reached again. Every time this occurs (depending on whether line C is reached by the **goto** instruction or not), either j_0 or both j_0 and τ are incremented. Hence, since $\tau \leq t(|w|)$, we have an upper bound of $O(t(|w|)^2)$ executions of the **while** loop body, resulting (as above) in an $O(t(|w|)^2 \log t(|w|))$ reporting time in total. □

3.5. Hardness Magnification for Sublinear-Time SCAs

Let $K > 0$ be constant such that, for any function $s\colon \mathbb{N}_+ \to \mathbb{N}_+$, every circuit of size at most $s(n)$ can be described by a binary string of length at most $\ell(n) = Ks(n)\log s(n)$. In addition, let $\perp$ denote a string (of length at most $\ell(n)$) such that no circuit of size at most $s(n)$ has $\perp$ as its description. Furthermore, let $\mathsf{Merge}[s]$ denote the following search problem (adapted from [75]):

Given: the binary representation of $n \in \mathbb{N}_+$, the respective descriptions (padded to length $\ell(n)$) of circuits C_0 and C_1 such that $|C_i| \leq s(n)$, and $\alpha, \beta, \gamma \in \{0,1\}^n$ with $\alpha \leq \beta \leq \gamma < 2^n$.

Find: the description of a circuit C with $|C| \leq s(n)$ and such that $\forall x \in [\alpha, \beta - 1] : C(x) = C_0(x)$ and $\forall x \in [\beta, \gamma - 1] : C(x) = C_1(x)$; if no such C exists or $C_i = \perp$ for any i, answer with $\perp$.

Note that the decision version of $\mathsf{Merge}[s]$, that is, the problem of determining whether a solution to an instance $\mathsf{Merge}[s]$ exists is in Σ_2^p. Moreover, $\mathsf{Merge}[s]$ is Turing-reducible (in polynomial time) to a decision problem very similar to $\mathsf{Merge}[s]$ and which is also in Σ_2^p, namely the decision version of $\mathsf{Merge}[s]$ but with the additional requirement that the description of C admits a given string v of length $|v| \leq s(n)$ as a prefix.[3]

We now formulate our main theorem concerning SCAs and MCSP:

Theorem 3.24. *Let $s\colon \mathbb{N}_+ \to \mathbb{N}_+$ be constructible in place by a CA in $O(s(n))$ time. Furthermore, let $m = m(n)$ denote the maximum instance length of $\mathsf{Merge}[s]$, and let $f, g\colon \mathbb{N}_+ \to \mathbb{N}_+$ with $f(m) \geq g(m) \geq m$ be constructible in place by a CA in $O(f(m))$ time and $O(g(m))$ space. Then, for $b(n) = \lfloor g(m)/2 \rfloor$, if $\mathsf{Merge}[s]$ is computable in place by a CA in at most $f(m)$ time and $g(m)$ space, then the search version of $\mathsf{Block}_b(\mathsf{MCSP}[s])$ is computable by an SCA in $O(nf(m))$ time, where the instance size of the latter is in $\Theta(2^n b(n))$.*

[3] This is a fairly common construction in complexity theory for reducing search to decision problems; refer to [50] for the same idea applied in other contexts.

We are particularly interested in the repercussions of Theorem 3.24 *taken in the contrapositive*. Since $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{P} = \Sigma_2^p$, it also implies there is a poly-time Turing machine for $\mathsf{Merge}[s]$; since a CA can simulate a Turing machine with no time loss, for m as above we obtain:

Theorem 3.3. *For a certain* $m \in \mathsf{poly}(s(n))$*, if* $\mathsf{Block}_b(\mathsf{MCSP}[s]) \notin \mathsf{SCA}[nf(m)]$ *for every* $f \in \mathsf{poly}(m)$ *and* $b = O(f)$*, then* $\mathsf{P} \neq \mathsf{NP}$.

We now turn to the proof of Theorem 3.24, which follows [75] closely. First, we generalize blockwise reductions (see Definition 3.15) to search problems:

Definition 3.25 (Blockwise reducible (for search problems))**.** Let L and L' be block languages that correspond to search problems S and S', respectively. Also, for an instance x, let $S(x)$ (resp., $S'(x)$) denote the set of solutions for x under the problem S (resp., S'). Then L is said to be (k-)*blockwise reducible* to L' if there is a computable k-blockwise map $g\colon \mathfrak{B}_b^{km} \to \mathfrak{B}_b^m$ such that, for every $w \in \mathfrak{B}_b^{km}$, we have $S(w) = S'(g(w))$.

Notice Lemma 3.16 readily generalizes to blockwise reductions in this sense.

Next, we describe the set of problems that we reduce $\mathsf{Block}_b(\mathsf{MCSP}[s])$ to. Let $r\colon \mathbb{N}_+ \to \mathbb{N}_+$ be a function. There is a straightforward 1-blockwise reduction from $\mathsf{Block}_b(\mathsf{MCSP}[s])$ to (a suitable block version of) the following search problem $\mathsf{Merge}_r[s]$:

Given: the binary representation of $n \in \mathbb{N}_+$ and the respective descriptions (padded to length $\ell(n)$) of circuits $C_1, \ldots, C_r$, where $|C_i| \leq s(n)$ for every i and $r = r(n)$.

Find: (the description of) a circuit C with $|C| \leq s(n)$ and such that, for every i and every $x \in [(i-1)2^n/r, i2^n/r - 1]$, $C(x) = C_i(x)$; if no such C exists or $C_i = \bot$ for any i, answer with $\bot$.

In particular, for the reduction mentioned above, we shall use $r = 2^n$. Evidently, $\mathsf{Merge}_r[s]$ is a generalization of the problem $\mathsf{Merge}[s]$ defined previously and, more importantly, every instance of $\mathsf{Merge}_r[s]$ is simply a concatenation of $r/2$ many $\mathsf{Merge}[s]$ instances where α, β, and γ are given implicitly. Using the assumption that $\mathsf{Merge}[s]$ is computable by a CA in at most $f(m)$ time and $g(m)$ space, we can solve each such instance in parallel, thus producing an instance of $\mathsf{Merge}_{r/2}[s]$ (i.e., halving r). This yields a 2-blockwise reduction from (the respective block versions of) $\mathsf{Merge}_r[s]$ to $\mathsf{Merge}_{r/2}[s]$ (cnf. the proof of Proposition 3.21). Using Lemma 3.16 and that $\mathsf{Merge}_1[s]$ is trivial, we obtain the purported SCA for $\mathsf{Block}_b(\mathsf{MCSP}[s])$.

Proof. Let n be fixed, and let $r = 2^n$. First, we describe the 1-blockwise reduction from $\mathsf{Block}_b(\mathsf{MCSP}[s])$ to a block version of $\mathsf{Merge}_r[s]$ (which we shall describe along with the reduction). Let T_a denote the (description of the) trivial circuit that is constant $a \in \{0,1\}$, that is, $T_a(x) = a$ for every $x \in \{0,1\}^n$. Then we map each block

$$\begin{pmatrix} \mathrm{bin}_n(x) \\ y0^{b(n)-1} \end{pmatrix}$$

with $y \in \{0,1\}$ to the block

$$\begin{pmatrix} \mathrm{bin}_n(x) \\ T_y\pi \end{pmatrix},$$

where $\pi \in \{0\}^*$ is a padding string so that the block length $b(n)$ is preserved. (This is needed to ensure enough space is available for the construction; see the details further below.) It is evident this can be done in time $O(b(n))$ and (since we just translate the truth-table 0 and 1 entries to the respective trivial circuits) that the reduction is correct, that is, that every solution to the original $\mathsf{Block}_b(\mathsf{MCSP}[s])$ instance must also be a solution of the produced instance of (the resulting block version of) $\mathsf{Merge}_r[s]$ and vice-versa.

Next, maintaining the block representation described above, we construct the 2-blockwise reduction from the respective block versions of $\mathsf{Merge}_\rho[s]$ to $\mathsf{Merge}_{\rho/2}[s]$, where $\rho = 2^k$ for some $k \in [1,n]$. Let A denote the CA that, by assumption, computes a solution to an instance of $\mathsf{Merge}[s]$ in place in at most $f(m)$ time and $g(m)$ space. Then, for $j \in [0, \rho/2-1]$, we map each pair

$$\begin{pmatrix} \mathrm{bin}_n(2j) \\ C_0\pi_0 \end{pmatrix} \# \begin{pmatrix} \mathrm{bin}_n(2j+1) \\ C_1\pi_1 \end{pmatrix}$$

of blocks (where $\pi_0, \pi_1 \in \{0\}^*$ again are padding strings) to

$$\begin{pmatrix} \mathrm{bin}_n(j) \\ C\pi \end{pmatrix},$$

where $\pi \in \{0\}^*$ is a padding string (as above) and C is the circuit produced by A for $\alpha = 2j2^n/\rho$, $\beta = (2j+1)2^n/\rho$, and $\gamma = (2j+2)2^n/\rho$.

To actually execute A, we need $g(m)$ space (which is guaranteed by the block length $b(n)$) and, in addition, to prepare the input so it is in the format expected by A (i.e., eliminating the padding between the two circuit descriptions and writing the representations of α, β, and γ), which can be performed in $O(b(n)) \leq O(g(m)) \leq O(f(m))$ time. For the correctness, suppose the above reduces an instance of $\mathsf{Merge}_\rho[s]$ with circuits $C_1, \ldots, C_\rho$ to an instance of $\mathsf{Merge}_{\rho'}[s]$ with circuits $D_1, \ldots, D_{\rho'}$ (and no $\perp$ was produced), where $\rho' = \rho/2$. Then, a circuit E is a solution to the latter if and only if $E(x) = D_i(x)$ for every i and $x \in [(i-1)\sigma', i\sigma' - 1]$, where $\sigma' = 2^n/\rho'$. Using the definition of $\mathsf{Merge}[s]$, every D_i must satisfy $D_i(x) = C_{2i-1}(x)$ and $D_i(y) = C_{2i}(y)$ for $x \in [(2i-2)\sigma, (2i-1)\sigma - 1]$ and $y \in [(2i-1)\sigma, 2i\sigma - 1]$, where $\sigma = 2^n/\rho$. Hence, E agrees with $C_1, \ldots, C_\rho$ if and only if it agrees with $D_1, \ldots, D_{\rho'}$ (on the respective intervals).

Since $s(n) \geq n$ and $\mathsf{Merge}_1[s]$ is trivial (i.e., it can be accepted in $O(b(n))$ time), applying the generalization of Lemma 3.16 to blockwise reductions for search problems completes the proof. □

Comparison with [75]. We conclude this section with a comparison of our result and proof with [75]. The most evident difference between the statements of Theorems 3.3 and 3.24 and the related result from [75] (i.e., Theorem 3.2) is that our results concern CAs

(instead of Turing machines) and relate more explicitly to the time and space complexities of $\mathsf{Merge}[s]$; in particular, the choice of the block length is tightly related with the space complexity of computing $\mathsf{Merge}[s]$. As for the proof, notice that we only merge two circuits at a time, which makes for a smaller instance size m (of $\mathsf{Merge}[s]$); this not only simplifies the proof but also minimizes the resulting time complexity of the SCA (as $f(m)$ is then smaller). Also, in our case, we make no additional assumptions regarding the first reduction from $\mathsf{Block}_b(\mathsf{MCSP}[s])$ to $\mathsf{Merge}_r[s]$; in fact, this step can be performed unconditionally. Finally, we note that our proof renders all blockwise reductions explicit and the connection to the self-reductions of [3] more evident. Despite these simplifications, the argument extends to generalizations of MCSP with similar structure and instance size (e.g., MCSP in the setting of Boolean circuits with oracle gates as in [75] or MCSP for multi-output functions as in [61]).

3.6. Concluding Remarks

Proving SCA Lower Bounds for $\mathsf{MCSP}[s]$. Recalling the language L_1 from the proof of Theorem 3.5, consider the intersection $L_1[s] = L_1 \cap \mathsf{MCSP}[s]$. Evidently, $L_1[s]$ is comparable in hardness to $\mathsf{MCSP}[s]$ (e.g., it is solvable in polynomial time using a single adaptive query to $\mathsf{MCSP}[s]$). By adapting the construction from the proof of Theorem 3.24 so the SCA additionally checks the L_1 property at the end in $\mathsf{poly}(s(n))$ time (e.g., using the circuit C produced to check whether $C(x) = 1$ for $x = C(0) \cdots C(n-1)$), we can derive a hardness magnification result for $L_1[s]$ too: If $\mathsf{Block}_b(L_1[s]) \notin \mathsf{SCA}[\mathsf{poly}(s(n))]$ (for every $b \in \mathsf{poly}(s(n))$), then $\mathsf{P} \neq \mathsf{NP}$. Using the methods from Section 3.3.3 and that there are $2^{\Omega(s(n))}$ many (unique) circuits of size $s(n)$ or less,[4] this means that, if $\mathsf{Block}_b(L_1[s]) \in \mathsf{SCA}[t(n)]$ for some $b \in \mathsf{poly}(n)$ and $t\colon \mathbb{N}_+ \to \mathbb{N}_+$, then $t(n) = \Omega(s(n))$. Hence, for an eventual proof of $\mathsf{P} \neq \mathsf{NP}$ based on Theorem 3.3, one would need to develop new techniques (see also the discussion below) to raise this bound at the very least beyond $\mathsf{poly}(s(n))$.

Seen from another angle, this demonstrates that, although we can prove a tight SCA worst-case lower bound for L_1 (Theorem 3.5), establishing similar lower bounds on instances of L_1 with low circuit complexity (i.e., instances which are also in $\mathsf{MCSP}[s]$) is at least as hard as showing $\mathsf{P} \neq \mathsf{NP}$. In other words, it is straightforward to establish a lower bound for L_1 using arbitrary instances, but it is absolutely non-trivial to establish similar lower bounds for *easy* instances of L_1 where instance hardness is measured in terms of circuit complexity.

[4] Let $K > 0$ be constant such that every Boolean function on m variables admits a circuit of size at most $K2^m/m$. Setting $m = \lfloor \log s(n) \rfloor$, notice that, for sufficiently large n (and $s(n) = \omega(1)$ and $s(n) = O(2^n/n)$), this gives us $s(n) \geq K2^m/m$, thus implying that every Boolean function on $m \leq n$ variables admits a circuit of size at most $s(n)$. Since there are 2^{2^m} many such (unique) functions, it follows there are $2^{\Omega(s(n))}$ (unique) circuits of size at most $s(n)$.

The Proof of Theorem 3.24 and the Locality Barrier. In a recent paper [20], Chen et al. propose the concept of a *locality barrier* to explain why current lower bound proof techniques (for a variety of non-uniform computational models) do not suffice to show the lower bounds needed for separating complexity classes in conjunction with hardness magnification (i.e., in our case above a $\mathrm{poly}(s(n))$ lower bound that proves $\mathsf{P} \neq \mathsf{NP}$). In a nutshell, the barrier arises from proof techniques relativizing with respect to *local aspects* of the computational model at hand (in [20], concretely speaking, oracle gates of small fan-in), whereas it is known that a proof of $\mathsf{P} \neq \mathsf{NP}$ must not relativize [11].

The proof of Theorem 3.24 confirms the presence of such a barrier also in the uniform setting and concerning the separation of P from NP. Indeed, the proof mostly concerns the construction of an SCA where the overall computational paradigm of blockwise reductions (using Lemma 3.16) is unconditionally compatible with the SCA model (as exemplified in Proposition 3.21); the $\mathsf{P} = \mathsf{NP}$ assumption is needed exclusively so that the local algorithm for $\mathsf{Merge}[s]$ in the statement of the theorem exists. Hence, the result also holds *unconditionally* for SCAs that are, say, augmented with oracle access (in a plausible manner, e.g., by using an additional oracle query track and special oracle query states) to $\mathsf{Merge}[s]$. (Incidentally, the same argument also applies to the proof of the hardness magnification result for streaming algorithms (i.e., Theorem 3.2) in [75], which also builds on the existence of a similar locally computable function.) In particular, this means the lower bound techniques from the proof of Theorem 3.5 do not suffice since they extend to SCAs having oracle access to any computable function.

Open Questions. We conclude with a few open questions:

- By weakening SCAs in some aspect, certainly we can establish an unconditional MCSP lower bound for the weakened model which, were it to hold for SCAs, would imply the separation $\mathsf{P} \neq \mathsf{NP}$ (using Theorem 3.3). *What forms of weakening* (conceptually speaking) are needed for these lower bounds? How are these related to the locality barrier discussed above?
- Secondly, we saw SCAs are strictly more limited than streaming algorithms. Proceeding further in this direction, can we identify *further (natural) models of computation that are more restricted than SCAs* (whether CA-based or not) and for which we can prove results similar to Theorem 3.24?
- Finally, besides MCSP, what other (natural) problems admit similar SCA hardness magnification results? More importantly, can we identify some *essential property* of these problems that would explain these results? For instance, in the case of MCSP there appears to be some connection to the length of (minimal) witnesses being much smaller than the instance length. Indeed, one sufficient condition in this sense (disregarding SCAs) is sparsity [21]; nevertheless, it seems rather implausible that this would be the sole property responsible for all hardness magnification phenomena.

4. Sublinear-Time Probabilistic Cellular Automata

Preprint version: [84]

Abstract

We propose and investigate a probabilistic model of sublinear-time one-dimensional cellular automata. In particular, we modify the model of ACA (which are cellular automata that accept if and only if all cells simultaneously accept) so that every cell changes its state not only dependent on the states it sees in its neighborhood but also on an unbiased coin toss of its own. The resulting model is dubbed *probabilistic ACA* (PACA). We consider one- and two-sided error versions of the model (in the same spirit as the classes RP and BPP) and establish a separation between the classes of languages they can recognize all the way up to $o(\sqrt{n})$ time. We also prove that the derandomization of $T(n)$-time PACA (to polynomial-time deterministic cellular automata) for various regimes of $T(n) = \omega(\log n)$ implies non-trivial derandomization results for the class RP (e.g., P = RP). The main contribution is an almost full characterization of the constant-time PACA classes: For one-sided error, the class is equal to that of the deterministic model; that is, constant-time one-sided error PACA can be fully derandomized with only a constant multiplicative overhead in time complexity. As for two-sided error, we prove that the respective class is "sandwiched" in-between the class of strictly locally testable languages (SLT) and that of locally threshold testable languages (LTT).

4.1. Introduction

Cellular automata (CAs) have been extensively studied as a natural model of distributed computation. A one-dimensional CA is composed of a row of fairly limited computational agents—the *cells*—which, by interacting with their immediate neighbors, realize a global behavior and work towards a common goal.

As every model of computation, CAs have been widely studied as language acceptors [69, 111]. These efforts apparently were almost exclusively devoted to the linear- or real-time case—to the detriment of the *sublinear-time* one (as discussed in Chapter 2). This is unfortunate since, as shown in Chapter 3, the study of sublinear-time CA variants might help better direct efforts in resolving outstanding problems in computational complexity theory.

In this chapter, we consider a *probabilistic* sublinear-time CA model. Our main goal is to analyze to what extent—if at all—the addition of randomness to the model is able to make up for inherent limitations of it. (For instance, sublinear-time CA models are usually restricted to a local view of their input and are also unable to cope with long unary subwords; see Chapters 2 and 3, in particular Lemmas 2.9, 2.11 and 3.17.) With our results we show yet another connection between randomness and counting—as has been observed in the past in multiple areas of complexity theory (for various examples thereof, see [5, 50]).

4.1.1. The Model

The usual acceptance condition for CA-based language recognizers is that of a distinguished cell (usually the leftmost one) entering an accepting state [69]. This is unsuitable for sublinear-time computation since then the automaton is limited to verifying prefixes of a constant length (as mentioned in Chapter 2). The most widely studied acceptance condition for sublinear-time (see [60, 66, 105] as well as Chapter 2) is that of all cells simultaneously accepting, yielding the model of *ACA* (where the first "A" in the acronym indicates that *all* cells must accept).

We propose a probabilistic version of the ACA model inspired by the stochastic automata of [7] and the definition of probabilistic Turing machines (see, e.g., [5]). In the model of *probabilistic ACA* (PACA), at every step, each cell tosses a fair coin $c \in \{0, 1\}$ and then changes its state based on the outcome of c. There is a nice interplay between this form of randomness access and the overall theme of *locality* in CAs: Random events pertaining to a cell i depend exclusively on what occurs in the vicinity of i, while those Furthermore, events corresponding to distinct cells i and j can only be dependent if i and j are near each other; otherwise, they are necessarily independent (Lemma 4.11). We consider both one- and two-sided error versions of the model (as natural counterparts to their Turing machine analogues).

Although a PACA is a conceptually simple extension of an ACA, the definition requires certain care. In particular, an offhand realization runs into trouble when defining the model's time complexity. To see why, recall that, in deterministic ACA (DACA),[1] time complexity of an automaton C is defined as the upper bound on the number of steps that C takes to accept an input in its language $L(C)$. In contrast, in a PACA C' there may be multiple computational branches (depending on the cells' coin tosses) for the same input $x \in L(C')$, and it may be the case that there is no upper bound on the number of steps for a branch starting at x to reach an accepting configuration. In non-distributed models such as Turing machines, these pathological cases can be dealt with by counting the number of steps computed and stopping if this exceeds a certain bound. In PACA, that would either require an extrinsic global agent that informs the cells when this is the case (which is undesirable since we would like a strictly distributed model) or it would need to be

[1] not to be confused with the *decider ACA* of Chapter 2

handled by the cells themselves, which is impossible in sublinear-time in general (since the cells cannot directly determine the input length). See Section 4.3 for further discussion.

Finally, we should also mention our model is more restricted than a *stochastic CA*,[2] which is a CA in which the next state of a cell is chosen according to an arbitrary distribution that depends on the cell's local configuration (remitting to the probabilistic finite automata of [97]). For a survey on stochastic CAs, see [73].

4.1.2. Results

Inclusion relations. As can be expected, two-sided error PACA are more powerful than their one-sided error counterparts. (Refer to Definition 4.9 for the precise definitions of one- and two-sided error PACA.) Say a DACA C is *equivalent* to a PACA C' if they accept the same language (i.e., $L(C) = L(C')$). We are able to prove this all the way up to $o(\sqrt{n})$ time:

Theorem 4.1. *The following hold:*

1. *If C is a one-sided error PACA with time complexity T, then there is an equivalent two-sided error PACA C' with time complexity $O(T)$.*
2. *There is a language L recognizable by constant-time two-sided error PACA but not by any $o(\sqrt{n})$-time one-sided error PACA.*

We stress the first item does not follow immediately from the definitions since it requires error reduction by a constant factor, which requires a non-trivial construction. It remains open whether in the second item we can improve the separation all the way up to $o(n)$ time.

Another result we show is how time-efficient derandomization of PACA classes imply derandomization results for RP (with a trade-off between the PACA time complexity and the efficiency of the derandomization).

Theorem 4.2. *Let $d \geq 1$. The following hold:*

- *If there is $\varepsilon > 0$ such that every n^ε-time (one- or two-sided error) PACA can be converted into an equivalent n^d-time deterministic CA, then* $\mathsf{P} = \mathsf{RP}$.
- *If every* $\mathrm{polylog}(n)$*-time PACA can be converted into an equivalent n^d-time deterministic CA, then, for every $\varepsilon > 0$,* $\mathsf{RP} \subseteq \mathsf{TIME}[2^{n^\varepsilon}]$.
- *If there is $b > 2$ so that any $(\log n)^b$-time PACA can be converted into an equivalent n^d-time deterministic CA, then, for every $a \geq 1$ and $\alpha > a/(b-1)$,* $\mathsf{RPTIME}[n^a] \subseteq \mathsf{TIME}[2^{O(n^\alpha)}]$.

[2] Unfortunately, the literature uses the terms stochastic and probabilistic CA interchangeably. We deem "probabilistic" more suitable since it is intended as a CA version of a probabilistic Turing machine.

Note we deliberately write "deterministic CA" instead of "DACA" since, for $T(n) = \Omega(n)$, a T-time DACA is equivalent to an $O(T)$-time deterministic CA with the usual acceptance condition (as mentioned in Chapter 2).

Characterization of constant time. As the first step towards a deeper study of sublinear-time PACA, we analyze and almost completely characterize constant-time PACA. The constant-time case is already very rich and worth considering in and of itself. This may not come as a surprise since other models of distributed computing (e.g., local graph algorithms [109]) also exhibit behavior in the constant-time case that is far from trivial.

In Section 4.3 we give an example of a one-sided error PACA that recognizes a language L strictly faster than any DACA for L. Nonetheless, we prove that one-sided error PACA can be derandomized with only a constant multiplicative overhead in time complexity.

Theorem 4.3. *For any constant-time one-sided error PACA C, there is a constant-time DACA C' such that $L(C) = L(C')$.*

In turn, the class of languages accepted by constant-time two-sided error PACA can be considerably narrowed down in terms of a novel subregular class LLT, dubbed the *locally linearly testable* languages. Below, $\mathsf{LLT}_{\cup\cap}$ is the closure over LLT under union and intersection and LTT its Boolean closure (i.e., its closure under union, intersection, and complement).

Theorem 4.4. *The class of languages that can be accepted by a constant-time two-sided error PACA contains $\mathsf{LLT}_{\cup\cap}$ and is strictly contained in LTT.*

It is known that the constant-time class of DACA equals the closure under union $\mathsf{SLT}_{\cup}$ of the strictly local languages SLT [105]. (We refer to Section 4.4.2.2 for the definitions.) Since $\mathsf{SLT}_{\cup} \subsetneq \mathsf{LLT}_{\cup}$ is a proper inclusion, this gives a separation of the deterministic and probabilistic classes in the case of two-sided error and starkly contrasts with Theorem 4.3.

The class LLT. As far as we are aware of, the class LLT does not previously appear in the literature. In Section 4.4.2.2 we show LLT lies in-between $\mathsf{SLT}_{\cup}$ and the class of locally threshold testable languages LTT. In this regard LLT is similar to the class LT of locally testable languages; however, as we also prove, both LLT and $\mathsf{LLT}_{\cup}$ are incomparable to LT. The relation between $\mathsf{LLT}_{\cup\cap}$ and LT is left as a topic for future work.

The languages in LLT are defined based on sets of allowed prefixes and suffixes (as, e.g., the languages in SLT) together with a *linear threshold* condition (hence their name): For the infixes m of a fixed length $\ell \in \mathbb{N}_+$ there are coefficients $\alpha(m) \in \mathbb{R}_0^+$ as well as a threshold $\theta \geq 0$ such that every word w in the language satisfies the following:

$$\sum_{m \in \Sigma^\ell} \alpha(m) \cdot |w|_m \leq \theta,$$

where $|w|_m$ is the number of occurrences of m in w.

As the classes SLT, LT, and LTT (see, e.g., [14]), LLT may also be characterized in terms of *scanners*, that is, memoryless devices that process their input by passing a sliding window of ℓ symbols over it. Namely, the class LLT corresponds to the languages that can be recognized by scanners possessing a *single counter* c with maximum value θ; the counter c is incremented by $\alpha(m)$ for every infix $m \in \Sigma^\ell$ read, and the scanner accepts if and only if $c \leq \theta$ holds at the end of the input (and the prefix and suffix of the input are also allowed).

A related restriction of the LTT languages that we should mention is that of the locally threshold testable languages in the strict sense (LTTSS) [48, 102]. The key difference between these languages and our class LLT is that, in the former, one sets a threshold condition for each infix separately (which corresponds to using multiple counters in their characterization in terms of scanners). In turn, in LLT there is a *single* threshold condition (i.e., the inequality above) and in which different infixes may have distinct weights (i.e., the coefficients $\alpha(m)$). For instance, this allows counting distinct infixes m_1 and m_2 towards the same threshold t, which is not possible in the LTTSS languages (as there each infix is considered separately).

4.1.3. Further Directions

LLT and two-sided error PACA. Besides giving a separation between one- and two-sided error, Theorem 4.4 considerably narrows down the position of the class of languages accepted by constant-time two-sided error PACA. Nevertheless, even though we now know the class is "sandwiched" in-between $\mathsf{LLT}_{\cup\cap}$ and LTT, a full characterization remains outstanding. Generalizing the proof of Theorem 4.4 is challenging because the strategy we follow in the proof relies on closure under complement, but (as we also prove) the class of two-sided error PACA is *not* closed under complement. It appears that clarifying the relation between it and $\mathsf{LLT}_{\cup\cap}$ as well as $\mathsf{LLT}_{\cup\cap}$ itself and $\mathsf{LLT}_{\cup}$ or also LT may give some "hints" on how to proceed.

The general sublinear-time case. Theorem 4.2 indicates that even polylogarithmic-time PACA can recognize languages for which no deterministic polynomial-time algorithm is currently known. Although the proof of Proposition 4.22 does yield explicit examples of such languages, they are rather unsatisfactory since the construction does not seem to rely on the full capabilities of the PACA model. (Namely, communication between blocks of cells is only required to check certain syntactic properties of the input; once this is done, the blocks operate independently from one another.) A promising next step would be to identify languages where the capabilities of the PACA model are put to more extensive use.

Pseudorandom generators. From the opposite direction, to investigate the limitations of the PACA model, one possibility would be to construct *pseudorandom generators* (PRGs) that fool sublinear-time PACA. Informally, such a PRG is a function $G\colon \{0,1\}^{s(n)} \to \{0,1\}^{r(n)}$ with $s(n) \ll r(n)$ and having the property that PACA (under given time constraints) are incapable of distinguishing $G(x)$ from uniform when the seed x is chosen uniformly at random. PRGs have found several applications in complexity theory (see, e.g., [113] for an introduction).

Theorem 4.2 indicates that unconditional time-efficient derandomization of PACA is beyond reach of current techniques, so perhaps *space-efficient* derandomization should be considered instead. Indeed, as a PACA can be simulated by a space-efficient machine (e.g., by adapting the algorithm from Theorem 3.4), it is possible to recast PRGs that fool space-bounded machines (e.g., [59, 91]) as PRGs that fool PACA. Nevertheless, we expect to obtain even better constructions by exploiting the locality of PACA (which space-bounded machines do not suffer from).

4.1.4. Organization

The rest of the chapter is organized as follows: Section 4.2 introduces basic concepts and notation and defines the underlying CA and ACA models. Following that, in Section 4.3 we introduce PACA and prove standard error reduction results as well as Theorem 4.1. In Section 4.4 we address the constant-time case and prove Theorems 4.3 and 4.4. Finally, in Section 4.5 we briefly address the general sublinear-time case and prove Theorem 4.2.

4.2. Preliminaries

It is assumed the reader is familiar with the theory of cellular automata as well as with basic notions of computational complexity theory (see, e.g., the standard references [5, 30, 50]).

All logarithms are to base 2. The set of integers is denoted by $\mathbb{Z}$, that of non-negative integers by $\mathbb{N}_0$, and that of positive integers by $\mathbb{N}_+$. For a set S and $n, m \in \mathbb{N}_+$, $S^{n\times m}$ is the set of n-row, m-column matrices over S. For $n \in \mathbb{N}_+$,

$$[n] = \{i \in \mathbb{N}_0 \mid i < n\}$$

is the set of the first n non-negative integers. Also, for $a, b \in \mathbb{Z}$, by

$$[a, b] = \{i \in \mathbb{Z} \mid a \le i \le b\}$$

we always refer to an interval containing only integers.

Symbols in words are indexed starting with zero. The i-th symbol of a word w is denoted by w_i. For an alphabet Σ and $n \in \mathbb{N}_0$, $\Sigma^{\le n}$ contains the words $w \in \Sigma^*$ for which $|w| \le n$. For an infix $m \in \Sigma^{\le |w|}$ of w, $|w|_m$ is the number of occurrences of m in w. Without restriction,

the empty word is not an element of any language that we consider. (This is needed for definitional reasons; see Definitions 4.6 and 4.7 below.) We write U_n (resp., $U_{n\times m}$) for a random variable distributed uniformly over $\{0,1\}^n$ (resp., $\{0,1\}^{n\times m}$).

Many of our low-level arguments make use of the notion of a *lightcone*.[3] For a set S and non-negative integers $n \le m$, a lightcone $L = (\ell_{i,j})$ of *radius* m and *height* n over S is a trapezoidal (when $n < m$) or triangular (when $n = m$) array of elements $\ell_{i,j} \in S$, where $i \in [0,n]$ and $j \in [-m,m]$:

$$\begin{array}{ccccccccccc}
\ell_{0,-m} & \ell_{0,-m+1} & \cdots & \cdots & \cdots & \ell_{0,0} & \cdots & \cdots & \cdots & \ell_{0,m-1} & \ell_{0,m} \\
 & \ell_{1,-m+1} & \cdots & \cdots & \cdots & \ell_{1,0} & \cdots & \cdots & \cdots & \ell_{1,m-1} & \\
 & & \ddots & & & \vdots & & & \iddots & & \\
 & & & \ell_{n,-m+n} & \cdots & \ell_{n,0} & \cdots & \ell_{n,m-n} & & &
\end{array}$$

The element $\ell_{0,0}$ is the *center* of the lightcone. The *layers* of L are indexed by i, where the i-th layer contains $2(m-i)+1$ elements. Hence, the top layer contains $2m+1$ elements and the bottom one $2(m-n)+1$; in particular, the bottom layer is a single element if and only if $n = m$. There are

$$\sum_{i=0}^{n}(2(m-i)+1) = (n+1)(2m-n+1)$$

elements in a lightcone in total.

We shall also need the following variant of the Chernoff bound (see, e.g., [113]):

Theorem 4.5 (Chernoff bound)**.** *Let $X_1,\dots,X_n$ be independently and identically distributed Bernoulli variables and $\mu = \mathrm{E}[X_i]$. There is a constant $c > 0$ such that the following holds for every $\varepsilon = \varepsilon(n) > 0$:*

$$\Pr\left[\left|\frac{\sum_i X_i}{n} - \mu\right| > \varepsilon\right] < 2^{-cn\varepsilon^2}.$$

4.2.1. Cellular Automata

We consider only bounded one-dimensional cellular automata.

Definition 4.6 (Cellular automaton)**.** A *cellular automaton* (CA) is a triple $C = (Q, \$, \delta)$ where Q is the finite set of *states*, $\$ \notin Q$ is the *boundary symbol*, and $\delta\colon Q_\$ \times Q \times Q_\$ \to Q$ is the *local transition function*, where $Q_\$ = Q \cup \{\$\}$. The elements in the domain of δ are the possible *local configurations* of the cells of C. For a fixed width $n \in \mathbb{N}_+$, the *global configurations* of C are the elements of Q^n. The cells 0 and $n-1$ are the *border cells* of C.

[3] Some sources distinguish between *future* and *past* lightcones. Here we shall need only past lightcones.

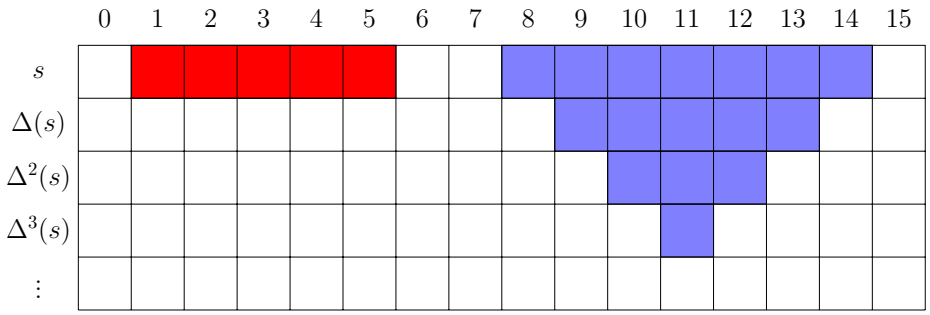


Figure 4.1.: Space-time diagram of a CA with 16 cells for an initial configuration s. (The states have been omitted for simplicity.) The cells marked in red form the 2-neighborhood of cell 3, the ones in blue the 3-lightcone of cell number 11.

The *global transition function* $\Delta\colon Q^n \to Q^n$ is obtained by simultaneous application of δ everywhere; that is, if $s \in Q^n$ is the current global configuration of C, then

$$\Delta(s) = \delta(\$, s_0, s_1)\,\delta(s_0, s_1, s_2)\,\cdots\,\delta(s_{n-2}, s_{n-1}, \$).$$

For $t \in \mathbb{N}_0$, Δ^t denotes the t-th iterate of Δ. For an initial configuration $s \in Q^n$, the sequence $s = \Delta^0(s), \Delta(s), \Delta^2(s), \ldots$ is the *orbit* of C (for s). Writing the orbit of C line for line yields its *space-time diagram*. For $i \in [n]$ and $r \in \mathbb{N}_0$, the interval $[i-r, i+r] \cap [n]$ forms the *r-neighborhood* of i; for $t \in \mathbb{N}_0$, the *t-lightcone* of i is the lightcone of radius and height t centered at i in the 0-th row (i.e., the initial configuration) of the space-time diagram of C.[4]

Definition 4.7 (DACA)**.** A *DACA* is a CA C with an *input alphabet* $\Sigma \subseteq Q$ as well as a subset $A \subseteq Q$ of *accepting states*. We say C *accepts* an input $x \in \Sigma^+$ if there is $t \in \mathbb{N}_0$ such that $\Delta^t(x) \in A^n$, and we denote the set of all such x by $L(C)$. In addition, C is said to have *time complexity* (bounded by) $T\colon \mathbb{N}_+ \to \mathbb{N}_0$ if, for every $x \in L(C) \cap \Sigma^n$, there is $t < T(|x|)$ such that $\Delta^t(x) \in A^n$.

4.3. Fundamentals

In this section, we introduce the definition of PACA. Following that, we prove basic error reduction results and conclude with the proof of Theorem 4.1.

As is customary for randomized models of computation, there are both online and offline views of our model. Since it gives a more natural presentation, in the definition below we first present the online model and then address the definitional issue mentioned in the introduction. In the last part, we switch to the offline view of the model, which will be more comfortable to work with since we can then refer to the cells' coin tosses explicitly.

[4] If the lightcone's dimensions overstep the boundaries of the space-time diagram (i.e., i is too close to either of the borders of C (e.g., $i < t$)), then some cells in the t-lightcone will have undefined states. In this case, we set the undefined states to \$, which ensures consistency with δ.

Definition 4.8 (PACA)**.** Let Σ be an alphabet and Q a finite set of states with $\Sigma \subseteq Q$. A *probabilistic ACA* (PACA) C is a CA with two local transition functions $\delta_0, \delta_1 \colon Q^3 \to Q$. At each step of C, each cell tosses a fair coin $c \in \{0,1\}$ and updates its state according to δ_c; that is, if the current configuration of C is $s \in Q^n$ and the result of the cells' coin tosses is $r = r_0 \cdots r_{n-1} \in \{0,1\}^n$ (where r_i is the coin toss of the i-th cell), then the next configuration of C is

$$\Delta_r(s) = \delta_{r_0}(\$, s_0, s_1)\, \delta_{r_1}(s_0, s_1, s_2) \cdots \delta_{r_{n-1}}(s_{n-2}, s_{n-1}, \$).$$

Seeing this process as a Markov chain M over Q^n, we recast the global transition function $\Delta = \Delta_{U_n}$ as a family of random variables $(\Delta(s))_{s \in Q^n}$ parameterized by the current configuration s of C, where $\Delta(s)$ is sampled by starting in state s and performing a single transition on M (having drawn the cells' coin tosses according to U_n). Similarly, for $t \in \mathbb{N}_0$, $\Delta^t(s)$ is sampled by starting in s and performing t transitions on M.

A *computation* of C for an input $x \in \Sigma^n$ is a path in M starting at x. The computation is *accepting* if the path visits A^n at least once. In addition, in order to be able to quantify the probability of a PACA accepting an input, we additionally require for every PACA C that there is a function $T \colon \mathbb{N}_+ \to \mathbb{N}_0$ such that, for any input $x \in \Sigma^n$, every accepting computation for x visits A^n for the first time in strictly less than $T(n)$ steps; that is, if there is $t \in \mathbb{N}_0$ with $\Delta^t(x) \in A^n$, then $\Delta^{t_1}(x) \in A^n$ for some $t_1 < T(n)$. (Hence, every accepting computation for x has an initial segment with endpoint in A^n and whose length is strictly less than $T(n)$.) If this is the case for any such T, then we say C has *time complexity* (bounded by) T.

With this restriction in place, we may now equivalently replace the coin tosses of C with a matrix $R \in \{0,1\}^{T(n) \times n}$ of bits with rows $R_0, \ldots, R_{T(n)-1}$ and such that $R_j(i)$ corresponds to the coin toss of the i-th cell in step j. (If C accepts in step t, then the coin tosses in rows $t, \ldots, T(n) - 1$ are ignored.) We refer to R as a *random input* to C. Blurring the distinction between the two perspectives (i.e., online and offline randomness), we write $C(x, R) = 1$ if C accepts x when its coin tosses are set according to R, or $C(x, R) = 0$ otherwise.

Observe that, for different choices of T, the random input matrix R also has a different number of rows. This is not an issue since (as required above) any superficial rows are ignored by C; that is, without restriction we may take T to be such that every value $T(n)$ is minimal and set the number of rows of R to $T(n)$. The reason for letting R be larger is that, when simulating a PACA, it may be the case that it is more convenient to compute only an upper bound $T'(n) \geq T(n)$ instead of the actual minimal value $T(n)$. By the convention above then, it does not matter if R has $T(n)$ or $T'(n)$ rows (as the behavior of C is the same).

As another remark, notice that in Definition 4.8 we opt for using binary coin tosses along with only two local transition functions. Nonetheless, this is sufficient to realize a set of 2^k local transition functions $\delta_0, \ldots, \delta_{2^k-1}$ for constant k with a multiplicative overhead of k. (Namely, by having each cell collect k coins in k steps, interpret these as the binary representation of $i \in [2^k]$, and then change its state according to δ_i.)

Definition 4.8 states the acceptance condition for a *single* computation (i.e., one fixed choice of a random input); however, we must still define acceptance based on *all* computations (i.e., for random inputs picked according to a uniform distribution). The two most natural candidates are the analogues of the well-studied classes RP and BPP, which we define next.

Definition 4.9 (p-error PACA)**.** Let $L \subseteq \Sigma^*$ and $p \in [0, 1)$. A *one-sided p-error PACA for L* is a PACA C with time complexity $T = T(n)$ such that, for every $x \in \Sigma^n$,

$$x \in L \iff \Pr[C(x, U_{T\times n}) = 1] \geq 1 - p \quad \text{and} \quad x \notin L \iff \Pr[C(x, U_{T\times n}) = 1] = 0.$$

If $p = 1/2$, then we simply say C is a *one-sided error PACA*. Similarly, for $p < 1/2$, a *two-sided p-error PACA for L* is a PACA C with time complexity $T = T(n)$ for which

$$x \in L \iff \Pr[C(x, U_{T\times n}) = 1] \geq 1 - p \quad \text{and} \quad x \notin L \iff \Pr[C(x, U_{T\times n}) = 1] \leq p$$

hold for every $x \in \Sigma^*$. If $p = 1/3$, then we simply say C is a *two-sided error PACA*. In both cases, we write $L(C) = L$ and say C *accepts* L.

Note that, to each 0-error PACA C, one can obtain an equivalent DACA C' with the same time complexity by setting the local transition function to δ_0.

In the rest of the chapter, if it is not specified which of the two variants above (i.e., one- or two-sided error) is meant, then we mean both variants collectively.

The next example shows that, in some cases, PACAs can be more efficient than DACAs.

Example 4.10. Let $\Sigma = \{0, 1, 2, 3\}$ and consider the language

$$L = \{0^k 1^l 2^m 3^n \mid k, l, m, n \in \mathbb{N}_0 \text{ and } ((l \geq 2 \text{ and } m \geq 3) \text{ or } (l \geq 3 \text{ and } m \geq 2))\}.$$

There is no DACA C that accepts L in at most 2 steps. This can be shown using methods from Chapter 2 as well as [66, 105]. Namely, given a DACA C with time complexity at most 2 we can fully determine if C accepts a word $x \in \Sigma^*$ by looking only at the infixes of length 5 (and the prefix and suffix of length 4) of x. Consider the words $x = 0^5 1^3 2^2 3^5 \in L$, $y = 0^5 1^2 2^3 3^5 \in L$, and $z = 0^5 1^2 2^2 3^5 \notin L$. (See Figure 4.2.) Then every infix of length 5 (and the prefix and suffix of length 4) of z appears in x except for the infixes 00112 and 01122, which both appear in y. Hence, it follows that, if C accepts x and y, it must also accept z, which proves there is no DACA for L with time complexity (at most) 3.

Nevertheless, there is a 3-time one-sided 7/8-error PACA C' for L. Checking that the input $x = 0^k 1^l 2^m 3^n$ is such that (x is of the form $0^*1^*2^*3^*$ and) $l, m \geq 2$ can be done without need of randomness simply by analyzing the infixes of length 5. (Again, we refer to Chapter 2 and [66, 105] for the general method.) Now we show how to use randomness to check the additional property that $m \geq 3$ or $l \geq 3$ hold. In time step 1, we have every cell of C' expose its coin toss of step 0 so that any of its neighbors can read it and use it to choose their state in step 2. Let c_σ denote the leftmost cell in C' in which $\sigma \in \Sigma$ appears, and let l_σ

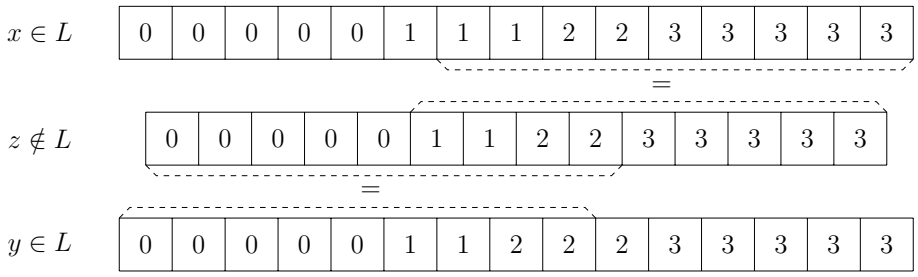


Figure 4.2.: Comparing the words $x = 0^5 1^3 2^2 3^5 \in L$, $y = 0^5 1^2 2^3 3^5 \in L$, and $z = 0^5 1^2 2^2 3^5 \notin L$, we notice that every infix of length 5 of z appears in either x or y. This implies there is no DACA that accepts L with time complexity 3 or less.

and r_σ be the coin tosses of the left and right neighbors of c_σ, respectively. We have c_1 accept if and only if $r_1 = 1$, c_3 if and only if $l_3 = 1$, and c_2 if and only if $l_2 + r_2 < 2$. All other cells of C' accept regardless of the coin tosses they see (as long as x satisfies the conditions we specified above).

For $i \in \{1, 2, 3\}$, let A_i denote the event of cell c_i accepting. The above results in the following behavior: If $l = m = 2$, we have $r_1 = l_2$ and $r_2 = l_3$ since the coin tosses belong to the same cells, in which case C' never accepts. If $l \geq 3$ and r_2 and l_3 belong to the same cell (i.e., $r_2 = l_3$), then r_1 and l_2 do not belong to the same cell and we have

$$\Pr[C(x, U_{T\times|x|}) = 1] = \Pr[A_1] \Pr[A_2 \wedge A_3] = \Pr[r_1 = 1] \Pr[l_2 = 0 \wedge r_2 = l_3 = 1] = \frac{1}{8}.$$

The case $m \geq 3$ and r_1 and l_2 belonging to the same cell is similar. Finally, if $l \geq 3$ and $m \geq 3$, the values r_1, l_2, r_2, and l_3 are all independent and we have

$$\Pr[C(x, U_{T\times|x|}) = 1] = \prod_{i=1}^{3} \Pr[A_i] = \Pr[r_1 = 1] \Pr[l_2 + r_2 < 2] \Pr[l_3 = 1] > \frac{1}{8}.$$

Lemma 4.11 (Independence of local events)**.** *Let C be a one- or two-sided error PACA, let $x \in \Sigma^n$ be an input to C, and let $T \in \mathbb{N}_+$. In addition, let $i, j \in [n]$ be such that $|i-j| > 2(T-1)$ and E_i (resp., E_j) be an event described exclusively by the states of the i-th (resp., j-th) cell of C in the time steps $0, \ldots, T-1$ (e.g., the i-th cell accepts in some step t where $t < T$). Then E_i and E_j are independent.*

Proof. For any random input R, the states of $k \in \{i, j\}$ in the time steps between 0 and $T-1$ is uniquely determined by the values of $R(t, k-T+t+1), \ldots, R(t, k+T-t-1)$ for $t \in [T]$. Without loss of generality, suppose $i \leq j$. Since $i + T - 1 < j - T + 1$, E_i and E_j are conditioned on disjoint sets of values of R, thus implying independence. □

Note the proof still holds in case $T = 1$, in which case the events E_i and E_j occur with probability either 0 or 1, thus also (trivially) implying independence.

4.3.1. Robustness of Definition

We now prove that the definition of PACA is robust with respect to the choice of $p = 1/2$ (resp., $p = 1/3$) for the error of one-sided (resp., two-sided) error PACA.

4.3.1.1. One-Sided Error

For one-sided error, we can reduce the error p to any desired constant value p'.

Proposition 4.12. *Let $p, p' \in (0, 1)$ be constant and $p' < p$. For every one-sided p-error PACA C, there is a one-sided p'-error PACA C' such that $L(C) = L(C')$. Furthermore, if C has time complexity $T(n)$, then C' has time complexity $O(T(n))$.*

Setting $p' = 1/2$, it follows that the definition of PACA is robust under the choice of p (as long as it is constant) and regardless of the time complexity (up to constant factors).

The proof is essentially a generalization of the idea from Proposition 2.7 to show that the sublinear-time DACA classes are closed under union. Namely, C' simulates several copies $C_0, \dots, C_{m-1}$ of C in parallel and accepting if and only if at least one C_i accepts. This idea is particularly elegant because m can be chosen to be constant and we update the C_i in a round-robin fashion (i.e., first C_0, then C_1, C_2, etc., and finally C_0 again after C_{m-1}). The alternative is to simulate each C_i for $T(n)$ steps at a time, which is not possible in general since it requires computing $T(n)$ first. The construction we give avoids this issue entirely.

Proof. We construct a PACA C' with the desired properties. Let $m = \lceil \log(1/p' - 1/p) \rceil$. Furthermore, let Q be the state set of C and Σ its input alphabet. We set the state set of C' to $Q^m \times [m] \cup \Sigma$. Given an input x, every cell of C' initially changes its state from $x(i)$ to $(x(i), \dots, x(i), 0)$. The cells of C' simulate m copies of C as follows: If the last component of a cell contains the value j, then its j-th component[5] q_0 is updated to $\delta(q_{-1}, q_0, q_1)$, where q_{-1} and q_1 are the j-th components of the left and right neighbors, respectively (or \$ in case of a border cell); at the same time, the last component of the cell is set to $j+1$ if $j < m$ or 0 in case $j = m$. A cell of C' is accepting if and only if its last component is equal to j and its j-th component is an accepting state of C.

Denote the i-th simulated copy of C by C_i. Clearly, C' accepts in step $mt + i + 1$ for $i \in [m]$ if and only if C_i accepts in step t, so we immediately have that C' has $O(T(n))$ time complexity. For the same reason and since C' never accepts in step 0, C' does not accept any input $x \notin L(C)$. As for $x \in L(C)$, note the m copies of C are all simulated using independent coin tosses, thus implying

$$\Pr[C' \text{ does not accept } x] = \Pr[\forall i \in [m] : C_i \text{ does not accept } x] < p^m \leq p'.$$

Hence, C' accepts x with probability at least $1 - p'$, as desired. □

[5] In the same manner as we do for the indices of a word, we number the components starting with zero.

4.3.1.2. Two-Sided Error

For two-sided error, we show the definition of PACA is robust for every choice of p for *constant-time* PACA. The construction is considerably more complex than the previous one.

Proposition 4.13. *Let $p, p' \in (0, 1/2)$ be constant and $p' < p$. For every two-sided p-error PACA C with constant time complexity $T = O(1)$, there is a two-sided p'-error PACA C' with time complexity $O(T) = O(1)$ and such that $L(C) = L(C')$.*

To reduce the error, we use the standard method based on the Chernoff bound (Theorem 4.5); that is, the PACA C' simulates m independent copies $C_0, \dots, C_{m-1}$ of C (for an adequate choice of m) and then accepts if and only if the majority of the C_i do. More precisely, C' loops over every possible majority $\mathcal{M} \subseteq [m]$ (i.e., every set $\mathcal{M} \subseteq [m]$ with $|\mathcal{M}| \geq m/2$) over the C_i and checks whether C_i accepts for every $i \in \mathcal{M}$ (thus reducing majority over the $L(C_i)$ to intersection over the $L(C_j)$ where $j \in \mathcal{M}$). In turn, to check whether every C_j accepts for $j \in \mathcal{M}$, C' tries every possible combination of time steps for the C_j to accept and accepts if such a combination is found. If this entire process fails, then the majority of the C_i do not accept, and thus C' does not accept as well. (Obviously, this idea is only feasible if m as well as the time complexities of the C_i are constant.)

Proof. Let Q be the state set of C and Σ its input alphabet. We also fix a constant m depending only on p and p' which will be set later and let $M = \binom{m}{\lceil m/2 \rceil}$.

Construction. The state set of C' is $Q' \cup \Sigma$, where Q' is a set of states consisting of the following components:

- an m-tuple from Q^m of states of C
- an m-tuple from $[T]^m$ representing an m-digit, T-ary counter $i_0 \cdots i_{m-1}$
- a value from $[M]$ representing a counter modulo M
- an input symbol from Σ
- a $(T \times m)$-matrix of random bits, which are all initially set to an undefined value different from 0 or 1

Given an input x, in the first step of C' every cell changes its state from $x(i)$ to a state in Q' where the Q^m components are all set to $x(i)$, the numeric ones (i.e., with values in $[T]^m$ and $[M]$) set to 0, and the value $x(i)$ is stored in the Σ component. In the first phase of C', which lasts for mT steps, the cells fill their $(T \times m)$-matrix with random bits. This is the only part of the operation of C' in which its random input is used (i.e., in all subsequent steps, C' operates deterministically and the outcome of the remaining coin tosses is ignored).

In this next phase, C' simulates m copies $C_0, \dots, C_{m-1}$ of C in its Q^m components (as in the proof of Proposition 4.12). The random bits for the simulation are taken from the previously filled $(T \times m)$-matrix, where the T entries in the i-th column are used as the coin tosses in the simulation of C_i (with the entry in the respective j-th row being used in the j-th step of the simulation). Meanwhile, the $i_0 \cdots i_{m-1}$ counter is taken to represent that the simulation of C_j is in its i_j-th step.

The cells update their states as follows: At each step, the counter is incremented and the respective simulations are updated accordingly; more precisely, if C_j is in step i_j and i_j was incremented (as a result of the counter being incremented), then the simulation of C_j is advanced by one step; if the value i_j is reset to 0, then the simulation of C_j is restarted by setting the respective state to $x(i)$. Every time the counter has looped over all possible values, the $[M]$ component of the cell is incremented (and the process begins anew with the counter set to all zeroes). When the value of the $[M]$ component is equal to $M-1$ and the counter reaches its final value (i.e., $i_j = T-1$ for every j), then the cell conserves its current state indefinitely.

We agree upon an enumeration $\mathcal{M}_0, \dots, \mathcal{M}_{M-1}$ of the subsets of $[m]$ of size $\lceil m/2 \rceil$ and identify a value of i in the $[M]$ component with $\mathcal{M}_i$. A cell whose $[M]$ component is equal to i is then accepting if and only if, for every $j \in \mathcal{M}_i$, its j-th component is an accepting state of C.

Correctness. By construction, if C' accepts in a time step where the T-ary counters have the value $i_0 \cdots i_{m-1}$ and the $[M]$ component the value j, then this is the case if and only if C_k accepts in step i_k for every $k \in \mathcal{M}_j$. Hence, C' accepts if and only if at least $\lceil m/2 \rceil$ of the simulated copies of C accept (which, by definition, must occur in a time step prior to T); that is, C' accepts if and only if a majority of the $C_0, \dots, C_{m-1}$ accept.

Finally, we turn to setting the parameter m so that C' only errs with probability at most p'. Let X_i be the random variable that indicates whether C_i accepts conditioned on its coin tosses. Then the probability that C' errs is upper-bounded by the probability that $(\sum_i X_i)/m$ deviates from the mean

$$\mu = \Pr[C(x, r) = 1] \geq 1 - p$$

by more than $\varepsilon = 1/2 - p$. By the Chernoff bound (Theorem 4.5), this occurs with probability at most $2^{-cm\varepsilon^2}$ for some constant $c > 0$, so setting m such that $m \geq \log(1/p')/c\varepsilon^2$ completes the proof. □

It remains open whether a similar result holds for general (i.e., non-constant-time) two-sided error PACA. Generalizing the proof would require at the very least a construction for intersecting non-constant-time PACA languages. If such a construction were to be known, then extending the idea above one could use that the union of constantly many T-time PACA languages can be recognized in $O(T)$ time (as we prove later in Proposition 4.21) and represent the majority over m PACA languages $L_0, \dots, L_{m-1}$ as the union over all possible intersections of $\lceil m/2 \rceil$ many L_i. Note that closure under intersection is open in the deterministic setting (i.e., of DACA) as well (see Chapter 2).

4.3.2. One- vs. Two-Sided Error

The results of Section 4.3.1 are useful in establishing the following:

Theorem 4.1. *The following hold:*

1. *If C is a one-sided error PACA with time complexity T, then there is an equivalent two-sided error PACA C' with time complexity $O(T)$.*
2. *There is a language L recognizable by constant-time two-sided error PACA but not by any $o(\sqrt{n})$-time one-sided error PACA.*

Proof. The first item follows from Proposition 4.12: Transform C into a one-sided error PACA C' with error at most $1/3$ and then notice that C' also qualifies as a two-sided error PACA (as it simply never errs on "no" instances).

For the second item, consider the language

$$L = \{x \in \{0,1\}^+ \mid |x|_1 \leq 1\}.$$

We obtain a constant-time two-sided error PACA for L by having its cells behave as follows: If a cell receives a 0 as input, then it immediately accepts; otherwise, it collects two random bits in the first two steps and then, seeing the bits as the binary representation of an integer $t \in [4]$, it accepts (only) in the subsequent t-th step. Hence, if the input x is such that $|x|_1 \leq 1$, the PACA always accepts; conversely, if $|x|_1 \geq 2$, then the PACA only accepts if all 1 cells pick the same value for t, which occurs with probability at most $1/4$.

It remains to show that $L(C) \neq L$ for any one-sided error PACA C with time complexity $T = o(\sqrt{n})$. Let n be large enough so that $T = T(n) \leq \sqrt{n}/2$. Notice that

$$L \cap \{0,1\}^n = \{0^n, x_1, \ldots, x_n\}$$

where $x_i = 0^{i-1}10^{n-i}$. Let us now assume that $x_i \in L(C)$ holds for every i. Since C accepts with probability at least $1/2$, by the pigeonhole principle there is a random input r such that $C(x_i, r) = 1$ for at least a $1/2$ fraction of the x_i. In addition, there are only T steps in which C may accept, which also implies there is a step $t < T$ such that at least a $1/2T$ fraction of the x_i is accepted by C in step t. Since there are $n/2T \geq 2T \geq 2t+2$ such x_i, we can find $i, j \in [n]$ such that $j \geq i + 2t + 1$ and $x_i, x_j \in L(C)$.

Consider now the input

$$x^* = 0^{i-1}10^{j-i-1}10^{n-j},$$

which is not in L. We argue $C(x^*, r) = 1$, thus implying $L(C) \neq L$ and completing the proof. We can see this by comparing the local views of the "bad" word x^* with the "good" ones x^i and x^j (see Figure 4.3 for an illustration): Let $k \in [n]$ be any cell of C. If $k < j - t$, then the configuration of the t-neighborhood of k in x^* is identical to that it has in x_i, so k must be accepting in step t. Similarly, if $k \geq j - t$, then the t-neighborhood of k in x^* is the same as in x_j, so k must be accepting as well. It follows that all cells of C are accepting in step t when C is given the inputs x^* and r. □

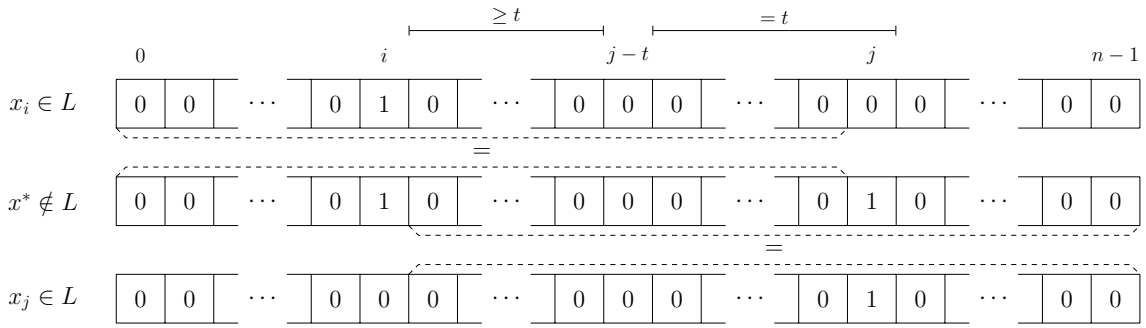


Figure 4.3.: Constructing $x^* \notin L$ from $x_i, x_j \in L$. The numbers above the cells indicate their respective indices. Since every t-neighborhood of x^* appears in either x_i or x_j and both x_i and x_j are accepted in (exactly) t steps, it follows that C accepts x^* in t steps.

4.4. The Constant-Time Case

In this section we now focus on constant-time PACA. Our goal will be to characterize the constant-time classes of both one- and two-sided error PACA (i.e., Theorems 4.3 and 4.4). First, we introduce the concept of *critical cells*, which is central to our analysis.

4.4.1. Critical Cells

Definition 4.14 (Critical cell)**.** Let C be a one- or two-sided error PACA, and let $x \in L(C) \cap \Sigma^n$. We say a cell $i \in [n]$ is *critical* for x in step $t \in \mathbb{N}_0$ if there are random inputs $R, R' \in \{0,1\}^{t \times n}$ such that i is accepting in step t of $C(x, R)$ but not in step t of $C(x, R')$.

In other words, if E is the event of i being accepting in step t of C on input x, then $0 < \Pr[E] < 1$ (where the probability is taken over the coin tosses of C). We should stress that whether a cell is critical or not may be *highly dependent* on x and t; for instance, there may be inputs $x_1 \neq x_2$ where the cell i is critical for x_1 but not for x_2.

As it turns out, the number of critical cells of a constant-time PACA is also constant.

Lemma 4.15. *Let C be a T-time (one- or two-sided error) PACA for $T \in \mathbb{N}_+$, and let $x \in L(C) \cap \Sigma^n$. In addition, let $t \in [T]$ be a time step in which x is accepted by C with non-zero probability. Then there are $2^{O(T^2)}$ cells that are critical for x in step t. It follows there are $T \cdot 2^{O(T^2)} = 2^{O(T^2)}$ critical cells for x in total (i.e., all such time steps comprised).*

Proof. For $i \in [n]$, let E_i denote the event in which the i-th cell accepts in step t. Assume towards a contradiction that there is $x \in L(C)$ in which strictly more than

$$2T \cdot (1 + \log T) \cdot 2^{T^2} = 2^{O(T^2)}$$

cells are critical for x (in step t). By inspection, this implies there is a set $K \subseteq [n]$ of $|K| \geq (1 + \log T) \cdot 2^{T^2}$ cells such that every $k \in K$ is critical for x and, for every distinct

$i, j \in K$, $|i - j| \geq 2T$.[6] Since there are at most T^2 coin tosses that determine whether a cell $i \in K$ accepts or not (i.e., those in the T-lightcone of i), $\Pr[E_i] < 1$ if and only if $\Pr[E_i] \leq 1 - 2^{-T^2}$. By Lemma 4.11, the events E_i are all independent, implying

$$\Pr[C \text{ accepts } x \text{ in step } t] \leq \Pr[\forall i \in K : E_i] = \prod_{i \in K} \Pr[E_i] \leq \left(1 - \frac{1}{2^{T^2}}\right)^{|K|} < \frac{1}{e^{1+\log T}} < \frac{1}{2T}.$$

Since t was arbitrary, by a union bound it follows that the probability that C accepts x in step t is strictly less than $1/2$. This contradicts $x \in L(C)$ both when C is a one- and a two-sided error PACA. □

We remark that, for one-sided error PACA, the upper bound above can be slightly improved to $2^{O(T)}$ critical cells. To show this, we prove a result akin to a pumping lemma for the languages accepted by these machines and which may be of independent interest.

Lemma 4.16. *Let C be a T-time one-sided error PACA for some $T \in \mathbb{N}_+$, and let $x \in L(C)$. Then, for any strings $u, v, w, y, z \in \Sigma^*$ with $|y|, |z| \geq T - 1$ and such that*

$$x = uyzwyzv,$$

we have that, for every $i \in \mathbb{N}_0$,

$$x_i = uy(zwy)^i zv \in L(C).$$

Moreover, none of the cells in the zwy segments of x_i are critical (in any time step in which C accepts x_i with non-zero probability). In particular, there are at most $2^{O(T)}$ cells that are critical for x (in any such time step).

Proof. First we prove that $x_i \in L(C)$ for every i. Let $R \in \{0,1\}^{T \times |x|}$ such that $C(x, R) = 1$, and let $n = |x|$ and $n_i = |x_i|$. We argue that, for any cell $j \in [n_i]$, there is a corresponding $j' \in [n]$ such that the $(T-1)$-lightcones of j and j' are identical. This can be easily observed using $|y|, |z| \geq T - 1$:

- The cells of u (resp., v) in x_i correspond to the cells of u (resp., v) in x.
- The cells of the first occurrence of y (resp., last occurrence of z) in x_i correspond to the cells of the first occurrence of y (resp., last occurrence of z) in x.
- For $i \geq 1$, the cells of any occurrence of zwy in x_i correspond to the cells of zwy in x.

As a result, we have $C(x_i, R) = 1$ and, by definition, $x_i \in L(C)$.

Next we show that none of the cells in the zwy segments are critical (in any fixed time step $t \in [T]$ in which x_i is accepted with non-zero probability). Assume towards a contradiction that there is a critical cell k in the zwy segment of x. (The proof for x_i is similar.) As in the proof of Lemma 4.15, the probability that k accepts is at most $1 - 2^{-T^2}$. Consider the input

[6] For instance, in the extreme case where every $0 \leq i < 2T \cdot 2^{T^2}$ is critical, pick $K = \{2jT \mid j \in [2^{T^2}]\}$.

x_m for $m = 2 \cdot (1 + \log T) \cdot 2^{T^2}$. Then the cells $k + jr$ for $j \in [m/2]$ and $r = 2|zwy|$ accept independently of one another (due to Lemma 4.11 and $r > 2(T-1)$). In addition, their $(T-1)$-neighborhoods are identical, and so their acceptance probabilities are exactly the same. Using an argument as in the proof of Lemma 4.15, this implies

$$\Pr[C \text{ accepts } x_m \text{ in step } t] \leq \left(1 - \frac{1}{m}\right)^m < \frac{1}{e^{1+\log T}} < \frac{1}{2T}.$$

Since t was arbitrary, (again, by a union bound) this contradicts C being a one-sided error PACA.

Finally, we turn to the upper bound on the number of critical cells. Identify the $(T-1)$-neighborhood of a critical cell k with yz where $|y| = T-1$, $|z| = T$, and k is the first cell of z. Then it follows from the above that any critical cell k' whose $(T-1)$-neighborhood is identical to k cannot be more than $2(T-1)$ cells away from k (i.e., $|k - k'| \leq 2(T-1)$). As there can be at most $|\Sigma|^{2T-1} = 2^{O(T)}$ many substrings of the type yz, this means there are at most $(2T-1) \cdot 2^{O(T)} = 2^{O(T)}$ critical cells in total. □

4.4.2. Characterization

We now almost fully characterize the constant-time PACA languages by proving Theorems 4.3 and 4.4.

4.4.2.1. One-Sided Error PACA

Theorem 4.3. *For any constant-time one-sided error PACA C, there is a constant-time DACA C' such that $L(C) = L(C')$.*

Lemma 4.15 implies that, given any input $x \in L(C)$, the decision of C accepting can be traced back to a set K of critical cells where $|K|$ is constant. To illustrate the idea behind the result, suppose that, if C accepts, then it always does so in a fixed time step $t < T$ and also that the cells in K are all far apart (e.g., more than $2(T-1)$ cells away from each other as in Lemma 4.11). Let us *locally* inspect the space-time diagram of C for x, that is, by looking at the t-lightcone of each cell i. Then we notice that, if $i \in K$, there is a choice of random bits in the t-lightcone of i that causes i to accept; conversely, if $i \notin K$, then *any* setting of random bits results in i accepting. Consider how this changes when $x \notin L(C)$ while assuming K remains unchanged. Every cell $i \notin K$ still behave the same; that is, it accepts regardless of the random input it sees. As for the cells in K, however, since they are all far apart, it cannot be the case that we still find random bits for every $i \in K$ that cause i to accept; otherwise C would accept x with non-zero probability, contradicting the definition of one-sided error PACA. Hence, there must be at least some $i \in K$ that *never* accepts. In summary, (under these assumptions) we can locally distinguish $x \in L(C)$ from $x \notin L(C)$ by looking at the cells of K and checking whether, for every $i \in K$, there is at least one setting of the random bits in the t-lightcone of i that causes it to accept.

In the proof we generalize this idea to handle the case where the cells in K are not necessarily far from each other—which in particular means we can no longer assume that the events of them accepting are independent—as well as of K varying with the input. Since Lemma 4.15 only gives an upper bound for critical cells when the input is in $L(C)$, we must also account for the case where the input is not in $L(C)$ and $|K|$ exceeds said bound.

Proof. Let $T \in \mathbb{N}_+$ be the time complexity of C, and let M be the upper bound on the number of critical cells (for inputs in $L(C)$) from Lemma 4.15. Without restriction, we may assume $T > 1$. We first give the construction for C' and then prove its correctness.

Construction. Given an input $x \in \Sigma^n$, the automaton C' operates in two phases. In the first one, the cells communicate so that, in the end, each cell is aware of the inputs in its r-neighborhood, where $r = (2M - 1)(T - 1)$. (Note this is possible because r is constant.) The second phase proceeds in T steps, with the cells assuming accepting states or not depending on a condition we shall describe next. After the second phase is over (and C' has not yet accepted), all cells unconditionally enter a non-accepting state and maintain it. Hence, C' only ever accepts during the second phase.

We now describe when a cell $i \in [n]$ is accepting in the t-th step of the second phase, where $t \in [T]$. Let K be the set of critical cells of x in step t. The decision process is as follows:

- First the cell checks whether there are strictly more than M cells in its r-neighborhood N that are critical in step t of C. (If i cannot determine this for any cell $j \in N$ (since it did not receive the entire t-neighborhood of j during the first phase), then j is simply ignored.) If this is the case, then i enters a non-accepting state and maintains it.
- The cell then determines whether it is critical itself in step t of C. If this is *not* the case, then it becomes accepting if and only if it is accepting in step t of C (regardless of the random input).
- Otherwise i is critical in step t. Let $B_i \subseteq [n]$ be the subset of cells that results from the following sequence of operations:
  1. Initialize B_i to $\{i\}$.
  2. For every cell $j \in B_i$, add to B_i every $k \in K$ such that $|j - k| \leq 2(T - 1)$.
  3. Repeat step 2 until a fixpoint is reached.

  (This necessarily terminates since there are at most M critical cells in N.) By choice of r, we have that $|i - j| \leq 2(M - 1)(T - 1)$ for every $j \in B_i$. In particular, we have that the t-neighborhood of every $j \in B_i$ is completely contained in N, which means

cell i is capable of determining B_i.[7] The cell i then accepts if and only if there is a setting of random bits in the lightcone L_i of radius r and height t centered at i that causes every cell in B_i to accept in step t of C.

This process repeats itself in every step t of the second phase. Note that it can be performed by i instantaneously (i.e., without requiring any additional time steps of C') since it can be hardcoded into the local transition function δ.

Correctness. It is evident that C' is a constant-time PACA, so all that remains is to verify its correctness. To that end, fix an input x and consider the two cases:

$x \in L(C)$. Then there is a random input R such that C accepts x in step $t \in [T]$. This means that, for every critical cell $i \in K$, if we set the random bits in L_i according to R, then every cell in B_i accepts in step t of C. Likewise, every cell $i \notin K$ is accepting in step t of C by definition. In both cases we have that i also accepts in the t-th step of the second phase of C', thus implying $x \in L(C')$.

$x \notin L(C)$. Then, for every random input R and every step $t \in [T]$, there is at least one cell $i \in [n]$ that is not accepting in the t-th step of $C(x, R)$. If i is not critical, then i is also not accepting in the t-th step of the second phase of C' (regardless of the random input), and thus C' also does not accept x. Hence, assume that every such i (i.e., every i such that there is a random input R for which i is not accepting in the t-th step of $C(x, R)$) is a critical cell.

Let $J \subseteq [n]$ denote the set of all such cells and, for $i \in J$, let $D_i \subseteq J$ be the subset that contains every $j \in J$ such that the events of i and j accepting in step t are *not* independent (conditioned on the random input to C). In addition, let A_i denote the event in which every cell of D_i is accepting in step t of $C(x, U_{T\times n})$. We shall show the following holds:

Claim. *There is an $i \in J$ such that $\Pr[A_i] = 0$; that is, for every R, there is at least one cell in D_i that is not accepting in step t of $C(x, R)$.*

This will complete the proof since then i is also not accepting in the t-th step of the second phase of C' (since any cell in D_i is necessarily at most $2(M-1)(T-1)$ cells away from i), thus implying $x \notin L(C')$.

To see the claim is true, suppose towards a contradiction that, for every $i \in J$, we have $\Pr[A_i] > 0$. If there are $i, j \in J$ such that $j \notin D_i$ (and similarly $i \notin D_j$), then by definition

$$\Pr[C(x, U_{T\times n}) = 1] \geq \Pr[A_i \wedge A_j] = \Pr[A_i] \Pr[A_j] > 0,$$

[7] Note it is not necessary for i to be aware of the actual numbers of the cells in B_i; it suffices for it to compute their positions relative to itself. For example, if $i = 5$ and $B_i = \{4, 5\}$, then it suffices for i to regard $j = 4$ as cell -1 (relative to itself). Hence, by "determining B_i" here we mean that i computes only these relative positions (and not the absolute ones, which would be impossible to achieve in only constant time).

contradicting $x \notin L(C)$. Thus, there must be $i \in J$ such that $J = D_i$; however, this then implies

$$\Pr[C(x, U_{T\times n}) = 1] = \Pr[A_i] > 0.$$

As this is also a contradiction, the claim (and hence the theorem) follows. □

4.4.2.2. Two-Sided Error PACA

This section in divided into two parts. In the first, we introduce the class LLT of locally linearly testable languages and relate it to other classes of subregular languages. The second part covers the proof of Theorem 4.4 proper.

Local languages. We introduce some notation. For $\ell \in \mathbb{N}_0$ and a word $w \in \Sigma^*$, $p_\ell(w)$ is the prefix of w of length ℓ if $|w| \geq \ell$, or w otherwise; similarly, $s_\ell(w)$ is the suffix of length ℓ if $|w| \geq \ell$, or w otherwise. The set of infixes of w of length ℓ is denoted by $I_\ell(w)$.

The subregular language classes from the next definition are due to McNaughton and Papert [77] and Beauquier and Pin [13].

Definition 4.17 (SLT, LT, LTT)**.** A language $L \subseteq \Sigma^*$ is *strictly locally testable* if there is $\ell \in \mathbb{N}_+$ and sets $\pi, \sigma \subseteq \Sigma^{\leq \ell}$ and $\mu \subseteq \Sigma^\ell$ such that, for every $w \in \Sigma^*$, $w \in L$ if and only if $p_{\ell-1}(w) \in \pi$, $I_\ell(w) \subseteq \mu$, and $s_{\ell-1}(w) \in \sigma$. The class of all such languages is denoted by SLT.

A language $L \subseteq \Sigma^*$ is *locally testable* if there is $\ell \in \mathbb{N}_+$ such that, for every $w_1, w_2 \in \Sigma^*$ with $p_{\ell-1}(w_1) = p_{\ell-1}(w_2)$, $I_\ell(w_1) = I_\ell(w_2)$, and $s_{\ell-1}(w_1) = s_{\ell-1}(w_2)$, we have that $w_1 \in L$ if and only if $w_2 \in L$. The class of locally testable languages is denoted by LT.

A language $L \subseteq \Sigma^*$ is *locally threshold testable* if there are $\theta, \ell \in \mathbb{N}_+$ such that, for any two words $w_1, w_2 \in \Sigma^*$ for which the following conditions hold, $w_1 \in L$ if and only if $w_2 \in L$:

1. $p_{\ell-1}(w_1) = p_{\ell-1}(w_2)$ and $s_{\ell-1}(w_1) = s_{\ell-1}(w_2)$.
2. For every $m \in \Sigma^\ell$, if $|w_i|_m < \theta$ for any $i \in \{1, 2\}$, then $|w_1|_m = |w_2|_m$.

The class of locally threshold testable languages is denoted by LTT.

The class LT equals the closure of SLT under Boolean operations (i.e., union, intersection, and complement) and the inclusion SLT $\subsetneq$ LT is proper. As for LTT, it is well-known that it contains every one of the languages

$$\mathsf{Th}(m, \theta) = \{w \in \Sigma^* \mid |w|_m \leq \theta\}$$

for $m \in \Sigma^*$ and $\theta \in \mathbb{N}_0$. Also, we have that LT $\subsetneq$ LTT and that LTT is closed under Boolean operations. We write $\mathsf{SLT}_\cup$ (resp., $\mathsf{LTT}_\cup$) for the closure of SLT (resp., LTT) under union and $\mathsf{LLT}_{\cup\cap}$ for the closure of LTT under union and intersection.

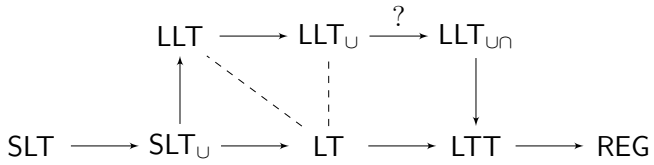


Figure 4.4.: Placement of the class LLT in the subregular hierarchy. The arrows indicate inclusion relations. Every inclusion is strict except for the one with a question mark (i.e., $\mathsf{LLT}_\cup \subseteq \mathsf{LLT}_{\cup\cap}$). The dashed lines indicate the respective classes are incomparable.

Definition 4.18 (LLT)**.** For $\ell \in \mathbb{N}_0$, $\theta \in \mathbb{R}_0^+$, $\pi, \sigma \subseteq \Sigma^{\leq \ell-1}$, and $\alpha \colon \Sigma^\ell \to \mathbb{R}_0^+$, $\mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$ denotes the language of all $w \in \Sigma^+$ that satisfy $p_{\ell-1}(w) \in \pi$, $s_{\ell-1}(w) \in \sigma$, and

$$\sum_{m \in \Sigma^\ell} \alpha(m) \cdot |w|_m \leq \theta.$$

A language $L \subseteq \Sigma^+$ is said to be *locally linearly testable* if there are ℓ, π, σ, α, and θ as above such that $L = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$. We denote the class of all such languages by LLT.

Proposition 4.19. *The following hold:*

1. $\mathsf{SLT}_\cup \subseteq \mathsf{LLT} \subseteq \mathsf{LLT}_\cup \subseteq \mathsf{LLT}_{\cup\cap} \subseteq \mathsf{LTT}$.
2. LLT *and* LT *as well as* $\mathsf{LLT}_\cup$ *and* LT *are incomparable.*
3. LTT *equals the Boolean closure over* LLT.
4. $\mathsf{LLT}_{\cup\cap}$ *is properly contained in* LTT.

Some relations between the classes are still open (see Figure 4.4) and are left as a topic for future work.

Proof. We skip the proof of the inclusion $\mathsf{SLT}_\cup \subseteq \mathsf{LLT}$ as well as the last item in the claim since both follow from Theorem 4.4 and whose proof does not depend on the results at hand. Furthermore, the inclusions $\mathsf{LLT} \subseteq \mathsf{LLT}_\cup$ and $\mathsf{LLT}_\cup \subseteq \mathsf{LLT}_{\cup\cap}$ are trivial, so we need only prove $\mathsf{LLT} \subseteq \mathsf{LTT}$ (and, since LTT is closed under union and intersection, $\mathsf{LLT}_{\cup\cap} \subseteq \mathsf{LTT}$ will follow).

To that end, let $L = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$ be given. The proof is by induction on the number k of words $m \in \Sigma^\ell$ such that $\alpha(m) \neq 0$. If $k = 0$, then the linear condition of L is always satisfied, directly implying $L \in \mathsf{LTT}$ (or, better yet, $L \in \mathsf{SLT}$). For the induction step, suppose the claim has been proven up to a number k, and let there be $k+1$ words $m \in \Sigma^\ell$ with $\alpha(m) \neq 0$. In addition, let $\mu \in \Sigma^\ell$ with $\alpha(\mu) \neq 0$ be arbitrary, and let $r \in \mathbb{N}_0$ be maximal with $r\alpha(\mu) \leq \theta$. Consider the languages $L_i = \mathsf{LTT}_\ell(\pi, \sigma, \alpha_i, \theta_i)$ for $i \in [0, r]$, $\theta_i = \theta - i\alpha(\mu)$, and α_i such that, for every $m \in \Sigma^\ell$,

$$\alpha_i(m) = \begin{cases} \alpha(m), & m \neq \mu; \\ 0, & m = \mu. \end{cases}$$

By the induction hypothesis, the L_i are all in LTT. On the other hand, we have

$$L = \bigcup_{i=0}^{r} (\mathsf{Th}(\mu, i) \cap L_i) \setminus \mathsf{Th}(\mu, i-1),$$

which implies $L \in \mathsf{LTT}$ since LTT is closed under Boolean operations.

For the second item in the claim, we show there are L_1 and L_2 such that $L_1 \in \mathsf{LLT} \setminus \mathsf{LT}$ and $L_2 \in \mathsf{LT} \setminus \mathsf{LLT}_\cup$. For the first one, probably the simplest is to set $L_1 = \mathsf{Th}(1, 1)$ (where the underlying alphabet is $\Sigma = \{0, 1\}$). The language is clearly not in LT (because otherwise either *both* $0^n 1 0^n$ and $0^n 1 0^n 1 0^n$ would be in L_1 or not (for sufficiently large n)); meanwhile, we have $L_1 \in \mathsf{LLT}$ since $\ell = 1$, $\pi = \sigma = \{\varepsilon\}$, $\alpha(0) = 0$, $\alpha(1) = 1$, and $\theta = 1$ satisfy $L_1 = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$. As for L_2, consider

$$L_2 = \{w \in \{0, 1\}^* \mid w \notin \{0\}^*\},$$

which is in LT because $L_2 = \{0, 1\}^* \setminus \{0\}^*$ (and LT is closed under Boolean operations). To argue that $L_2 \notin \mathsf{LLT}_\cup$, suppose towards a contradiction that $L_2 = \bigcup_{i=1}^{k} L_i'$ for $L_i' = \mathsf{LLin}_{\ell_i}(\pi_i, \sigma_i, \alpha_i, \theta_i)$. Since k is finite, there must be at least one L_i' that contains infinitely many words of the form $0^j 1 0^j$ for $j \in \mathbb{N}_0$. This implies $0^{\ell_i - 1} \in \pi_j, \sigma_j$ as well as $\alpha(0^{\ell_i}) = 0$, from which it follows that

$$\sum_{m \in \{0,1\}^{\ell_i}} \alpha_i(m) \cdot |0^k|_m = \alpha_i(0^{\ell_i}) \cdot |0^k|_{0^{\ell_i}} = 0 \leq \theta_i$$

for any $k \in \mathbb{N}_0$, thus contradicting $0^k \notin L_i'$.

Finally, the third item directly follows from the following well-known characterization of LTT: A language L is in LTT if and only if it can be expressed as the Boolean combination of languages of the form $\mathsf{Th}(m, \theta)$, $\pi\Sigma^*$, and $\Sigma^*\sigma$ where $m, \pi, \sigma \in \Sigma^*$ and $\theta \in \mathbb{N}_0$. Obviously these three types of languages are all contained in LLT; since $\mathsf{LLT} \subseteq \mathsf{LTT}$, it follows that LTT equals the Boolean closure of LLT. □

The proof of Theorem 4.4. We restate the result here for the reader's convenience:

Theorem 4.4. *The class of languages that can be accepted by a constant-time two-sided error PACA contains* $\mathsf{LLT}_{\cup\cap}$ *and is strictly contained in* LTT.

The theorem is proven by two inclusions, the first being that every $\mathsf{LLT}_{\cup\cap}$ language can be recognized by a constant-time two-sided error PACA. The first step is showing (Lemma 4.20) that we can "tweak" the components of the LLT condition so that it is more amenable to being tested by a PACA. Having done so, the construction is more or less straightforward: We collect the subwords of length ℓ in every cell and then accept with the "correct" probabilities. To lift the construction from LLT to its closure $\mathsf{LLT}_{\cup\cap}$, we show (Proposition 4.21) that the constant-time PACA languages are closed under union and intersection.

The second inclusion (i.e., showing that $L(C) \in \mathsf{LTT}$ for every constant-time two-sided error PACA C) is more complex. The proof again bases on the class LLT and uses the fact that LTT equals the Boolean closure over LLT (Proposition 4.19). If the cells of C all accept independently from one another in time at most T, then (as we address as a warm-up in the proof) things are relatively simple since we need only consider subwords of length $\ell = 2T + 1$ and set their weight according to the respective acceptance probability. Lifting this idea to the general case, however, requires quite a bit of care since the LLT condition does not account for subword overlaps. For instance, there may be cells c_1 and c_2 that are further than T cells apart and that both accept with non-zero probability but where c_i accepts if and only if c_{3-i} does not (recall Example 4.10). We solve this issue by blowing up ℓ so that a subword covers not only a single cell's neighborhood but that of an *entire group* of cells whose behavior may be correlated with one other. Here we once more resort to Lemmas 4.11 and 4.15 to upper-bound the size of this neighborhood by a constant.

As discussed above, for the proof we first need to make the linear condition of LLT a bit more manageable:

Lemma 4.20. *For any $L = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$, there is α' such that $L = \mathsf{LLin}_\ell(\pi, \sigma, \alpha', \theta')$ and:*

1. *The threshold θ' is equal to 1.*
2. *There is a constant $\varepsilon > 0$ such that, for every $w \in \Sigma^*$, we have either $f'(w) < 1 - \varepsilon$ or $f'(w) > 1 + \varepsilon$, where*
$$f'(w) = \sum_{m \in \Sigma^\ell} \alpha'(m) \cdot |w|_m.$$
3. *There is $k \in \mathbb{N}_0$ such that, for every m, $2^{\alpha'(m)} = 2^k/(n+1)$ for some $n \in [2^k]$.*

Proof. The case where $\alpha(m) = 0$ for every m is trivial, so suppose there is at least one m for which $\alpha(m) > 0$. Because $|w|_m \in \mathbb{N}_0$ for every w and every m,

$$f(w) = \sum_{m \in \Sigma^\ell} \alpha(m) \cdot |w|_m$$

is such that, given any $b \in \mathbb{R}_0^+$, there are only finitely many values of $f(w) \leq b$ (i.e., the set $\{f(w) \mid f(w) \leq b\}$ is finite). Hence, by setting

$$r = \frac{1}{2}\left(\max_{f(w) \leq \theta} f(w) + \min_{f(w) > \theta} f(w)\right)$$

and $\alpha''(m) = \alpha(m)/r$, we have that

$$f''(w) = \sum_{m \in \Sigma^\ell} \alpha''(m) \cdot |w|_m$$

satisfies the second condition from the claim (i.e., for every $w \in \Sigma^*$, either $f''(w) < 1 - \varepsilon$ or $f''(w) > 1 + \varepsilon$) for some adequate choice of $\varepsilon > 0$.

Now we show how to satisfy the last condition without violating the first two. To that end, let a be the minimal $\alpha''(m)$ for which $\alpha''(m) > 0$. In addition, let $k \in \mathbb{N}_0$ be such that $\alpha''(m) < k/2$ for every m and that

$$\log(2^{k/2} + 1) - \log(2^{k/2}) = \log(2^{k/2} + 1) - \frac{k}{2} \leq \frac{a\varepsilon}{2|\Sigma|^\ell}.$$

(This is possible because $\log(n+1) - \log n$ tends to zero as $n \to \infty$.) Then, for every m, we set $\alpha'(m) = k - \log(n+1)$ where $n \in [2^k]$ is minimal such that $\alpha'(m) \geq \alpha''(m)$. By construction, $f'(w) \geq f''(w)$, so we need only argue that there is $\varepsilon' > 0$ such that, for every w for which $f''(w) < 1 - \varepsilon$, we also have $f'(w) < 1 - \varepsilon'$. In particular every said w must be such that, for every m, $|w|_m \leq 1/a$ (otherwise we would have $f''(w) > 1$). Noting that $|\alpha'(m) - \alpha''(m)|$ is maximal when $\alpha'(m) = k/2$ and $\alpha''(m) = k - \log(2^{k/2} + 1) + \delta$ for very small $\delta > 0$, we observe that

$$f'(w) = \sum_{m \in \Sigma^\ell} \alpha'(m) \cdot |w|_m \leq f''(w) + \frac{|\Sigma|^\ell}{a} \left(\log(2^{k/2} + 1) - \frac{k}{2} \right) < 1 - \varepsilon + \frac{\varepsilon}{2} = 1 - \frac{\varepsilon}{2},$$

that is, $f'(w) < 1 - \varepsilon'$ for $\varepsilon' = \varepsilon/2$, as desired. □

The next proposition shows that the constant-time two-sided error PACA languages are closed both under union and intersection.

Proposition 4.21. *Let C_1 and C_2 be constant-time two-sided error PACA. Then there are constant-time two-sided error PACA $C_\cup$ and $C_\cap$ such that $L(C_\cup) = L(C_1) \cup L(C_2)$ and $C_\cap = L(C_1) \cap L(C_2)$.*

Proof. We first address the closure under union. Using Proposition 4.13, we may assume that C_1 and C_2 are two-sided ε-error PACA for some $\varepsilon < 1/6$. As we show further below, it suffices to have $C_\cup$ simulate both C_1 and C_2 simultaneously (using independent random bits) and accept if either of them does. To realize this, we can simply adapt the construction from Proposition 4.12 using $m = 2$ to simulate one copy of C_1 and C_2 each (instead of two independent copies of the same PACA).

The probability that $C_\cup$ accepts an $x \in L_i \setminus L_j$ for $i, j \in \{1, 2\}$ and $i \neq j$ is at least $1 - \varepsilon$, and the probability that $C_\cup$ accepts $x \in L_1 \cap L_2$ is even larger (i.e., at least $1 - \varepsilon^2 \geq 1 - \varepsilon$). Conversely, the probability that $C_\cup$ accepts an $x \notin L_1 \cup L_2$ is

$$\Pr[C_\cup \text{ accepts } x] \leq \Pr[C_1 \text{ accepts } x] + \Pr[C_2 \text{ accepts } x] = 2\varepsilon < \frac{1}{3}.$$

Since $C_\cup$ has time complexity $2T = O(T)$, the claim follows.

For the closure under intersection, we will use a similar strategy. This time suppose that the error probability ε of C_1 and C_2 is so that $(1 - \varepsilon)^2 \geq 2/3$ (e.g., $\varepsilon = 1/10$ suffices). Again we let $C_\cap$ simulate two copies of C_1 of C_2 simultaneously; however, this time we use a simplified version of the construction from the proof of Proposition 4.13. More

specifically, we set $m = 2$ and leave out the $[M]$ component from the construction. That is, $C_\cap$ randomly (and independently) picks random inputs r_1 and r_2 for C_1 and C_2, respectively, and accepts if and only if C_1 accepts with coin tosses from r_1 and also C_2 accepts with coin tosses from r_2. (One must also be careful since C_1 and C_2 may accept in different time steps, but this is already accounted for in the construction from Proposition 4.13.)

Since the copies of C_1 and C_2 are simulated independently from one another and $C_\cap$ accepts if and only both do (in the simulation), we have

$$\Pr[C_\cap \text{ accepts } x] = \Pr[C_1 \text{ accepts } x] \Pr[C_2 \text{ accepts } x].$$

In case $x \in L(C_1) \cap L(C_2)$, this probability is at least $(1-\varepsilon)^2 \geq 2/3$; otherwise it is upper-bounded by $\varepsilon < 1/3$. Since C has time complexity $O(T^2)$, which is constant, the claim follows. □

We are now in position to prove Theorem 4.4.

Proof of Theorem 4.4. We prove the two intersections from the theorem's statement. We address first the more immediate one, which is the inclusion of $\mathsf{LLT}_{\cup\cap}$ in the class of constant-time two-sided error PACA.

First inclusion. Given $L = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$, we construct a constant-time two-sided error PACA C with $L(C) = L$. This suffices since, by Proposition 4.21, the class of constant-time two-sided error PACA languages is closed under union and intersection. We apply Lemma 4.20 and assume $\theta = 1$ and that there are k and ε as in the statement of Lemma 4.20. For simplicity, we also assume $\ell = 2k + 1$.

The automaton C operates in k steps as follows: Every cell sends its input symbol in both directions as a signal and, at the same time, aggregates the symbols it sees, thus allowing it to determine the initial configuration $m \in \Sigma^\ell$ of its k-neighborhood. Meanwhile, every cell also collects k random bits $r \in \{0,1\}^k$. The decision to accept is then simultaneously made in the k-th step, where a cell with k-neighborhood m accepts with probability $2^{-\alpha(m)}$ (independently of other cells). (This can be realized, for instance, by seeing r as the representation of a k-bit integer in $[2^k]$ and accepting if and only if $r \leq n$, where n is such that $2^{-\alpha(m)} = (n+1)/2^k$.) In the case of the first (resp., last) cell of C, it also checks that the prefix (resp., suffix) of the input is in π (resp., σ), rejecting unconditionally if this is not the case.

Hence, for an input word $w \in \Sigma^+$, the probability that C accepts is

$$\prod_{m \in \Sigma^\ell} \left(\frac{1}{2^{\alpha(m)}}\right)^{|w|_m} = 2^{-f(w)},$$

where

$$f(w) = \sum_{m \in \Sigma^\ell} \alpha(m) \cdot |w|_m.$$

Thus, if $w \in L$, then C accepts with probability $2^{-f(w)} > (1/2)^{1-\varepsilon}$; conversely, if $w \notin L$, the probability that C accepts is $2^{-f(w)} < (1/2)^{1+\varepsilon}$. Since ε is constant, we may apply Proposition 4.13 and reduce the error to $1/3$.

Second inclusion. The proof of the second implication is more involved. Let C be a T-time two-sided error PACA for some $T \in \mathbb{N}_+$. We shall obtain the result $L(C) \in \mathsf{LTT}$ in three steps:

1. The first step is a warm-up in which we consider the case where the cells of C accept all independently from one another and that, if C accepts, then it does so in a fixed time step $t < T$.
2. In the second step, we relax the requirement on independence between the cells by considering groups of cells (of maximal size) that may be correlated with one another regarding their acceptance.
3. Finally, we generalize what we have shown so that it also holds in the case where C may accept in any step $t < T$. This is the only part in the proof where closure under complement is required. (Here we use item 3 of Proposition 4.19.)

Step 1. Suppose that C only accepts in a fixed time step $t < T$ and that the events of any two cells accepting are independent from one another. We show that $L(C) = \mathsf{LLin}(\pi, \sigma, \alpha, \theta)$ for an adequate choice of parameters.

Set $\ell = 2t + 1$ and let p_m be the probability that a cell with t-neighborhood $m \in \Sigma^\ell$ accepts in step t. In addition, let $\pi = \{p_{\ell-1}(w) \mid w \in L(C)\}$ and $\sigma = \{s_{\ell-1}(w) \mid w \in L(C)\}$ as well as $\theta = \log(3/2)$ and $\alpha(m) = \log(1/p_m)$ for $m \in \Sigma^\ell$. The probability that C accepts a word $w \in \Sigma^+$ is

$$\prod_{m \in \Sigma^\ell} p_m^{|w|_m} = \prod_{m \in \Sigma^\ell} \left(\frac{1}{2^{\alpha(m)}}\right)^{|w|_m} = 2^{-f(w)},$$

which is at least $2/3$ if and only if $f(w) \leq \theta$. It follows that $L(C) = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$.

Step 2. We now relax the requirements from the previous step so that the events of any two cells accepting need no longer be independent from one another. (However, C still only accepts in the fixed time step t.)

Let $K = 2^{O(T^2)}$ be the upper bound on critical cells of C guaranteed by Lemma 4.15, and let $\ell = 2(K + 2)T$. Again, we set $\pi = \{p_{\ell-1}(w) \mid w \in L(C)\}$ and $\sigma = \{s_{\ell-1}(w) \mid w \in L(C)\}$ as well as $\theta = \log(3/2)$. As for $\alpha(m)$, we set $\alpha(m) = 0$ unless m is such that there is $d \leq K$ with

$$m = abr_1s_1r_2s_2 \cdots r_ds_dc$$

where $a \in \Sigma^T$ is arbitrary, $b \in \Sigma^{2T}$ contains *no critical cells*, each $r_j \in \Sigma$ is a critical cell (for step t), the $s_j \in \Sigma^{\leq 2T-1}$ are arbitrary, and $c \in \Sigma^*$ has length $|c| \geq T$ and, if c contains any critical cell, then this cell accepts independently from r_d. In addition, we require c to be of maximal length with this property.

Note that a is needed as context to ensure that b indeed does not contain any critical cell (since determining this requires knowledge of the states in the T-neighborhood of the respective cell); the same is true for r_d and c. By construction together with Lemma 4.11, the group of cells $r_1, \dots, r_d$ is such that (although the cells in it do not accept independently from one another) its cells accepts independently from any other critical cell in C. The segment b ensures that m aligns properly with the group and that we consider every such group exactly once.

Letting p_m be the probability that every one of the r_j accept, for m as above we set $\alpha(m) = \log(1/p_m)$. Then, as before, the probability that C accepts a word $w \in \Sigma^+$ is

$$\prod_{m \in \Sigma^\ell} p_m^{|w|_m} = \prod_{m \in \Sigma^\ell} \left(\frac{1}{2^{\alpha(m)}}\right)^{|w|_m} = 2^{-f(w)},$$

which is at least $2/3$ if and only if $f(w) \leq \theta$, thus implying $L(C) = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta)$.

Step 3. In this final step we generalize the argument so that it also applies to the case where C may accept in any time step $t < T$. The first observation is that, given any $p > 0$, if we set $\theta = \log(1/p)$ in the second step above (instead of $\log(3/2)$), then we have actually shown that

$$\{w \in \Sigma^+ \mid \Pr[C \text{ accepts } w \text{ in step } t] \geq p\} = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta).$$

In fact, we can generalize this even further: Given any $\varnothing \neq \tau \subseteq [T]$, by setting α adequately we can consider the acceptance probability for the steps in τ altogether:[8]

$$L(\tau, p) = \{w \in \Sigma^+ \mid \Pr[C \text{ accepts } w \text{ in every step } t \in \tau] \geq p\} = \mathsf{LLin}_\ell(\pi, \sigma, \alpha, \theta).$$

This is because the bound on critical cells of Lemma 4.15 holds for all steps where C accepts with non-zero probability and, in addition, as defined above m already gives enough context to check if the respective critical cells also accept in any previous step. (That is, we construct m as above by using $t = \max \tau$; however, since the sets of critical cells for different time steps may not be identical, we must also relax the condition for the r_i so that r_i needs only be a critical cell in at least one of the time steps of τ.) Since LTT is closed under complement, we then also have

$$\overline{L(\tau, p)} = \{w \in \Sigma^+ \mid \Pr[C \text{ accepts } w \text{ in every step } t \in \tau] < p\} \in \mathsf{LTT}.$$

Fix some input word $w \in \Sigma^+$ to C. For $\varnothing \neq \tau \subseteq [T]$, let Z_τ denote the event where C accepts w in every step $t \in \tau$. By the inclusion-exclusion principle, we have

$$\begin{aligned}\Pr[C \text{ accepts } w] &= \Pr\left[\exists t \in [T] : Z_{\{t\}}\right] \\ &= \sum_{\substack{\tau \subseteq [T] \\ |\tau|=1}} \Pr[Z_\tau] - \sum_{\substack{\tau \subseteq [T] \\ |\tau|=2}} \Pr[Z_\tau] + \cdots + (-1)^{T+1} \Pr[Z_{[T]}]\end{aligned}$$

[8] Of course we are being a bit sloppy here since Definition 4.8 demands that a PACA should halt whenever it accepts. What is actually meant is that, having fixed some random input, if we extend the space-time diagram of C on input w so that it spans all of its first T steps (which we can do simply by applying the transition function of C), then, for every $t \in \tau$, the t-th line in the diagram contains only accepting cells.

(where the probabilities are taken over the coin tosses of C). This means that, if we are somehow given values for the $\Pr[Z_\tau]$ so that the sum above is at least $2/3$, then we can intersect a finite number of $L(\tau, p(\tau))$ languages and their complements and obtain some language that is guaranteed to contain only words in $L(C)$. Concretely, let $p(\tau) \geq 0$ for every $\varnothing \neq \tau \subseteq [T]$ be given so that

$$\sum_{\substack{\varnothing \neq \tau \subseteq [T] \\ |\tau| \text{ odd}}} p(\tau) - \sum_{\substack{\varnothing \neq \tau \subseteq [T] \\ |\tau| \text{ even}}} p(\tau) \geq \frac{2}{3}.$$

Let

$$\mathcal{T}_{\text{odd}} = \{\tau \subseteq [T] \mid \tau \neq \varnothing, |\tau| \text{ odd}, p(\tau) > 0\}$$

and similarly

$$\mathcal{T}_{\text{even}} = \{\tau \subseteq [T] \mid \tau \neq \varnothing, |\tau| \text{ even}, p(\tau) > 0\}.$$

Then necessarily

$$L(p) = \left(\bigcap_{\tau \in \mathcal{T}_{\text{odd}}} L(\tau, p(\tau)) \right) \cap \left(\bigcap_{\tau \in \mathcal{T}_{\text{even}}} \overline{L(\tau, p(\tau))} \right) \subseteq L(C)$$

contains every $w \in L(C)$ for which $\Pr[Z_\tau] \geq p(\tau)$ for $\tau \in \mathcal{T}_{\text{odd}}$ and $\Pr[Z_\tau] \leq p(\tau)$ for $\tau \in \mathcal{T}_{\text{even}}$.

The punchline is that *there are only finitely many values the* $\Pr[Z_\tau]$ *may assume.* This is because Z_τ only depends on a finite number of coin tosses, namely the ones in the lightcones of the cells that are critical in at least one of the steps in τ (which, again, is finite due to Lemma 4.15). Hence, letting P denote the set of all possible mappings of the τ subsets to these values, we may write

$$L(C) = \bigcup_{p \in P} L(p),$$

which proves $L(C) \in \mathsf{LTT}$.

Strictness of inclusion. The final statement left to prove is that the inclusion just proven is proper. This is comparatively much simpler to prove. We show that the language

$$L = \{w \in \{0,1\}^+ \mid |w|_1 \geq 2\} \in \mathsf{LTT}$$

of binary words with at least two occurrences of a 1 cannot be accepted by two-sided error PACA in constant time.

For the sake of argument, assume there is such a PACA C with time complexity $T \in \mathbb{N}_+$. Let us consider which cells in C are critical based on their initial local configuration. Certainly a cell with an all-zeroes configuration 0^{2T-1} cannot be critical. Since $0^n 1 0^n \notin L(C)$ for any

n (but $0^n10^n1 \in L(C)$), there must be m_1, m_2 so that $m_1 + m_2 = 2T - 2$ and $c = 0^{m_1}10^{m_2}$ is the initial local configuration of a critical cell. However, this means that in

$$x = 0^{2T}(c0^{2T})^{T2^T} \in L$$

we have at least 2^T cells in x that are critical for the same time step $t \in [T]$ (by an averaging argument) and that are also all independent from one another (by Lemma 4.11). In turn, this implies

$$\Pr[C \text{ accepts } x] \leq \left(1 - 2^{-T}\right)^{2^T} < \frac{1}{e} < \frac{2}{3},$$

contradicting $x \in L(C)$. □

4.5. The General Sublinear-Time Case

Proposition 4.22. *There is a constant $c > 0$ such that the following holds: Let monotone functions $T, T', h, p : \mathbb{N}_+ \to \mathbb{N}_+$ be given with $h(n) \leq 2^n$, $p(n) = \mathrm{poly}(n)$, and $p(n) \geq n$ and such that, for every n and $p'(n) = \Theta(p(n) \log h(n))$,*

$$T(6h(n)p(n)) \geq cp'(n).$$

In addition, let the binary representation of $h(n)$ and $p'(n)$ be computable in $O(p'(n))$ time by a Turing machine (given n in unary), and let $T'(N)$ be computable in $O(T'(N))$ time by a Turing machine (again, given N in unary). Furthermore, suppose that, for every T-time one-sided error PACA C, there is a T'-time DACA C' such that $L(C) = L(C')$. Then

$$\mathsf{RPTIME}[p(n)] \subseteq \mathsf{TIME}[h(n) \cdot T'(h(n) \cdot \mathrm{poly}(n)) \cdot \mathrm{poly}(n)].$$

Recall we say a DACA C is *equivalent* to a PACA C' if $L(C) = L(C')$. As a corollary of Proposition 4.22, we get:

Theorem 4.2. *Let $d \geq 1$. The following hold:*

- *If there is $\varepsilon > 0$ such that every n^ε-time (one- or two-sided error) PACA can be converted into an equivalent n^d-time deterministic CA, then* $\mathsf{P} = \mathsf{RP}$.
- *If every* $\mathrm{polylog}(n)$*-time PACA can be converted into an equivalent n^d-time deterministic CA, then, for every $\varepsilon > 0$,* $\mathsf{RP} \subseteq \mathsf{TIME}[2^{n^\varepsilon}]$.
- *If there is $b > 2$ so that any $(\log n)^b$-time PACA can be converted into an equivalent n^d-time deterministic CA, then, for every $a \geq 1$ and $\alpha > a/(b-1)$,* $\mathsf{RPTIME}[n^a] \subseteq \mathsf{TIME}[2^{O(n^\alpha)}]$.

The first item is obtained by setting (say) $T(n) = n^\varepsilon$, $T'(n) = n^d$, and $h(n) = p(n)^{2(1/\varepsilon-1)}$ (assuming $\varepsilon < 1$) in Proposition 4.22. For the second one, letting $p(n) = n^a$ where $a > 0$ is arbitrary and (again) $T'(n) = n^d$, set $T(n) = (\log n)^{2+a/\varepsilon}$ and $h(n) = 2^{n^\varepsilon/(d+1)}$. Finally, for the last one, letting again $p(n) = n^a$ and $T'(n) = n^d$, set $T(n) = (\log n)^b$ and $h(n) = 2^{n^\alpha}$.

At the core of the proof of Proposition 4.22 is a padding argument. Nevertheless, we stress this padding must be cleverly implemented so that it can be verified *in parallel* and also *without initial knowledge of the input length* (since a cell initially knows nothing besides its input symbol). To do so, we resort to a technique that can be traced back to the work of Ibarra, Palis, and Kim [60], splitting the input into blocks of the same size that redundantly encode the input length in a locally verifiable way. More importantly, the blocks are numbered from left to right in ascending order, which also allows us to verify that we have the number of blocks that we need (so that the input is "long enough" and the PACA achieves the time complexity that we desire). We refer the reader to Chapters 2 and 3 (in particular Section 3.3.1) for other applications of the same technique also in the context of sublinear-time CA.

Proof. Let $L \in \mathsf{RP}$ be decided by an RP machine R whose running time is upper-bounded by p. Without restriction, we assume $p(n) \geq n$. Using standard error reduction in RP, there is then an RP machine R' with running time $p'(n) = \Theta(p(n) \log h(n))$ (i.e., polynomial in n), space complexity at most $p(n)$, and which errs on $x \in L$ with probability strictly less than $1/2h(n)$. Based on L we define the language

$$L' = \{\mathrm{bin}_n(0)\#x_0\#0^{p(n)}\% \cdots \% \,\mathrm{bin}_n(h(n)-1)\#x_{h(n)-1}\#0^{p(n)} \mid n \in \mathbb{N}_+, x_i \in L \cap \Sigma^n\},$$

where $\mathrm{bin}_n(i)$ denotes the n-bit representation of $i < 2^n$. Note the length of an instance of L' is

$$N \leq 6h(n)p(n) = O(h(n)\mathrm{poly}(n)).$$

We claim there is $c > 0$ such that L' can be accepted in at most $cp'(n) = O(p'(n))$ (and in particular less than $T(N)$) time by a one-sided error PACA C. The construction is relatively straightforward: We refer to each group of cells

$$\mathrm{bin}_n(i)\#x_i\#0^{p'(n)}$$

separated by the % symbols as a *block* and the three binary strings in each block (separated by the # symbols) as its *components*. First each block $a_1\#a_2\#a_3$ checks that its components have correct sizes, that is, that $|a_1| = |a_2|$ and $|a_3| = p(|a_1|)$. Then the block communicates with its right neighbor $b_1\#b_2\#b_3$ (if it exists) and checks that $|a_i| = |b_i|$ for every i and that, if $a_1 = \mathrm{bin}_n(j)$, then $b_1 = \mathrm{bin}_n(j+1)$. In addition, the leftmost block checks that its first component is equal to $\mathrm{bin}_n(0)$; similarly, the rightmost block computes $h(n)$ (in $O(p'(n))$ time) and checks that its first component is equal to $\mathrm{bin}_n(h(n)-1)$. Following these initial checks, each block then simulates R' (using bits from its random input as needed) on the input given in its second component using its third component as the tape. If R' accepts, then all cells in the respective block turn accepting. In addition, the delimiter % is always accepting unless it is a border cell.

Clearly C accepts if and only if its input is correctly formatted and R' accepts every one of the x_i (conditioned on the coin tosses that are chosen for it by the respective cells of C). Using a union bound, the probability that C errs on an input $x \in L'$ is

$$\Pr[C(x, U_{T\times n}) = 0] \leq \sum_{i=0}^{h(n)-1} \Pr[R(x_i) = 0] < \sum_{i=0}^{h(n)-1} \frac{1}{2h(n)} = \frac{1}{2}.$$

In addition, the total running time of C is the time needed for the syntactic checks (requiring $O(p'(n))$ time), plus the time spent simulating R' (again, $O(p'(n))$ time using standard simulation techniques). Hence, we can implement C so that it runs in at most $cp'(n)$ time for some constant $c > 0$, as desired.

Now suppose there is a DACA C' to C as in the statement of the theorem. We shall show there is a deterministic (multi-tape) Turing machine that decides L with the purported time complexity. Consider namely the machine S which, on an input $x \in \{0,1\}^n$ of L, produces the input

$$x' = \mathrm{bin}_n(0)\#x\#0^{p(n)}\% \cdots \% \,\mathrm{bin}_n(h(n)-1)\#x\#0^{p(n)}$$

of L' and then simulates C' on x' for $T'(N)$ steps, accepting if and only if C' does. Producing x' from x requires $O(N)$ time and, using the standard simulation of cellular automata by Turing machines, C' can be simulated in $O(N \cdot T'(N))$ time. Checking whether C' accepts or not can be performed in parallel to the simulation and requires no additional time. Hence, S runs in

$$O(N \cdot T'(N)) = O(h(n)\mathrm{poly}(n) \cdot T'(h(n)\mathrm{poly}(n)))$$

time, as required. □

5. Pseudorandom Generators for Sliding-Window Algorithms

Abstract

A sliding-window algorithm of window size t is an algorithm whose current operation depends solely on the last t symbols read. We construct pseudorandom generators (PRGs) for low-space randomized sliding-window algorithms that have access to a binary randomness source. More specifically, we lift these algorithms to the non-uniform setting of branching programs and study them as sliding-window branching programs (SWBPs), which we propose as the branching program analogue of sliding-window algorithms. For general SWBPs, given a base PRG G_{base} that $\varepsilon_{\text{base}}$-fools width-$w$, length-$t$ (general) branching programs, we construct a PRG that fools any same-width SWBP of length n using an additional $O(\log(n/t)\log(1/\varepsilon_{\text{base}}))$ random bits with a $(n/2t)^{O(1)}$ multiplicative loss in the error parameter. We also consider a subclass of SWBPs called δ-critical SWBPs, which are SWBPs whose layers must either accept every input or reject with probability at least $1-\delta$. For δ-critical SWBPs, starting from a PRG G_{base} as before, we construct a PRG that fools any width-w, length-n SWBP using an additional $O(\log(t/\delta)) \cdot \tilde{O}(\log(1/\varepsilon_{\text{base}})) + O(\log(n/t))$ random bits and with a $(t/\delta)^{O(1)}$ multiplicative loss in error (up to polylogarithmic factors).

As an application, we show how to decide the language of a sublinear-time probabilistic cellular automaton using small space. Our results target the model of *PACAs*, which are probabilistic cellular automata that accept if and only if all cells are simultaneously accepting. For sublinear T, we prove that the every language accepted by a T-time one-sided error PACA (the PACA equivalent of RP) can be decided using $O(T + (\log n)^2)$ space. We also introduce δ-critical PACAs, which are PACAs in which every cell is such that, if a cell does not always accept, then the probability that it does is bounded away from 1 by at least δ. For this subclass of PACA, we obtain similar results for both one-sided and two-sided error PACAs (which are the PACA equivalent of BPP).

5.1. Introduction

The processing of long streams of data using as little memory resources as possible is a central computational paradigm of high relevance in the modern age. Through its presentation as *streaming algorithms*, the topic has been the subject of intense study within the realm of theoretical computer science. When dealing with data that ages quickly, however, it appears a more accurate representation of the process may be found in the

subclass of *sliding-window algorithms*. These are algorithms that maintain a window (hence the name) of only the last few symbols in their stream and model processes where only the most recent data is considered accurate or of relevance. Natural examples of this strategy may be found in weather forecasting and social media analysis.

A defining feature of sliding-window algorithms is *self-synchronization*: If there are two machines executing the same algorithm with a window of size t on the same stream and they receive faulty, inconsistent data at some point in the process, then the machines will return to having identical states (i.e., synchronize) after having read at most t identical symbols. It turns out this behavior is not only potentially desirable in practice but, from a theoretical point of view, it suggests the defining characteristic for these kind of algorithms. In this chapter, we shall adopt this standpoint and consider *low-space* algorithms with this property that are *augmented with access to a (binary) randomness source*. We will attempt to answer the following central question: How much (if any) randomness is required by the class of algorithms with this property in order to perform the tasks required of them?

Randomized sliding-window algorithms. Intuitively, we should obtain a randomized version of sliding-window algorithms by having the control unit operate according to a randomized process. Of course, this immediately raises the question of how one should adapt the sliding-window requirement to this new model. As the defining characteristic (in the deterministic case) is that the machine's current operation *depends only on the last t symbols read*, it suggests itself to require the algorithm's behavior to be dependent not only on the last t symbols *but also on the random choices made while reading said bits*. Since a true randomized algorithm (in our setting) is expected to use at least one bit of randomness for every new symbol it reads, it appears as if the algorithm's window would in its greater part (or at least as much so) contain bits originating from the randomness source rather than the input stream. Actually, we altogether forgo referring to the latter when generalizing the sliding-window property to randomized algorithms. That is, we phrase the property exclusively in terms of the randomness source and, using non-uniformity, forgo referencing the input stream altogether. This approach has a couple of advantages:

1. It not only simplifies the presentation (since then the algorithm passes its window over a single input source instead of two) but also renders the model more amenable to the usual complexity-theoretical methods for analyzing low-space algorithms.
2. The resulting class of machines is actually *stronger* than the model where the sliding-window property applies to both the data stream and the randomness source. Hence, our results not only hold in the latter case but also in the more general one.

As mentioned in passing above, we will rely on classical methods from complexity theory to analyze low-space algorithms in the context of derandomization, namely the non-uniform model of *branching programs*.

5.1.1. Branching Programs

Branching programs can be seen as a non-uniform variant of deterministic finite automata and have found broad application in the derandomization of low-space algorithms.

Definition 5.1. An *(ordered, read-once) branching program* P of length n and width w is a set of states Q, $|Q| = w$, of which a subset $Q_{\text{acc}} \subseteq Q$ are *accepting states*, along with n transition functions $P_1, \dots, P_n \colon Q \times \{0,1\} \to Q$. The program P starts its operation in an *initial state* $q_0 \in Q$ and then processes its input $x = x_1 \cdots x_n \in \{0,1\}^n$ from left to right while updating its state in step i according to P_i. That is, if P is in state q_{i-1} after having read $i-1$ bits of its input, then it next reads x_i and changes its current state to $P_i(q_{i-1}, x_i)$. We say P *accepts* x if $q_n \in Q_{\text{acc}}$, where q_n is state of P after processing the final input symbol x_n.

Although P reuses the same set of states throughout its processing of x, it is also natural to view P as a directed acyclic graph with $n+1$ *layers*, where the i-th layer Q_i contains a node for every state of Q that is reachable in (exactly) i steps of P. (We shall refer to Q_i as a set of nodes and as a subset of Q interchangeably.) A node $v \in Q_{i-1}$ is then connected to the nodes $u_0 = P_i(v,0)$ and $u_1 = P_i(v,1)$ of Q_i and we label the respective edges with 0 and 1. (If $u_0 = u_1$, then we have a double edge from v to u_0.)

We write $P(x)$ for the indicator function $\{0,1\}^n \to \{0,1\}$ of P accepting x (i.e., $P(x) = 1$ if P accepts x, or $P(x) = 0$ otherwise). For $y \in \{0,1\}^*$ with $|y| \leq n$, we write $P_0(y)$ to indicate the *state* of P after having read y when starting its operation in its initial state q_0. Letting λ denote the empty word, this can be defined recursively by setting $P_0(\lambda) = q_0$ and $P_0(y'z) = P_{|y'|+1}(P_0(y'), z)$ for $y = y'z$ and $z \in \{0,1\}$. We extend the domain of P_i from $Q \times \{0,1\}$ to $Q \times \{0,1\}^{\leq n-i}$ in the natural way as follows: $P_i(q, \lambda) = q$ and $P_i(q, yy') = P_{i+1}(P_i(q,y), y')$ for $yy' \in \{0,1\}^{\leq n-i}$ where $y \in \{0,1\}$.

Unanimity programs [15] are a generalization of branching programs that accept if and only if every state during its computation is accepting (instead of only its final one). Formally, this means we mark a subset $Q^i_{\text{acc}} \subseteq Q_i$ of every i-th layer as accepting and say that a unanimity program U *accepts* $x \in \{0,1\}^n$ if and only if $U_0(y) \in Q^{|y|}_{\text{acc}}$ for every prefix $y \neq \lambda$ of x. (Note this is indeed a generalization since setting $Q^i_{\text{acc}} = Q$ for every $i < n$ yields a standard branching program.) It is easy to see that a width-w unanimity program can be simulated by a width-$(w+1)$ standard branching program (e.g., by adding a "fail" state to indicate the unanimity program did not accept at some point).

Pseudorandom generators. The key concept connecting branching programs and the topic of derandomization is that of *pseudorandom generators*. These are general-purpose functions that take a (conceptually speaking) small subset of inputs and "scatter" them across their range in such a way that "appears random" to the class of procedures they intend to fool. In the definition below, U_n denotes a random variable distributed uniformly over $\{0,1\}^n$.

Definition 5.2. Let $n \in \mathbb{N}_+$ and $\varepsilon > 0$, and let $\mathcal{F}$ be a class of functions $f\colon \{0,1\}^n \to \{0,1\}$. We say a function $G\colon \{0,1\}^d \to \{0,1\}^n$ is a *pseudorandom generator* (PRG) that *ε-fools $\mathcal{F}$* if the following holds for every $f \in \mathcal{F}$:

$$|\Pr[f(G(U_d)) = 1] - \Pr[f(U_n) = 1]| \leq \varepsilon.$$

In addition, we say G is *explicit* if there is a (uniform) linear-space algorithm which, on input n (encoded in binary) and $s \in \{0,1\}^d$, outputs $G(s)$.

The connection between the above and low-space algorithms is as follows: Suppose we have an s-space algorithm A that takes as input a string x along with a stream $r \in \{0,1\}^n$ of random coin tosses. If we take a PRG G that ε-fools branching programs of width 2^s and length n, then $A(x, G(U_d))$ must be ε-close to $A(x, U_n)$. To see why, notice that we can convert $A(x, \cdot)$ into a branching program P_x by taking as state set for P_x the configuration space of A and non-uniformly hardcoding the bits of x read by A (into the transition function of P_x). The resulting program P_x then takes as input the random stream of A (which might appear counter-intuitive, but yields the desired reduction).

Sliding-window branching programs. In this chapter, our goal is to construct PRGs against the restricted class of unanimity programs satisfying the sliding-window property. As previously discussed, the defining feature is self-synchronization.

Definition 5.3. A *sliding window branching program* (SWBP) is a unanimity program S that satisfies the following property: There is a number $t \in \mathbb{N}_+$, called the *window size* of S, such that, for every $i \leq n - t$, every $y \in \{0,1\}^t$, and every pair of states $q, q' \in Q_i$, we have that $S_i(q, y) = S_i(q', y)$.

In other words, the current state of S depends exclusively on the last t bits read. From this perspective, it is not hard to see that a generalization to unanimity programs is actually necessary for the sliding-window property to be interesting. Indeed, if an SWBP S is a standard branching program (i.e., not just a unanimity program), then its decision only depends on the last t bits of its input.

5.1.2. Our Results

5.1.2.1. Structural Characterization of SWBPs

Let $n \in \mathbb{N}_0$. Recall the *de Bruijn graph* of dimension n is the directed graph $B_n = (V_n, E_n)$ with vertex set $V_n = \{0,1\}^n$ and edge set

$$E_n = \{(xw, wy) \mid w \in \{0,1\}^{n-1} \text{ and } x, y \in \{0,1\}\}.$$

Similarly, the *prefix tree* of dimension n is the graph $T_n = (V'_n, E'_n)$ with $V'_n = \{0,1\}^{\leq n}$ and

$$E'_n = \{(w, wx) \mid w \in \{0,1\}^{\leq n-1} \text{ and } x \in \{0,1\}\}.$$

As our first contribution, we observe that, for every fixed window size t, there is a "prototypical" SWBP Π whose topology is such that its first t layers are isomorphic to the prefix tree T_t and its subsequent layers are connected according to (the labeled version of) the de Bruijn graph B_t. Thus, every SWBP S of window size t can be obtained from this prototypical SWBP Π by merging nodes in the same layer. (Note this process may also incur merging nodes in subsequent layers as well so as to ensure that, for every i, S_i is indeed a function.)

Theorem 5.4. *A unanimity program S is a SWBP of window size t if and only if there are functions $\alpha_0, \ldots, \alpha_n \colon \{0,1\}^{k_i} \to Q$ with*

$$k_i = \begin{cases} i, & i < t; \\ t, & i \geq t \end{cases}$$

and such that the following are true for every $x, y \in \{0,1\}$ and $w \in \{0,1\}^{k_i - 1}$:

1. *For every $i < t$, $S_i(\alpha_i(w), y) = \alpha_{i+1}(wy)$.*
2. *For every $t \leq i < n$, $S_i(\alpha_i(xw), y) = \alpha_{i+1}(wy)$.*

From this characterization it immediately follows that, as long as S does not have redundant states, we have $w \leq 2^t$.

5.1.2.2. PRGs for SWBPs

We construct PRGs for SWBPs both in the general case and in a restricted one where the SWBPs are assumed to satisfy an additional property.

General SWBPs. We first present our generator for general SWBPs of window size t.

Theorem 5.5. *Let $n, w, t \in \mathbb{N}_+$ with $n \leq w$ and $\varepsilon > 0$ be given, and let $G_{\text{base}} \colon \{0,1\}^{d_{\text{base}}} \to \{0,1\}^t$ be a PRG that $\varepsilon_{\text{base}}$-fools width-$w$, length-$t$ unanimity programs. Then there is an explicit PRG $G \colon \{0,1\}^d \to \{0,1\}^n$ with seed length*

$$d = 2d_{\text{base}} + O(\log(n/t)\log(1/\varepsilon_{\text{base}}))$$

that ε_G-fools any width-w, length-n SWBP of window size t, where $\varepsilon_G = \varepsilon_{\text{base}} \cdot (n/2t)^{O(1)}$.

Armoni [4] gives an explicit generator $G_{\text{A}} \colon \{0,1\}^{d_{\text{A}}} \to \{0,1\}^n$ that (with the improvements by Kane, Nelson, and Woodruff [65]) ε_{A}-fools any width-w, length-n branching program for any choice of $\varepsilon_{\text{A}} > 0$ and $n, w \in \mathbb{N}_+$ (and, in particular, also ε_{A}-fools any width-$(w-1)$, length-n unanimity program). Assuming $n \leq w$, G_{A} has seed length

$$d_{\text{A}} = O\left(\frac{\log(w/\varepsilon_{\text{A}})\log n}{\max\{1, \log\log w - \log\log(n/\varepsilon_{\text{A}})\}}\right).$$

Given $\varepsilon > 0$, plugging in $G_{\text{base}} = G_{\text{A}}$ in Theorem 5.5 with $\varepsilon_{\text{base}} = \varepsilon_{\text{A}} = \varepsilon \cdot (2t/n)^{\Omega(1)}$, we get:

Corollary 5.6. *For every $n, w, t \in \mathbb{N}_+$ with $n \leq w$ and $\varepsilon > 0$, there is an explicit PRG $G\colon \{0,1\}^d \to \{0,1\}^n$ with seed length*

$$d = O\left(\frac{\log(w/\varepsilon)\log t}{\max\{1, \log\log w - \log\log(n/\varepsilon)\}} + \log(n/t)\log(n/t\varepsilon)\right)$$

that ε-fools any width-w, length-n SWBP of window size t.

Hence, in regimes where $\log w = \Omega(\log(n/\varepsilon))^2$, we effectively replace a $\log n$ with a $\log t$ factor in the seed length of Armoni's PRG. This is a considerable improvement for most applications, where the window size t is much smaller than the input length n. For instance, if the window size is $t = \mathrm{polylog}(w)$ and ε is constant, then the seed length is $O(\log w)$, which is essentially optimal.

δ-critical SWBPs. We also construct a PRG that fools a particular subclass of SWBPs. One motivation for this is that these SWBPs are related to a very natural subclass of probabilistic cellular automata (see Section 5.1.2.3).

For $\delta \in [0,1]$, a layer L of a unanimity program P is said to be *δ-critical* if the fraction of inputs to P that do not pass through an accept state in L is at least δ. A *δ-critical SWBP* is an SWBP in which every layer is either δ-critical or contains only accepting states. (That is, for every layer there is a "gap" of δ between surely accepting or not.)

Note that, by its structural properties (see Section 5.1.2.1), in an SWBP of window size t, if L contains at least one non-accepting state, then it must be (2^{-t})-critical. (In contrast, in general unanimity programs δ may be as small as 2^{-n}.)

Theorem 5.7. *Let $n, w, t \in \mathbb{N}_+$ with $n \leq w$ and $\varepsilon > 0$ be given, and let $G_{\mathrm{base}}\colon \{0,1\}^{d_{\mathrm{base}}} \to \{0,1\}^t$ be a PRG that $\varepsilon_{\mathrm{base}}$-fools width-$w$, length-$t$ unanimity programs where $\varepsilon_{\mathrm{base}} \leq \varepsilon^{1+\log 3}$. Then there is an explicit PRG $G\colon \{0,1\}^d \to \{0,1\}^n$ with seed length*

$$d = 2d_{\mathrm{base}} + O(\log(t/\delta)) \cdot \tilde{O}(\log(1/\varepsilon)) + O(\log(n/t))$$

that ε_G-fools δ-critical width-w, length-n SWBPs of window size t, where $\varepsilon_G = \tilde{O}(\varepsilon) \cdot (t/\delta)^{O(1)}$.

Given $\varepsilon > 0$, plugging in $G_{\mathrm{base}} = G_{\mathrm{A}}$ in Theorem 5.7 as before with $\varepsilon_{\mathrm{base}} = \varepsilon_{\mathrm{A}} = (\delta/t)^{\Omega(1)}\varepsilon^{1+\log 3}$, we get:

Corollary 5.8. *For every $n, w, t \in \mathbb{N}_+$ with $n \leq w$ and $\varepsilon, \delta > 0$, there is an explicit PRG $G\colon \{0,1\}^d \to \{0,1\}^n$ with seed length*

$$d = O\left(\frac{\log(w/\varepsilon\delta)\log t}{\max\{1, \log\log w - \log\log(n/\varepsilon\delta)\}}\right) + O(\log(t/\delta)) \cdot \tilde{O}(\log(t/\varepsilon\delta)) + O(\log(n/t))$$

that ε-fools any δ-critical width-w, length-n SWBP of window size t.

Besides the application we discuss next in Section 5.1.2.3, Corollary 5.8 allows us to obtain partial results for the case where $\log w$ is in-between $\Omega(\log(n/\varepsilon))$ and $O(\log(n/\varepsilon))^2$ (cnf. the discussion following Theorem 5.5). Consider, for instance, the regime where $t = \text{polylog}(n)$, $\delta = 1/\text{poly}(t)$, ε is constant, and $\log w = \Theta(\log n)^{1+\eta}$ for some $\eta > 0$. Then using as little as $O(\log w)$ space we can derandomize any $O(\log w)$-space sliding-window algorithm of window size $t = \text{polylog}(n)$ that is $(1/\text{poly}(t))$-critical (i.e., conditioned on the last t random bits read, if the algorithm has a non-zero probability of halting and rejecting its input, then this probability is at least $1/\text{poly}(t)$).

5.1.2.3. Application to Probabilistic Cellular Automata

As an application, we obtain space-efficient algorithms for deciding the languages accepted by sublinear-time probabilistic cellular automata. More specifically, our results target the model of PACAs (probabilistic ACAs; see Chapter 4), which are cellular automata with two local transition functions δ_0 and δ_1 and where, at every step, each cell tosses a fair coin $c \in \{0, 1\}$ and then updates its state according to δ_c. The acceptance condition is that of ACA, which is the most usual one [60, 66, 105] (recall also Chapter 2) that allows for non-trivial sublinear-time computations: A computation is accepting if and only if a configuration is reached in which every cell is accepting. Our results target both *one-sided* and *two-sided error* PACAs, which in a sense are the PACA analogues of the classical complexity classes RP and BPP, respectively. (We refer the reader to Section 5.6.1 for the definitions.)

These results are interesting because similar positive results in the sister setting of efficiently deciding the language of a PACA where efficiency is measured in terms of *time* complexity would have surprising consequences for the derandomization of Turing machines (e.g., P = RP; see Theorem 4.2).

One-sided error PACAs. As shown in Chapter 2 (Proposition 2.18), every language accepted by a deterministic ACA with time complexity T can be decided using $O(T)$ space. For general one-sided error PACAs, we obtain the following result:

Theorem 5.9. *Let $T : \mathbb{N}_+ \to \mathbb{N}_+$ be a (constructible) function and $\varepsilon \geq 1/\text{poly}(T)$. For any one-sided ε-error PACA C that recognizes its language $L(C)$ in time at most $T = T(n)$, there is a deterministic algorithm for $L(C)$ with space complexity $O(T + (\log n)^2)$.*

Hence, for $T \in \Omega(\log n)^2$, with $O(T)$ space we can also decide languages that are accepted by T-time *probabilistic* (one-sided error) ACA.

We also consider *δ-critical* PACAs, which are PACAs in which, for any given input, if a cell is such that it does not always accept, then the probability that it does is at most $1 - \delta$. For this subclass of PACAs, we prove:

Theorem 5.10. *Let $T\colon \mathbb{N}_+ \to \mathbb{N}_+$ be a (constructible) function, and let $\delta > 0$ and $\varepsilon \geq 1/\mathrm{poly}(T)$. For any δ-critical one-sided ε-error PACA C that recognizes its language $L(C)$ in time at most $T = T(n)$, there is a deterministic algorithm for $L(C)$ with space complexity*

$$O\left(\frac{(T + \log(1/\delta))\log T}{\max\{1, \log T - \log\log(n/\delta)\}}\right) + \tilde{O}(\log(T/\delta))^2 + O(\log n).$$

As with Theorem 5.7, here we are able to obtain partial results for efficiently deciding (with respect to space complexity) the language of a T-time PACA where T is in-between $\Omega(\log n)$ and $O(\log n)^2$. For instance, if we take the case where $T = \Theta(\log n)^{1+\eta}$ for some $\eta > 0$ and the PACA is δ-critical for $\delta = 1/\mathrm{poly}(T)$, then we get that $O(T)$ space suffices to decide $L(C)$ (as in the $\Omega(\log n)^2$ case). Interestingly, in the low-end case where $T = \Theta(\log n)$ and δ is may be as small as $1/2^{O(\sqrt{T})}$, the resulting upper bound is $O(\log n \log\log n)$, thus implying a *barely* superpolynomial upper bound of $O(n^{\log\log n})$ on the *time* complexity. It remains open whether $O(T)$ space always suffices for deciding $L(C)$ for *any* PACA C with time complexity in-between $\Omega(\log n)$ and $O(\log n)^2$ (i.e., without the additional δ-criticality assumption).

Two-sided error PACAs. Our results in the case of two-sided error PACAs are a bit more modest. In Section 5.6.3.2 we do prove a result similar to Theorem 5.9 but note it does not actually improve on what already can already be achieved by simply using Armoni's PRG. Nonetheless, for δ-critical PACAs we are able to prove the following based on Corollary 5.8:

Theorem 5.11. *Let $T\colon \mathbb{N}_+ \to \mathbb{N}_+$ be a (constructible) function, and let $\delta > 0$ and $\varepsilon \geq 2^{-O(T)}$. For any δ-critical two-sided $(1/2 - \varepsilon)$-error PACA C that recognizes its language $L(C)$ in time at most $T = T(n)$, there is a deterministic algorithm for $L(C)$ with space complexity $\tilde{O}(T \cdot (\log(1/\delta))^2) + O(\log n)$.*

In particular, for $T = \Omega(\log n)$ and $\delta = 1/\mathrm{poly}(T)$, this gives us a $\tilde{O}(T)$-space algorithm for every language accepted by a two-sided error PACA even when the gap between accepting and rejecting is exponentially small in T. As in the case of one-sided error, obtaining a similar result without the δ-criticality assumption is an interesting further research direction.

5.1.3. Technical Overview

Generators for general SWBPs. Our construction relies on two main ideas, the first of which is especially suited for exploiting the sliding-window property. With this technique, which we call *interleaving*, we are able to effectively "shatter" SWBPs into a collection of short programs. To visualize the idea, take some SWBP S of window size t and consider the behavior on S on an input

$$X \parallel Y = X_1 Y_1 X_2 Y_2 \cdots X_n Y_n$$

where the X_i and Y_i all have length t and $X = X_1 \cdots X_n$ and $Y = Y_1 \cdots Y_n$ are chosen independently from one another (but possibly follow the same distribution). The point is that, since S must read the t bits of Y_i between reading X_i and X_{i+1}, when it starts processing X_{i+1}, it has essentially "forgotten" all information about X_i. In addition, as X and Y are chosen independently from one another, Y_i contains no information about X_{i+1} whatsoever; the only relevance Y_i has regarding X_{i+1} is in determining which state the processing of X_{i+1} starts in. Hence, when the output of our generator is of the form $X \parallel Y$, we can simply (say) set Y to some fixed string y and then analyze S as a collection of n many length-t unanimity programs, each of which receives an X_i as input.

We have thus reduced our original task to that of fooling a collection of n programs *simultaneously*; that is, given unanimity programs $P_1, \ldots, P_n$ of the same width and length, we wish to generate pseudorandom inputs $X_1, \ldots, X_n$ so that

$$\left| \Pr[\forall i \in [n] : P_i(X_i) = 1] - \prod_{i=1}^{n} \Pr[P_i(U) = 1] \right|$$

is small, where U is the uniform distribution on all possible inputs to P_i. As it turns out, here we may employ almost the same idea as that behind the celebrated construction of Nisan [91]. That is, supposing we have a generator G that simultaneously fools $n/2$ programs using a random seed s of length d, we use an adequate *extractor* (Theorem 5.14) to generate a fresh seed s' using "just a couple more" random bits and then output the concatenation $G(s)G(s')$. (Refer to Section 5.2 for the definitions.) In our case, this strategy works even *better* than in the general setting of Nisan (i.e., we need even fewer random bits to succeed) because we only give very little entropy away in the first $n/2$ programs—namely, all that can be learned about s is that the $P_1, \ldots, P_{n/2}$ are all accepting. All that is left then is the base case $n = 1$, in which case we plug in any generator G_{base} for unanimity programs of our liking, the point being that G_{base} only has to fool programs that are as long as the *window size* t of our original SWBP S (rather than programs that are as long as S itself).

Generators for δ-critical SWBPs. To simplify the discussion, having fixed $\delta > 0$, we refer to a δ-critical layer of an SWBP simply as *critical* and the others as *non-critical*, accordingly. For an SWBP S of window size t, a *section* of S is a sequence of t contiguous layers of S (i.e., $Q_i, \ldots, Q_{i+t-1}$ for some $i \leq n - t + 1$).

The first observation we make is the following:

Lemma 5.12. *Let $t \in \mathbb{N}_+$ and $\varepsilon, \delta > 0$, and let S be a δ-critical SWBP of window size t that accepts at least an ε fraction of its inputs, that is, $\Pr[S(U_n) = 1] \geq \varepsilon$. Then there are at most $O((t/\delta)\log(1/\varepsilon))$ critical layers in S.*

Hence, provided S has non-negligible acceptance probability, we get an upper bound on the number of sections of S containing critical layers.

Imagine now we can construct a PRG G that fools (general) unanimity programs no matter the order in which the bits of G are read (e.g., the generator of Forbes and Kelley [38]). Assuming we know which sections of S are critical and which are not, we only need to use G to fill in the critical ones; the other ones can be given arbitrary values (since they accept regardless of what their input is). In particular, this means that we need G to output much less bits than the full length n of S (thus potentially giving us better parameters than in the general setting).

Of course, since our PRG must fool not only a single SWBP S (but any other SWBP with similar parameters), we cannot simply set some of our generator's outputs according to G and the others to non-arbitrary values. Nevertheless, we may *split* the output of G into blocks of size t and then use a *hash function* to order the blocks for us, repeating them as needed to obtain a string that is n bits long. As we prove, with the correct choice of parameters, we can guarantee that most hash functions are such that every critical section is assigned a unique block (i.e., the restriction of the hash function to the critical sections of S is injective). If we then remove the non-critical sections of S, we get a (much shorter) unanimity program S' that is equivalent to S (modulo the non-critical sections) and which is fooled by G, thus reducing the correctness of our generator to the pseudorandomness of G.

To obtain the best seed length for our generator, we actually refrain from using a PRG with as strong a requirement as above. Instead, we show that an adaptation of the same ideas used to prove Theorem 5.5 suffices. In particular, we use a PRG that simultaneously fools the critical sections of S, the point being that the notion of simultaneously fooling is not sensitive to reordering the programs that are to be fooled.

5.1.4. Related Work

Branching programs. The standard line of attack in complexity theory when derandomizing low-space algorithms is to lift these to the more general model of (non-uniform) branching programs, which are more amenable to a combinatorial analysis. This approach can be traced at least 30 years back to the seminal work of Nisan [91]. Since then, there has been progress in derandomizing branching programs in diverse settings including, for instance, branching programs in which the transition function at each layer is a permutation [58] or that may read their input in some fixed but unknown order [38]. Unanimity programs were recently proposed by Bogdanov et al. [15]. To the best of our knowledge, ours is the first work to study branching programs with the sliding-window property or similar.

Sliding-window algorithms. The sliding-window paradigm is a natural form of stream processing that has been considered in the context of database management systems [8], network monitoring [27], and reinforcement learning [49]. Starting with the work of Datar et al. [29], the sliding-window model has also been extensively studied in the context of maintaining statistics over data streams. (See, e.g., [16] for a related survey.)

Sliding-window algorithms have also been studied by Ganardi et al. [44–47] in the setting of language recognition (among others). We point out a couple fundamental differences between the model we consider and theirs:

- Their results also apply to the non-uniform case—but parameterized on the *window size*. In particular, their model allows radically different behaviors on the same stream for different window sizes. In our case, non-uniformity is parameterized on the *input size* (i.e., the length of the data stream).
- The underlying probabilistic model in the work of Ganardi et al. is the *probabilistic automata* model of Rabin [97], which allows state transitions according to arbitrary distributions. In contrast, our model draws randomness from a binary source and—most importantly—the sliding-window property *applies to the random input* (whereas in the model of Ganardi et al. it only applies to the data stream).

In summary, Ganardi et al. focus on a model that verifies (or computes some quantity for) every window of fixed size on its stream; we focus on a model that verifies *the stream as a whole*.

We also mention a recent paper by Pacut et al. [94] that points out a connection between distributed and sliding-window algorithms. In our case, we may see our application to probabilistic cellular automata as a direct consequence of this connection.

Probabilistic cellular automata. Our contribution to probabilistic cellular automata adds another link to a recent chain of results (Chapters 2 to 4) targeted at the study of sublinear-time cellular automata. As mentioned in Chapter 2, the topic has been seemingly neglected by the cellular automata community at large and, as far as we are aware of, the body of theory on the subject predating these more recent results resumes itself to [60, 66, 105]. A probabilistic model similar to the probabilistic cellular automata we consider was previously proposed by Arrighi, Schabanel, and Theyssier [7], but the results from Chapter 4 are the first to address the sublinear-time case.

Finally, note that as in Chapter 4 we use the term "probabilistic" (due to their similarity to probabilistic Turing machines) to refer to these automata and treat them separately from the more general *stochastic cellular automata* in which the local transition function may follow an arbitrary distribution (in the same spirit as the aforementioned work by Rabin [97]). Unfortunately, there is no consensus on the distinction between the two terms in the literature, and the two have been used interchangeably. For a survey on stochastic cellular automata, see [73].

5.1.5. Organization

The rest of the chapter is organized as follows: In Section 5.2, we recall the basic definitions and results that we need. The subsequent sections each cover one set of results: Section 5.3 addresses the structural result on SWBPs. In Section 5.4 and Section 5.5 we construct

the PRGs for general and δ-critical SWBPs, respectively. Finally, Section 5.6 covers the applications to probabilistic cellular automata.

5.2. Preliminaries

It is assumed the reader is familiar with basic notions of computational complexity theory and pseudorandomness (see, e.g., the standard references [5, 50, 113]).

All logarithms are to base 2. The set of integers is denoted by $\mathbb{Z}$, that of non-negative integers by $\mathbb{N}_0$, and that of positive integers by $\mathbb{N}_+$. For a set S and $n, m \in \mathbb{N}_+$, $S^{n\times m}$ is the set of n-row, m-column matrices over S. For $n \in \mathbb{N}_+$,

$$[n] = \{i \in \mathbb{N}_0 \mid i < n\}$$

is the set of the first n non-negative integers.

Symbols in words are indexed starting with one. The i-th symbol of a word w is denoted by w_i. For an alphabet Σ and $n \in \mathbb{N}_0$, $\Sigma^{\leq n}$ contains the words $w \in \Sigma^*$ for which $|w| \leq n$. Without restriction, the empty word is not an element of any language that we consider.

We write U_n (resp., $U_{n\times m}$) for a random variable distributed uniformly over $\{0,1\}^n$ (resp., $\{0,1\}^{n\times m}$). We will need the following variant of the Chernoff bound (see, e.g., [113]):

Theorem 5.13 (Chernoff bound)**.** *Let $X_1, \ldots, X_n$ be independently and identically distributed Bernoulli variables and $\mu = \mathrm{E}[X_i]$. Then there is a constant $c > 0$ such that the following holds for every $\varepsilon > 0$:*

$$\Pr\left[\left|\frac{\sum_{i=1}^n X_i}{n} - \mu\right| > \varepsilon\right] < 2^{-cn\varepsilon^2}.$$

Hash functions. For $N, M \in \mathbb{N}_+$, a family $H = \{h\colon [N] \to [M]\}$ of functions is said to be *pairwise independent* if, for $x_1, x_2 \in [N]$ with $x_1 \neq x_2$ and h chosen uniformly from H, the random variables $h(x_1)$ and $h(x_2)$ are independent and uniformly distributed. Equivalently, H is pairwise independent if for arbitrary $y_1, y_2 \in [M]$ we have

$$\Pr\left[h(x_1) = y_1 \wedge h(x_2) = y_2\right] = \frac{1}{M^2}.$$

It is a well-known fact (see, e.g., [113]) that there is a family H of pairwise independent functions such that one can uniformly sample a function h from H with $O(\log N + \log M)$ bits.

Extractors. Let $n \in \mathbb{N}_+$, and let X and Y be random variables taking values in $\{0,1\}^n$. Then the *statistical distance* between X and Y is

$$\Delta(X,Y) = \frac{1}{2}\sum_w |\Pr[X=w] - \Pr[Y=w]|.$$

The *min-entropy* $\mathrm{H}_\infty(X)$ of X is defined by

$$\mathrm{H}_\infty(X) = \min_{w \in \mathrm{Supp}(X)} \log \frac{1}{\Pr[X=w]}.$$

For $k \leq n$, if $\mathrm{H}_\infty(X) \geq k$, then X is said to be a *k-source*. For $d, m \in \mathbb{N}_+$ and $\varepsilon > 0$, a *(k,ε)-extractor* is a function $\mathrm{Ext}\colon \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ such that, for every k-source X, we have that

$$\Delta(\mathrm{Ext}(X, U_d), U_m) \leq \varepsilon.$$

In this context, the second argument of Ext is its *seed* and, correspondingly, d is its *seed length*. The following is due to the work of Goldreich and Wigderson [51] (see also [113]):

Theorem 5.14 ([51])**.** *For every $n, k \in \mathbb{N}_+$ and $\varepsilon > 0$, there is a (k,ε) extractor $\mathrm{Ext}\colon \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$ with seed length $d = O(n - k + \log(1/\varepsilon))$ that is computable in $O(n+d)$ space.*

5.3. De Bruijn Graphs Fully Characterize SWBPs

In this section, we prove:

Theorem 5.4. *A unanimity program S is a SWBP of window size t if and only if there are functions $\alpha_0, \ldots, \alpha_n\colon \{0,1\}^{k_i} \to Q$ with*

$$k_i = \begin{cases} i, & i < t; \\ t, & i \geq t \end{cases}$$

and such that the following are true for every $x, y \in \{0,1\}$ and $w \in \{0,1\}^{k_i - 1}$:

1. *For every $i < t$, $S_i(\alpha_i(w), y) = \alpha_{i+1}(wy)$.*
2. *For every $t \leq i < n$, $S_i(\alpha_i(xw), y) = \alpha_{i+1}(wy)$.*

Proof. We prove first the forward implication. Let S be an SWBP of window size t. The α_i are defined recursively, the basis case being $\alpha_0(\lambda)$, which is set to be initial state of S. Having defined α_i for $i < n$, we set α_{i+1} so that

$$\alpha_{i+1}(wy) = \begin{cases} S_i(\alpha_i(w), y), & i < t; \\ S_i(\alpha_i(xw), y), & i \geq t \end{cases}$$

for $x, y \in \{0,1\}$ and $w \in \{0,1\}^{k_i - 1}$ (thus automatically satisfying the requirements in the statement of the theorem).

It remains to show that the α_i are well-defined, which we shall prove by induction. The induction basis $i = 0$ is trivial, and the induction step for $i < t$ follows easily from applying the induction hypothesis and S_i being a function. Hence, suppose $i \geq t$. Letting $w \in \{0,1\}^{t-1}$ and $y \in \{0,1\}$, we shall show $S_i(\alpha_i(0w), y) = S_i(\alpha_i(1w), y)$. Using the induction hypothesis and the properties of the α_i, there are states q_0 and q_1 in the $(i-t+1)$-th layer of S so that $S_{i-t+1}(q_x, w) = \alpha_i(xw)$ for $x \in \{0,1\}$. Thus, by the sliding-window property of S (and, again, by the properties of the α_i),

$$S_i(\alpha_i(0w), y) = S_{i-t+1}(q_0, wy) = S_{i-t+1}(q_1, wy) = S_i(\alpha_i(1w), y).$$

For the converse implication, let α_i as in the statement be given. Then we argue that, for any i and $z \in \{0,1\}^t$ as well as any states q and q' that are reachable in the i-th layer of S, $S_i(q, z) = S_i(q', z)$ holds. This is simple to see by induction provided that q and q' are in the image of α_i. To see that this is indeed the case, note that, if q is reachable in the i-th layer by an input $wz \in \{0,1\}^i$ with $|z| = t$, then $q = \alpha_i(z)$ (again, due to the sliding-window property) or, in case $|z| = 0$ and $i < t$, $q = \alpha_i(w)$. In either case, the claim follows. □

5.4. Pseudorandom Generators for General SWBPs

In this section, we recall and prove:

Theorem 5.5. *Let $n, w, t \in \mathbb{N}_+$ with $n \leq w$ and $\varepsilon > 0$ be given, and let $G_{\text{base}} \colon \{0,1\}^{d_{\text{base}}} \to \{0,1\}^t$ be a PRG that $\varepsilon_{\text{base}}$-fools width-$w$, length-$t$ unanimity programs. Then there is an explicit PRG $G \colon \{0,1\}^d \to \{0,1\}^n$ with seed length*

$$d = 2d_{\text{base}} + O(\log(n/t)\log(1/\varepsilon_{\text{base}}))$$

that ε_G-fools any width-w, length-n SWBP of window size t, where $\varepsilon_G = \varepsilon_{\text{base}} \cdot (n/2t)^{O(1)}$.

As discussed in Section 5.1.3, our construction will rely on stretching the output of G_{base} so as to simultaneously fool as many unanimity programs as possible. Then we use interleaving to convert the resulting construction into a PRG for SWBPs. We address the two steps in this order.

Definition 5.15. Let $m, t \in \mathbb{N}_+$ and $\varepsilon > 0$, and let $\mathcal{F}$ be a class of functions $f \colon \{0,1\}^t \to \{0,1\}$. We say a distribution $X = (X_1, \ldots, X_m)$ over $(\{0,1\}^t)^m$ *m-simultaneously ε-fools $\mathcal{F}$* if the following holds for every $f_1, \ldots, f_m \in \mathcal{F}$:

$$\left| \Pr[\forall i \in [m] : f_i(X_i) = 1] - \prod_{i=1}^{m} \Pr[f_i(U_t) = 1] \right| \leq \varepsilon.$$

Similarly, we say a function $G \colon \{0,1\}^d \to (\{0,1\}^t)^m$ *m-simultaneously ε-fools $\mathcal{F}$* if $G(U_d)$ fools $\mathcal{F}$.

Lemma 5.16. *Let $G\colon \{0,1\}^d \to (\{0,1\}^t)^m$ be a function that m-simultaneously ε-fools width-w, length-t unanimity programs. Then there is a function $G'\colon \{0,1\}^{d'} \to (\{0,1\}^t)^{2m}$ with $d' = d + O(\log(1/\varepsilon))$ that $2m$-simultaneously 3ε-fools width-w, length-t unanimity programs. In addition, if G is explicit, then so is G'.*

Proof. Let $\mathrm{Ext}\colon \{0,1\}^d \times \{0,1\}^{d_{\mathrm{Ext}}} \to \{0,1\}^d$ be the (k,ε) extractor of Theorem 5.14 where $k = d - \log(1/\varepsilon)$ and $d_{\mathrm{Ext}} = O(\log(1/\varepsilon))$. We set

$$G'(s_G, s_{\mathrm{Ext}}) = G(s_G)G(\mathrm{Ext}(s_G, s_{\mathrm{Ext}}))$$

and denote the i-th component in the output of G' by $G'(\cdot)_i$. Now let $P_1, \dots, P_{2m}$ be width-w, length-t unanimity programs. For $j \in \{1,2\}$, let

$$A_j = \{x_1 \cdots x_m \in \{0,1\}^{mt} \mid x_1, \dots, x_m \in \{0,1\}^t \text{ and } \forall i \in [m] : P_{(j-1)m+i}(x_i) = 1\}$$

and $\mu_j = \mu(A_j)$. Observe that $\mu_1\mu_2 = \prod_{i=1}^{2m} \Pr[P_i(U_t) = 1]$. If $\mu_1 < 2\varepsilon$, then we immediately have a distance of at most 3ε between $\mu_1\mu_2 < 2\varepsilon$ and

$$\Pr[\forall i \in [2m] : P_i(G'(U_d)_i) = 1] \le \Pr[G(U_d) \in A_1] \le \mu_1 + \varepsilon < 3\varepsilon.$$

Hence, suppose that $\mu_1 \ge 2\varepsilon$. Observe that, by assumption on G, this means

$$\Pr[U_d = x \mid G(U_d) \in A_1] \le \frac{\Pr[U_d = x]}{\Pr[G(U_d) \in A_1]} \le \frac{2^{-d}}{\mu_1 - \varepsilon} \le \frac{2^{-d}}{\varepsilon}$$

for $x \in \{0,1\}^d$. In particular, this implies that, if Z is a random variable that is distributed according to $\Pr[Z = x] = \Pr[U_d = x \mid G(U_d) \in A_1]$, then $\mathrm{H}_\infty(Z) \ge k$. Thus, by the extraction property,

$$\begin{aligned}
&\left|\Pr[\forall i \in [2m] : P_i(G'_i(U_d)) = 1] - \mu_1\mu_2\right| \\
&\quad = \left|\Pr[G(U_d) \in A_1 \wedge G(\mathrm{Ext}(U_d, U_{d_{\mathrm{Ext}}})) \in A_2] - \mu_1\mu_2\right| \\
&\quad \le \left|\Pr[G(\mathrm{Ext}(U_d, U_{d_{\mathrm{Ext}}})) \in A_2 \mid G(U_d) \in A_1] - \mu_2\right| + \varepsilon \\
&\quad \le |\Pr[G(U_d) \in A_2] - \mu_2| + 2\varepsilon \\
&\quad \le 3\varepsilon.
\end{aligned}$$

□

Starting from a PRG that fools (single) width-w, length-t unanimity programs and repeating r times the construction of Lemma 5.16, we obtain:

Lemma 5.17. *Let $G\colon \{0,1\}^d \to \{0,1\}^t$ be a function that ε-fools width-w, length-t unanimity programs. For every $r > 0$, there is a function $G'\colon \{0,1\}^{d'} \to (\{0,1\}^t)^{2^r}$ with $d' = d + O(r\log(1/\varepsilon))$ that 2^r-simultaneously $3^r\varepsilon$-fools width-w, length-t unanimity programs. In addition, if G is explicit, then so is G'.*

In order to convert the PRG G' of Lemma 5.17 into a generator for SWBPs, we will show that *interleaving* of two independent copies of G' is sufficient. To this end, for $m, t \in \mathbb{N}_+$ and tuples $x = (x_1, \dots, x_m)$ and $y = (y_1, \dots, y_m)$ in $(\{0,1\}^t)^m$, let

$$x \parallel y = x_1y_1x_2y_2 \cdots x_my_m \in \{0,1\}^{2tm}.$$

Lemma 5.18. *Let $m, t \in \mathbb{N}_+$ and $\varepsilon > 0$ be arbitrary. Then, for any (independent) random variables $X = (X_1, \ldots, X_m)$ and $Y = (Y_1, \ldots, Y_m)$ taking values in $(\{0,1\}^t)^m$ and such that X (resp., Y) m-simultaneously ε-fools width-w, length-t unanimity programs, we have that $X \| Y$ (i.e., the random variable which assumes values $x \| y$ where x and y are drawn from X and Y, respectively) 2ε-fools width-w, length-$2tm$ SWBPs of window size t.*

Proof. Let S be a width-w, length-$2tm$ SWBP of window size t. We prove that $\Pr[S(X\|Y) = 1]$ is ε-close to $\Pr[S(X\|U) = 1]$, where U is a random variable that is uniformly distributed on $(\{0,1\}^t)^m$. By an analogous argument, we also have that $\Pr[S(X \| U) = 1]$ is ε-close to $\Pr[S(U \| U) = 1]$ (where the two occurrences of U denote independent copies of the same random variable). This then gives the statement of the lemma.

Fix some $x = (x_1, \ldots, x_m) \in \mathrm{Supp}(X)$ and let S_x^i be the width-w, length-t unanimity program defined as follows:

- S_x^i has the same states as S.
- The state transition function in the j-th layer of S_x^i is the same as that in the (k_i+t+j)-th layer of S, where $k_i = 2(i-1)t$.
- The initial state is $S_{k_i}(x_i)$. (Note this is well-defined due to S having window size t and $|x_i| = t$.)
- For $j < t$, the state s is accepting in the j-th layer of S_x^i if and only if it is accepting in the $(k_i + t + j)$-th layer of S. A state s in the t-th layer of S_x^i is accepting if and only if it is also accepting in the $(k_i + 2t)$-th layer of S and, in addition, $S_{k_i+2t}(x_{i+1}) = 1$. (If $i = m$, then this last condition holds vacuously.) Furthermore, the initial state of S_1 is only accepting if $S_0(x_1) = 1$.

Note that, by construction, the states that S assumes when reading the y parts of any input $x \| y$ are the same as the corresponding states of S_x^i. Hence, we observe that $S(x \| y) = 1$ if and only if $S_x^i(y_i) = 1$ for every i:

- If $S(x \| y) = 0$, then the input $x \| y$ passes through a rejecting state in the r-th layer of S for some $r \in [n]$.
  - If r corresponds to the y part of the input, that is, $r = k_i + t + j$ for some i and $j \in [t]$, then (by construction) y_i passes through the same state in the j-th layer of S_x^i.
  - Conversely, if r corresponds to the x part of the input, then $r = k_i + j$ for some i and $j \in [t]$. If $i > 1$, then $S_x^{i-1}(y_{i-1}) = 0$ since the state in the last layer of S_x^{i-1} rejects; otherwise, $i = 1$ and then $S_x^1(y_1) = 0$ since the initial state of S_x^1 is rejecting.

  In either case, we find some i so that $S_x^i(y_i) = 0$.

- If $S_x^i(y_i) = 0$ for some i, then there is $j \in [t]$ such that y_i passes through a rejecting state in the j-th layer of S_x^i. If the corresponding state in the $(k_i + j)$-th layer of S is also rejecting, then $S(x \parallel y) = 0$, so assume otherwise. Then either $j = t$ and then $S_{k+i+2t}(x_{i+1}) = 0$ or $j = 1$ and $i = 1$, in which case $S_0(x_1) = 0$. In either case, $S(x \parallel y) = 0$ follows.

By the above and using that X and Y are independent, it follows that

$$\begin{aligned}
&|\Pr[S(X \parallel Y) = 1] - \Pr[S(X \parallel U) = 1]| \\
&\quad = \left| \sum_x \Pr[X = x] \left(\Pr[S(x \parallel Y) = 1] - \Pr[S(x \parallel U) = 1]\right) \right| \\
&\quad \leq \sum_x \Pr[X = x] \left| \Pr[\forall i \in [m] : S_x^i(Y_i) = 1] - \prod_{i=1}^{m} \Pr[S_x^i(U_t) = 1] \right| \\
&\quad \leq \varepsilon.
\end{aligned}$$

□

We now round up the ideas above to prove Theorem 5.5.

Proof of Theorem 5.5. We instantiate two copies of the generator G_{base} with independent seeds, stretch each to at least $n/2$ bits using Lemma 5.17, and then combine them by interleaving blocks of t bits. Details follow.

For simplicity, we assume n is a multiple of $2t$. Plugging in G_{base} with $r = \log(n/2t)$ in Lemma 5.17, we obtain a generator $G' \colon \{0,1\}^{d'} \to (\{0,1\}^t)^{n/2t}$ with

$$d' = d_{\text{base}} + O(\log(n/t)\log(1/\varepsilon_{\text{base}}))$$

that $(n/2t)$-simultaneously ε'-fools width-w, length-t unanimity programs, where $\varepsilon' = \varepsilon_{\text{base}} \cdot (n/2t)^{\log 3}$.

For the actual construction of the generator $G \colon \{0,1\}^d \to \{0,1\}^n$ from the claim, we shall use two copies of G' (with independent seeds), which for convenience we denote G_1 and G_2. In turn, for $i \in \{1,2\}$, the components in the output of $G_i(\cdot)$ are denoted $G_i(\cdot)_1, \ldots, G_i(\cdot)_{n/2}$. Then G simply interleaves the outputs of G_1 and G_2; that is,

$$G(s_1, s_2) = G_1(s_1)_1 G_2(s_2)_1 G_1(s_1)_2 G_2(s_2)_2 \cdots G_1(s_1)_{n/2t} G_2(s_2)_{n/2t}$$

for $s_1, s_2 \in \{0,1\}^{d'}$. Hence, the seed length of G is $2d'$, and naturally G is explicit. The correctness of G follows directly from Lemmas 5.17 and 5.18. □

5.5. Pseudorandom Generators for δ-critical SWBPs

In this section, we shall prove Theorem 5.7. First we present a couple observations that are needed for the proof. As in Section 5.1.3, once $\delta > 0$ is fixed, we refer to a δ-critical layer of an SWBP simply as *critical* and the others as *non-critical*, accordingly. If we have a lower bound on the acceptance probability of a δ-critical SWBP S, then a simple observation allows us to upper-bound the number of critical layers in S:

Lemma 5.12. *Let $t \in \mathbb{N}_+$ and $\varepsilon, \delta > 0$, and let S be a δ-critical SWBP of window size t that accepts at least an ε fraction of its inputs, that is, $\Pr[S(U_n) = 1] \geq \varepsilon$. Then there are at most $O((t/\delta)\log(1/\varepsilon))$ critical layers in S.*

Proof. If S has $K = (t/\delta)\ln(1/\varepsilon)$ critical layers (where ln denotes the natural logarithm), then by the pigeonhole principle it has at least K/t layers that are at least t layers apart from one another. Hence, by the sliding-window property, for an uniformly chosen input x, the events of x passing through an accept or reject state in each of these layers are all independent. It follows that the probability that S accepts a uniformly chosen input is upper-bounded by $(1-\delta)^{K/t} < \varepsilon$. □

We shall also use the following simple property of hash functions.

Lemma 5.19. *Let $\mathcal{H} = \{h\colon [N] \to [M]\}$ be a family of pairwise independent hash functions, and let $S \subseteq [N]$ be some set. Then, for all but at most an $|S|^2/M$ fraction of $h \in H$, we have:*

$$\forall x, y \in S : x \neq y \implies h(x) \neq h(y)$$

(i.e., the restriction of h to S is injective).

Proof. Since H is pairwise independent, for any fixed $x, y \in S$, $x \neq y$, and h drawn uniformly at random from H, we have $\Pr[h(x) = h(y)] = 1/M$. Hence, by a union bound,

$$\Pr[\exists x, y \in S : x \neq y \wedge h(x) = h(y)] < \frac{|S|^2}{M}.$$ □

We are now in position to prove our result.

Theorem 5.7. *Let $n, w, t \in \mathbb{N}_+$ with $n \leq w$ and $\varepsilon > 0$ be given, and let $G_{\text{base}}\colon \{0,1\}^{d_{\text{base}}} \to \{0,1\}^t$ be a PRG that $\varepsilon_{\text{base}}$-fools width-$w$, length-$t$ unanimity programs where $\varepsilon_{\text{base}} \leq \varepsilon^{1+\log 3}$. Then there is an explicit PRG $G\colon \{0,1\}^d \to \{0,1\}^n$ with seed length*

$$d = 2d_{\text{base}} + O(\log(t/\delta)) \cdot \tilde{O}(\log(1/\varepsilon)) + O(\log(n/t))$$

that ε_G-fools δ-critical width-w, length-n SWBPs of window size t, where $\varepsilon_G = \tilde{O}(\varepsilon) \cdot (t/\delta)^{O(1)}$.

Proof. Let $K = O((t/\delta)\log(1/\varepsilon))$ be the upper bound from Lemma 5.12. We use a similar strategy as in the proof of Theorem 5.5, using two independent copies of the same generator G' together with interleaving. However, since now the number of critical layers is small compared to n, we will only need G' to K-simultaneously (instead of $(n/2t)$-simultaneously) fool length-t unanimity programs. We then "stretch" the output of G' to $n/2t$ components by reusing its components in the order dictated by a pairwise independent hash function. Details follow.

Construction. Let $M = K^2/\varepsilon$. We start by plugging in G_{base} in Lemma 5.17 with $r = \log M$, thus yielding a generator that M-simultaneously ε'-fools width-w, length-t unanimity programs, where

$$\varepsilon' = \varepsilon_{\text{base}} \cdot M^{\log 3} = \tilde{O}(\varepsilon) \cdot (t/\delta)^{O(1)}.$$

Again, we instantiate two copies of this generator, denoted G_1 and G_2, which will use independent seeds s_1 and s_2, respectively. For $i \in \{1, 2\}$, the components in the output of $G_i(\cdot)$ are denoted $G_i(\cdot)_1, \ldots, G_i(\cdot)_{K/t}$. We independently draw two hash functions $h_1, h_2 \colon [n/2t] \to [M]$ and set

$$G_{h_i}(s_i) = G_i(s_i)_{h(1)} G_i(s_i)_{h(2)} \cdots G_i(s_i)_{h(n/2t)}$$

for $i \in \{1, 2\}$. Finally, we set

$$G(s_1, s_2; h_1, h_2) = G_{h_1}(s_1) \parallel G_{h_2}(s_2).$$

Correctness. Let S be any length-n, δ-critical SWBP of window size t, and let $C \subseteq [n]$ be the set of layers that are critical in S. For simplicity, we assume n is a multiple of t. Since we are dividing the input of S into blocks of size t, it shall be more natural to think in terms of the set

$$B = \left\{ \left\lfloor \frac{i}{t} \right\rfloor \mid i \in C \right\}$$

of critical *blocks* of S. In fact, because G results from the interleaving of G_{h_1} and G_{h_2}, we consider

$$B_1 = \left\{ \frac{s+1}{2} \mid s \in S \text{ odd} \right\}$$

and

$$B_2 = \left\{ \frac{s}{2} \mid s \in S \text{ even} \right\}$$

separately. By Lemma 5.12, we have $|B_i| \leq K$ for $i \in \{1, 2\}$. Moreover, by Lemma 5.19, the probability that (the restriction of) h_1 is not injective on B_1 or h_2 is not injective on B_2 (or both) is at most $O(\varepsilon')$.

Hence, suppose that h_i is injective on B_i for $i \in \{1, 2\}$. We now argue as in the proof of Lemma 5.18. Letting $X = G_{h_1}(U_{d_{\text{base}}})$ and $Y = G_{h_2}(U_{d_{\text{base}}})$ (where the two occurrences of $U_{d_{\text{base}}}$ in X and Y are independent), we will show that, for every SWBP S as in the theorem's statement, $\Pr[S(X \parallel Y) = 1]$ is $O(\varepsilon')$-close to $\Pr[S(X \parallel U) = 1]$, where U is a random variable that is uniformly distributed on $\{0, 1\}^{n/2}$. By a similar argument, $\Pr[S(X \parallel U) = 1]$ is $O(\varepsilon')$-close to $\Pr[S(U \parallel U) = 1]$ (where the two occurrences of U are independent copies of the same random variable). From this it follows that $\Pr[S(G(U_d)) = 1]$ is $O(\varepsilon')$-close to $\Pr[S(U_n) = 1]$, yielding the theorem.

To show the above, fix some $x = x_1 \cdots x_{n/2} \in \text{Supp}(X)$ and let $S_x^1, \ldots, S_x^{n/2t}$ be as in the proof of Lemma 5.18. The point is that, for any $y \in \{0, 1\}^{n/2}$, $S(x \parallel y) = 1$ if and only if $S_x^i(y_i) = 1$ for every $i \in B_2$ (instead of just every $i \in [n/2t]$). Using that the restriction of h_2 to B_2 is injective, we find pairwise distinct $z_1, \ldots, z_{K/t}$ so that $h_2(z_i) = i$

and $B_2 \subseteq \{z_1, \dots, z_{K/t}\}$.[1] Hence, we can "glue" the $S_x^{z_i}$ together and obtain a SWBP S' such that $S'(y_{z_1} \cdots y_{z_{K/t}}) = 1$ if and only if $S_x^i(y_{z_i}) = 1$ for every $i \in [K/t]$. To construct S', proceed as follows:

1. Concatenate the layers of $S_x^{z_1}, \dots, S_x^{z_{K/t}}$ (in this order). Have a state be accepting in S' if and only if it is also accepting in the corresponding layer of $S_x^{z_i}$.
2. For the states belonging to both $S_x^{z_i}$ and $S_x^{z_{i+1}}$, have the state be accepting if and only if it is accepting in both $S_x^{z_i}$ and $S_x^{z_{i+1}}$.

Since $S(x \parallel G_{h_2}(s_2)) = 1$ if and only if $S'(G_2(s_2)) = 1$, the claim then follows from the pseudorandomness of G_2. □

5.6. Application to Sublinear-Time Probabilistic Cellular Automata

For the results in this section, we assume the reader is familiar with the theory of cellular automata. (See, e.g., [30] for a standard reference.)

5.6.1. Probabilistic Cellular Automata

As mentioned in the introduction, the PACA model was defined in Chapter 4. We repeat here the definitions for the reader's convenience.

We consider only bounded one-dimensional cellular automata.

Definition 5.20 (Cellular automaton)**.** A *cellular automaton* is a triple $C = (Q, \$, \delta)$ where Q is the finite set of *states*, $\$ \notin Q$ is the *boundary symbol*, and $\delta\colon Q_\$ \times Q \times Q_\$ \to Q$ is the *local transition function*, where $Q_\$ = Q \cup \{\$\}$. The elements in the domain of δ are the possible *local configurations* of the cells of C. For a fixed width $n \in \mathbb{N}_+$, the *global configurations* of C are the elements of Q^n. The cells 1 and n are the *border cells* of C. The *global transition function* $\Delta\colon Q^n \to Q^n$ is obtained by simultaneous application of δ everywhere; that is, if $s \in Q^n$ is the current global configuration of C, then

$$\Delta(s) = \delta(\$, s_1, s_2)\, \delta(s_1, s_2, s_3) \cdots \delta(s_{n-1}, s_n, \$).$$

For $t \in \mathbb{N}_0$, Δ^t denotes the t-th iterate of Δ. For an initial configuration $s \in Q^n$, the sequence $s = \Delta^0(s), \Delta(s), \Delta^2(s), \dots$ is the *trace* of C (for s). Writing the trace of C line for line yields its *space-time diagram*. Finally, for a cell $i \in [n]$ and $r \in \mathbb{N}_0$, the cells in $[i - r, i + r] \cap [n]$ form the *r-neighborhood* of i.

[1] Here we implicitly assume that h_2 is surjective since it is the more interesting case. Nevertheless, even if h_2 is not surjective, the same construction for S' applies by simply setting the respective layers to the trivial unanimity program that accepts all inputs.

Definition 5.21 (DACA)**.** A *DACA* is a cellular automaton C with an *input alphabet* $\Sigma \subseteq Q$ as well as a subset $A \subseteq Q$ of *accepting states*. We say C *accepts* an input $x \in \Sigma^+$ if there is $t \in \mathbb{N}_0$ such that $\Delta^t(x) \in A^n$, and we denote the set of all such x by $L(C)$. In addition, C is said to have *time complexity* (bounded by) $T\colon \mathbb{N}_+ \to \mathbb{N}_0$ if, for every $x \in L(C) \cap \Sigma^n$, there is $t < T(|x|)$ such that $\Delta^t(x) \in A^n$.

Definition 5.22 (PACA)**.** Let Σ be an alphabet and Q a finite set of states with $\Sigma \subseteq Q$. A *probabilistic ACA* (*PACA*) C is a cellular automaton with two local transition functions $\delta_0, \delta_1\colon Q^3 \to Q$. At each step of C, each cell tosses a fair coin $c \in \{0,1\}$ and updates its state according to δ_c; that is, if the current configuration of C is $s \in Q^n$ and the result of the cells' coin tosses is $r = r_1 \cdots r_n \in \{0,1\}^n$ (where r_i is the coin toss of the i-th cell), then the next configuration of C is

$$\Delta_r(s) = \delta_{r_1}(\$, s_1, s_2)\, \delta_{r_2}(s_1, s_2, s_3)\, \cdots\, \delta_{r_n}(s_{n-1}, s_n, \$).$$

Seeing this process as a Markov chain M over Q^n, we recast the global transition function $\Delta = \Delta_{U_n}$ as a family of random variables $(\Delta(s))_{s \in Q^n}$ parameterized by the current configuration s of C, where $\Delta(s)$ is sampled by starting in state s and performing a single transition on M (having drawn the cells' coin tosses according to U_n). Similarly, for $t \in \mathbb{N}_0$, $\Delta^t(s)$ is sampled by starting in s and performing t transitions on M.

A *computation* of C for an input $x \in \Sigma^n$ is a path in M starting at x. The computation is *accepting* if the path visits A^n at least once. In addition, in order to be able to quantify the probability of a PACA accepting an input, we additionally require for every PACA C that there is a function $T\colon \mathbb{N}_+ \to \mathbb{N}_0$ such that, for any input $x \in \Sigma^n$, every accepting computation for x visits A^n for the first time in strictly less than $T(n)$ steps; that is, if there is $t \in \mathbb{N}_0$ with $\Delta^t(x) \in A^n$, then $\Delta^{t_1}(x) \in A^n$ for some $t_1 < T(n)$. (Hence, every accepting computation for x has an initial segment with endpoint in A^n and whose length is strictly less than $T(n)$.) If this is the case for any such T, then we say C has *time complexity* (bounded by) T.

With this restriction in place, we may now equivalently replace the coin tosses of C with a matrix $R \in \{0,1\}^{T(n)\times n}$ of bits with rows $R_0, \ldots, R_{T(n)-1}$ and such that $R_j(i)$ corresponds to the coin toss of the i-th cell in step j. (If C accepts in step t, then the coin tosses in rows $t, \ldots, T(n)-1$ are ignored.) We refer to R as a *random input* to C. Blurring the distinction between the two perspectives (i.e., online and offline randomness), we write $C(x,R) = 1$ if C accepts x when its coin tosses are set according to R, or $C(x,R) = 0$ otherwise.

Definition 5.23 (p-error PACA)**.** Let $L \subseteq \Sigma^*$ and $p \in [0,1)$. A *one-sided p-error PACA for L* is a PACA C with time complexity T such that, for every $x \in \Sigma^n$, the following holds:

$$\begin{aligned} x \in L &\iff \Pr[C(x, U_{T(n)\times n}) = 1] \geq 1 - p \qquad \text{and} \\ x \notin L &\iff \Pr[C(x, U_{T(n)\times n}) = 1] = 0. \end{aligned}$$

If $p = 1/2$, then we simply say C is a *one-sided error PACA*. Similarly, for $p < 1/2$, a *two-sided p-error PACA for L* is a PACA C such that, for every $x \in \Sigma^*$, the following holds:

$$x \in L \iff \Pr[C(x, U_{T(n)\times n}) = 1] \geq 1 - p \qquad \text{and}$$
$$x \notin L \iff \Pr[C(x, U_{T(n)\times n}) = 1] \leq p.$$

If $p = 1/3$, then we simply say C is a *two-sided error PACA*. In both cases, we write $L(C) = L$ and say C *accepts* L.

A novelty in this chapter are δ-critical PACAs. In a sense, the notion gives a quantitative view of the concept of critical cells that is central to the results of Chapter 4 (see Lemma 4.15). (A *critical cell* is a cell that can either accept or not with non-zero probability.)

Definition 5.24 (δ-critical PACA)**.** Let $\delta > 0$. For a time step $T \in \mathbb{N}_0$, a PACA C is said to be *δ-critical with respect to T* if, on any input $x \in \Sigma^n$ to C, the following holds for every cell $i \in n$:

1. Either i is always accepting in time step T; or
2. The probability that i is not accepting in time step T is at least δ.

We say C is simply *δ-critical* if it is δ-critical with respect to every time step in which it accepts with non-zero probability.

5.6.2. Simulating a PACA with a Low-Space Sliding-Window Algorithm

We now show how a PACA can be simulated by a randomized sliding-window algorithm with low space. This can be achieved with little difficulty simply by adapting the streaming algorithm from Theorem 3.4. (The algorithm there is geared toward a different variant of cellular automata, but the same strategy works fairly well in our setting as well.) For the sake of self-containedness, we provide the adaptation in full.

Theorem 5.25. *Let C be a (one- or two-sided error) PACA with state set Q and $T \in \mathbb{N}_0$. Then there is a $O(T \log|Q|)$-space randomized non-uniform sliding-window algorithm $S = S_T$ of window size $O(T^2)$ such that*

$$\Pr[S(x) = 1] = \Pr[C \text{ accepts } x \text{ in time step } T].$$

We note the non-uniformity of S is required only to set T. Every other aspect of S is realized in an uniform manner.

The basic idea involved is that, in order to emulate the behavior of C on an input x, it suffices to move a sliding window over its time-space diagram that is T cells long and $O(1)$ cells wide (see Figure 5.1), feeding random bits to the cells as needed. Every time a new symbol from x is read, the window is moved one cell to the right; if it positioned beyond

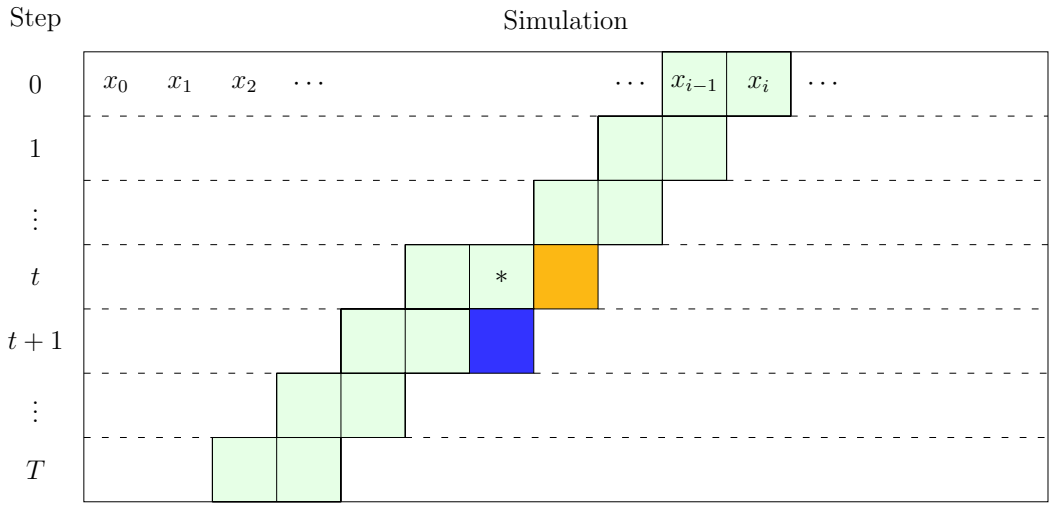


Figure 5.1.: Simulation of C by the sliding-window algorithm S. In the picture, S has last read the input symbol x_i and is now determining the state of the cell marked with an asterisk in time step $t+1$ (depicted in blue). The light green cells are the ones maintained by S in its stateLeft and stateCenter arrays, while the orange cell is the one corresponding to stateRight.

the borders of C, then the respective part of the window is filled with the border symbol \$. If S notes that any one cell in step T is not accepting, then it immediately halts and rejects; otherwise it reads the entire input and eventually accepts.

Proof. We prove that Algorithm 2, hereafter referred to as S, satisfies the properties in the claim. For simplicity of presentation, in this proof the input $x = x_0 \cdots x_{n-1}$ as well as the cells of the PACA C are indexed starting with zero.

We first address the correctness of S. Fix a random input $r \in \{0,1\}^{(n+T)T}$ to S. It shall be convenient to recast r as a matrix $R \in \{0,1\}^{T\times(n+T)}$ where $R(i,j) = r(i+jT)$; that is, the j-th column of R corresponds to the randomness used by S in the j-th run of the outer **while** loop, and the i-th row of said column equals the value of b in the i-th run of the inner **while** loop. In addition, let $R' \in \{0,1\}^{T\times n}$ be the matrix with $R'(i,j) = R(i, i+j+1)$. As we shall see, R' corresponds exactly to the random input to C in its simulation by S and the bits of R that do not have a corresponding entry in R' do not affect the outcome of S.

By $D = D_{R'} : \{0,\dots,T\} \times \mathbb{Z} \to Q_\$$ we denote the time-space diagram of C when using coin tosses from R' where \$ is used to fill states "beyond the borders" of C; that is, for $i \in [n]$, $D(t,i)$ equals the state of the i-th cell in the t-th step of C on input x when using the coin tosses given by R', or $D(t,i) = \$$ otherwise. We shall show the following invariants are satisfied by the outer loop (line A):

I_1: For every $t \in \{0,\dots,T-1\}$, $\mathsf{stateCenter}[t] = D(t, i-t-1)$ and $\mathsf{stateLeft}[t] = D(t, i-t-2)$.

I_2: S has not rejected if and only if, for every $i' < i$, $D(T, i'-T)$ is either accepting or equal to \$.

Algorithm 2: Randomized sliding-window algorithm S

```
for t ← 0, ..., T − 1 do
    stateLeft[t] ← $;
    stateCenter[t] ← $;
end
i ← 0;
A while i < n + T do
    if i < n then
        stateRight ← x_i;
    else
        stateRight ← $;
    end
    t ← 0;
B   while t < T do
C       sample b ∈ {0, 1} from randomness source;
        newState ← δ_b(stateLeft[t], stateCenter[t], stateRight);
        stateLeft[t] ← stateCenter[t];
        stateCenter[t] ← stateRight;
        stateRight ← newState;
        t ← t + 1;
    end
    if stateRight ≠ $ and stateRight is not accepting then
        reject;
    end
    i ← i + 1;
end
accept;
```

In particular, I_2 directly implies the correctness of S (if we also have that S depends only on the random bits of R that have a corresponding entry in R', which is indeed the case).

We prove I_1 and I_2 by showing the inner loop (line B) satisfies an invariant I_3 of its own, namely that $\mathsf{stateRight} = D(t, i - t)$. Concretely, if we have that I_1 and I_2 hold prior to the i-th execution of the outer loop and I_3 holds at the end of the inner loop, then I_1 and I_2 are conserved as follows: The invariant I_1 holds since the instructions executed ensure that $\mathsf{stateLeft}[t] = D(t, i - t - 1)$ and $\mathsf{stateCenter}[t] = D(t, i - t)$ hold for every t at the end of the inner loop and i is incremented at the end. Similarly, since $t = T$ holds after the inner loop is done, we have then $\mathsf{stateRight} = D(t, i - T)$ after the loop, implying I_2.

To show I_3 is an invariant, suppose I_1 holds for some i. Clearly, $\mathsf{stateRight} = D(0, i)$ holds prior to its first execution of the loop since then $\mathsf{stateRight} = x(i) = D(0, i)$ if $i < n$, or

stateRight = \$ = $D(0, i)$ otherwise. Subsequently, in the t-th execution of the loop, if $t + 1 \leq i \leq n + t$, then b is set to $R(t, i) = R'(t, i - t - 1)$ and stateRight to

$$\begin{aligned}\delta_b(\mathsf{stateLeft}[t], \mathsf{stateCenter}[t], \mathsf{stateRight}) &= \delta_b(D(t, i - t - 2), D(t, i - t - 1), D(t, i - t)) \\ &= D(t + 1, i - t - 1).\end{aligned}$$

If $i < t + 1$, then $i - t - 1 < 0$, and so $\mathsf{stateLeft}[t] = \mathsf{stateCenter}[t] = \$$, which means stateRight is set to \$ regardless of the value of $R(t, i)$; the same is the case for $i > n+t$ since then $i - t - 1 > n - 1$. Hence, the operation of S depends only on $R(t, i)$ if $t + 1 \leq i \leq n + t$, which is precisely the case when $R'(t, i - t - 1) = R(t, i)$. Since t is incremented at the end of the loop, it follows that I_3 is an invariant, as desired.

The space complexity of S evident since it is dominated by the arrays stateLeft and stateRight, which both contain T many elements of Q. Finally, regarding the sliding window size of S, we can argue based on the invariants above and the properties of the time-space diagram D. Clearly, an entry $R'(t, i)$ only affects the states $D(t + j, i + k)$ for $j \in \{1, \ldots, T - t\}$ and $k \in \mathbb{Z}$ with $|k| < j$ (since, for every t and every i, $R'(t, i)$ only affects $D(t + 1, i)$ and $D(t, i)$ only $D(t + 1, i - 1)$, $D(t + 1, i)$, and $D(t + 1, i + 1)$). Hence, after having read $R'(t, i)$, if S reads another $2T^2$ random bits, then its state will be independent of the entry $R'(t, i)$. This means that S has a window size of $O(T^2)$, as desired. □

5.6.3. Derandomizing Sublinear-Time PACAs with Small Space

In this final section, we recall and prove our results on obtaining low-space algorithms from PACAs. We start with the results Theorems 5.9 and 5.10 on one-sided error PACAs.

5.6.3.1. One-Sided Error PACAs

Theorem 5.9. *Let $T : \mathbb{N}_+ \to \mathbb{N}_+$ be a (constructible) function and $\varepsilon \geq 1/\mathrm{poly}(T)$. For any one-sided ε-error PACA C that recognizes its language $L(C)$ in time at most $T = T(n)$, there is a deterministic algorithm for $L(C)$ with space complexity $O(T + (\log n)^2)$.*

The proof is more or less straightforward: Given a T-time one-sided ε-error PACA C as in the theorem's statement, we use the construction from Theorem 5.25 together with the PRG from Corollary 5.6 to obtain good estimates for the probability that C accepts for every time step $t < T$. Then we show we can use these estimates to determine whether C accepts the input or not. This is where we use that C has only one-sided error: If C accepts, then by an averaging argument there is at least one time step t in which C accepts with probability that is "far enough" from 0; conversely, if C rejects, then all estimates will be "near" 0.

Proof. We construct an algorithm A for $L(C)$ with the desired space complexity. Let G be the PRG from Corollary 5.6 that ε_G-fools SWBPs of width $2^{O(T)}$, length $m = (n + T)T$, and window size $O(T^2)$, where $\varepsilon_G = \varepsilon/4T \geq 1/\mathrm{poly}(T)$. In addition, for $t < T$, let S_t be the sliding-window algorithm of Theorem 5.25. Our algorithm A operates as follows:

1. For every $t < T$, enumerate over every possible seed to G and simulate S_t on its output to obtain an estimate $\eta_t = \Pr[S_t(G(U_d)) = 1]$ that is ε_G-close to $\Pr[S_t(U_m) = 1]$.
2. If there is t such that $\eta_t \geq \varepsilon/2T$, accept; otherwise reject.

Observe that A has the required space complexity because, by our setting of parameters, G has seed length $d = O(T + (\log n)^2)$ and, in addition, every S_t can be executed in $O(T)$ space. For the correctness of A, fix some input x to C and consider the two cases:

$x \in L(C)$. Let $Z_t(R)$ denote the event that C accepts x in step t using coin tosses from $R \in \{0,1\}^{T \times n}$. By a union bound, we have

$$\Pr[C(x, U_{T\times n}) = 1] \leq \sum_{t=0}^{T-1} \Pr[Z_t(U_{T\times n})] = \sum_{t=0}^{T-1} \Pr[S_t(U_m) = 1],$$

where the last equality is due to Theorem 5.25. By an averaging argument, there is t such that $\Pr[S_t(U_m) = 1] \geq \varepsilon/T$, which means that

$$\Pr[S_t(G(U_d)) = 1] \geq \frac{\varepsilon}{T} - \varepsilon_G > \frac{\varepsilon}{2T}.$$

$x \notin L(C)$. Since the probability that C accepts x is zero, we also have $\Pr[S_t(U_m) = 1] = 0$ for every t. It follows that

$$\Pr[S_t(G(U_d)) = 1] \leq \varepsilon_G < \frac{\varepsilon}{2T}.$$ □

We shall use essentially the same strategy to obtain:

Theorem 5.10. *Let $T\colon \mathbb{N}_+ \to \mathbb{N}_+$ be a (constructible) function, and let $\delta > 0$ and $\varepsilon \geq 1/\mathrm{poly}(T)$. For any δ-critical one-sided ε-error PACA C that recognizes its language $L(C)$ in time at most $T = T(n)$, there is a deterministic algorithm for $L(C)$ with space complexity*

$$O\left(\frac{(T + \log(1/\delta))\log T}{\max\{1, \log T - \log\log(n/\delta)\}}\right) + \tilde{O}(\log(T/\delta))^2 + O(\log n).$$

The difference is that we now apply Corollary 5.8 while also having to account for δ-criticiality.

Proof. We use the same algorithm A as in the proof of Theorem 5.9. The parameters for G are also the same but now we use the PRG from Corollary 5.8 that targets δ-critical SWBPs. By our setting of parameters, G has seed length

$$d = O\left(\frac{(T + \log(1/\delta))\log T}{\max\{1, \log T - \log\log(n/\delta)\}}\right) + \tilde{O}(\log(T/\delta))^2 + O(\log n),$$

which again dominates the space complexity of A. The same analysis as in the proof of Theorem 5.9 applies since, as C is δ-critical, lifting S_t to a (non-uniform) SWBP yields a δ-critical SWBP (since each layer of the resulting SWBP simply corresponds in an entry of the time-space diagram D of C; see the proof of Theorem 5.25). □

5.6.3.2. Two-Sided Error PACAs

For two-sided error PACAs, we would like to achieve a result as Theorem 5.9 using a similar apporach. Unfortunately, the same strategy as in the proof of Theorem 5.9 fails even if we have a means of determining the probability that the PACA accepts in every single time step t *without any error*.

We illustrate why by means of an example. For concreteness, let the input alphabet be $\Sigma = \{0, 1\}$. Consider the PACAs C_1 and C_2 that operate as follows: Every cell except for the first one is unconditionally in an accepting state. During the first 8 steps, the leftmost cell collects random bits to form a random string $r \in \{0, 1\}^8$. Following this, in the next steps, the cell behaves as follows:

- In C_1, the cell turns accepting if and only if the first two bits of r are equal to zero (i.e., $r_1 = r_2 = 0$). The cell remains accepting for exactly four steps, then changes into a non-accepting state and maintains it.
- In C_2, in each of the subsequent four steps $j \in [4]$, the cell turns accepting in step j if and only if $r_{2j-1} = r_{2j} = 0$. After this, the cell assumes a non-accepting state and maintains it.

It is easy to see that both C_1 and C_2 are two-sided error PACAs that accept completely different languages: The probability that C_1 accepts any input $w \in \{0, 1\}^*$ is $1/4$, which means $L(C_1) = \varnothing$; conversely, C_2 accepts any input with probability $1 - (3/4)^4 > 2/3$, thus implying $L(C_2) = \{0, 1\}^*$. Nevertheless, the probability that the PACA accepts at any *fixed* time step t (considered on its own) is *the same* in either of the C_i.

This means that a new strategy is called for. One possible solution would be to adapt the algorithm S from Theorem 5.25 so that it checks multiple steps of the PACA for acceptance (e.g., maintain a Boolean variable $\texttt{accept}_i$ for every step i initially set to true and set it to false if any non-accepting state in step i is seen). Unfortunately, it does not seem to be possible to do so without forgoing the sliding-window property. Nevertheless, this strategy *can* be applied (while still yielding a sliding-window algorithm) if instead of a single time step t we have S verify that the PACA is accepting in every step that is in a *subset* $t \subseteq [T]$ (that is provided to S non-uniformly). Indeed, this is a straightforward adaptation and it is not hard to see that the resulting algorithm S_t is such that

$$\Pr[S_t(x) = 1] = \Pr[\forall i \in t : \text{on input } x, C \text{ is accepting in time step } i]$$

where the second probability is conditioned on the coin tosses of C. (Hence, the original algorithm works in the special case where $|t| = 1$.) Using the inclusion-exclusion principle and our PRGs of before (with appropriate parameters), we can then obtain good estimates for the probability that C accepts and proceed in almost the same fashion as in the proof of Theorem 5.9.

Theorem 5.26. *Let $T : \mathbb{N}_+ \to \mathbb{N}_+$ be a function and $\varepsilon \geq 2^{-O(T)}$. For any two-sided $(1/2 - \varepsilon)$-error PACA C that recognizes its language $L(C)$ in time at most $T = T(n)$, there is a deterministic algorithm for $L(C)$ with space complexity $O(T \log T + (T + \log n) \log(n/T^2))$.*

Proof. We construct an algorithm A for $L(C)$ with the desired space complexity. Let G be the PRG from Corollary 5.6 that ε_G-fools SWBPs of width $2^{O(T)}$, length $m = (n+T)T$, and window size $O(T^2)$, where $\varepsilon_G = \varepsilon/2^{2T} \geq 2^{-O(T)}$. (These are the same parameters as in the proof of Theorem 5.9 except for ε_G, which we now need to be exponentially small in T.) For $\emptyset \neq t \subseteq [T]$, let S_t be the generalization of algorithm of Theorem 5.25 as previously described. The algorithm A proceeds as follows:

1. For every $\emptyset \neq t \subseteq [T]$, enumerate over every possible seed to G and simulate S_t on its output to obtain an estimate $\eta_t = \Pr[S_t(G(U_d)) = 1]$ that is ε_G-close to $\Pr[S_t(U_m) = 1]$.
2. Compute
$$\eta = \sum_{\substack{t \subseteq [T] \\ |t|=1}} \eta_t - \sum_{\substack{t \subseteq [T] \\ |t|=2}} \eta_t + \cdots + (-1)^{T+1} \eta_{[T]}$$
and accept if $\eta > 1/2$; otherwise reject.

By our setting of parameters, G has seed length

$$d = O(T \log T + (T + \log n) \log(n/T^2)).$$

This also gives the space complexity of A as before. For the correctness, in a similar way as in the proof of Theorem 5.9 we let $Z_t(R)$ denote the event that C accepts x in every one of the steps in $\emptyset \neq t \subseteq [T]$ when using coin tosses from $R \in \{0,1\}^{T \times n}$. Note that, by the inclusion-exclusion principle, we have

$$\begin{aligned}
&\Pr[C(x, U_{T\times n}) = 1] \\
&\quad = \Pr[\exists t \in [T] : Z_{\{t\}}(U_{T\times n})] \\
&\quad = \sum_{\substack{t \subseteq [T] \\ |t|=1}} \Pr[Z_t(U_{T\times n})] - \sum_{\substack{t \subseteq [T] \\ |t|=2}} \Pr[Z_t(U_{T\times n})] + \cdots + (-1)^{T+1} \Pr[Z_{[T]}(U_{T\times n})].
\end{aligned}$$

Since there are strictly less than 2^T terms on the right-hand side in total and replacing $U_{T\times n}$ with $G(U_d)$ causes an additive deviation of at most ε_G in each term, it follows that

$$|\eta - \Pr[C(x, U_{T\times n} = 1)]| \leq \varepsilon_G 2^T < \varepsilon. \qquad \square$$

Unfortunately, (asymptotically speaking) Theorem 5.26 does not give us a better space complexity than we would have had by simply using the PRG of Armoni (see Section 5.1.2.2), which would already have given us a seed length of $d = O(T \log n)$. Nevertheless, the proof of Theorem 5.26 is very valuable since it can be easily generalizes to the case of δ-*critical* PACAs, in which case *we do get an improvement on the space complexity*:

Theorem 5.11. *Let $T : \mathbb{N}_+ \to \mathbb{N}_+$ be a (constructible) function, and let $\delta > 0$ and $\varepsilon \geq 2^{-O(T)}$. For any δ-critical two-sided $(1/2 - \varepsilon)$-error PACA C that recognizes its language $L(C)$ in time at most $T = T(n)$, there is a deterministic algorithm for $L(C)$ with space complexity $\tilde{O}(T \cdot (\log(1/\delta))^2) + O(\log n)$.*

Proof. Use the same algorithm A as in the proof of Theorem 5.26 with the PRG from Corollary 5.8 (with the same parameters) while noting that, for any t, lifting S_t to a (non-uniform) SWBP yields a δ-critical SWBP (as in the proof of Theorem 5.7). □

6. Embedding Arbitrary Boolean Circuits into Fungal Automata

Co-authored with Thomas Worsch

Published version: [86]

Abstract

Fungal automata are a variation of the two-dimensional sandpile automaton of Bak, Tang, and Wiesenfeld (Phys. Rev. Lett., 1987). In each step toppling cells emit grains only to *some* of their neighbors chosen according to a specific update sequence. We show how to embed any Boolean circuit into the initial configuration of a fungal automaton with update sequence HV. In particular we give a constructor that, given the description B of a circuit, computes the states of all cells in the finite support of the embedding configuration in $O(\log|B|)$ space. As a consequence the prediction problem for fungal automata with update sequence HV is P-complete. This solves an open problem of Goles et al. (Phys. Lett. A, 2020).

6.1. Introduction

The two-dimensional sandpile automaton by Bak, Tang, and Wiesenfeld [10] has been investigated from different points of view. Because of the simple local rule, it is easily generalized to the d-dimensional case for any integer $d \geq 1$.

Several *prediction problems* for these cellular automata (CA) have been considered in the literature. Their difficulty varies with the dimensionality. The recent survey by Formenti and Perrot [39] gives a good overview. For one-dimensional sandpile CA the problems are known to be easy (see, e.g., [78]). For d-dimensional sandpile CA where $d \geq 3$, they are known to be P-complete [89]. In the two-dimensional case the situation is unclear; analogous results are not known.

Fungal automata (FA) as introduced by Goles et al. [53] are a variation of the two-dimensional sandpile automaton where a toppling cell (i.e., a cell with state ≥ 4) emits 2 excess grains of sand either to its two horizontal ("H") or to its two vertical neighbors ("V"). These two modes of operation may alternate depending on an *update sequence* specifying in which steps grains are moved horizontally and in which steps vertically.

The construction in [53] shows that some natural prediction problem is P-complete for two-dimensional fungal automata with update sequence H^4V^4 (i.e., grains are first transferred horizontally for 4 steps and then vertically for 4 steps, alternatingly). The paper leaves open whether the same holds for shorter update sequences. The shortest non-trivial sequence is HV (and its complement VH); at the same time this appears to be the most difficult to use. By a reduction from the well-known *circuit value problem* (CVP), which is P-complete, we will show:

Theorem 6.1. *The following prediction problem is* P*-complete for FA with update sequence HV:*

Given *as inputs initial states for a finite rectangle R of cells, a cell index y (encoded in binary), and an upper bound T (encoded in unary) on the number of steps of the FA,*

decide *whether cell x is in a state $\neq 0$ or not at some time $t \leq T$ when the FA is started with R surrounded by cells all in state 0.*

We assume readers are familiar with cellular automata (see Section 6.2 for the definition). We also assume knowledge of basic facts about Boolean circuits and complexity theory, some of which we recall next.

6.1.1. Boolean circuits and the CVP

A Boolean circuit is a directed acyclic graph of *gates*: NOT gates (with one input), AND and OR gates with two inputs, $n \geq 1$ INPUT gates and one OUTPUT gate. The output of a gate may be used by an arbitrary number of other gates. Since a circuit is a dag and each gate obtains its inputs from gates in previous layers, ultimately the output of each gate can be computed from a subset of the input gates in a straightforward way.

It is straightforward to realize NOT, AND, and OR gates in terms of NAND gates with two inputs (with an only constant overhead in the number of gates). To simplify the construction later on, we assume that circuits consist exclusively of NAND gates.

Each gate of a circuit is described by a 4-tuple (g, t, g_1, g_2) where g is the number of the gate, t describes the type of the gate, and g_1 and g_2 are the numbers of the gates (called sources of g) that produce the inputs for gate g; all numbers are represented in binary. If gate g has only one input, then $g_2 = g_1$ by convention. Without loss of generality the INPUT gates have numbers 1 to n and since their predecessors g_1 and g_2 will never be used, assume they are set to 0. All other gates have subsequent numbers starting at $n + 1$ such that the inputs for gate g are coming from gates with strictly smaller numbers. Following Ruzzo [103] the description B of a complete circuit is the concatenation of the descriptions of all of its gates, sorted by increasing gate numbers.

Problem instances of the *circuit value problem* (CVP) consist of the description B of a Boolean circuit C with n inputs and a list x of n input bits. The task is to decide whether $C(x) = 1$ holds or not. It is well known that the CVP is P-complete.

6.1.2. Challenges

A standard strategy for showing P-completeness of a problem Π in some computational model $\mathcal{M}$ (and also the one employed by Goles et al. in [53]) is by a reduction from the CVP to Π, which entails describing how to "embed" circuits in $\mathcal{M}$.

In our setting of fungal automata with update sequence HV, while realizing wires and signals as in [53] is possible, there is no obvious implementation for negation nor for a reliable wire crossing. Hence, it seems one can only directly construct circuits that are *both* planar and monotone. Although it is known that the CVP is P-complete for *either* planar or monotone circuits [52], it is unlikely that one can achieve the same under both constraints. This is because the CVP for circuits that are both monotone and planar lies in NC^2 (and is thus certainly not P-complete unless $\mathsf{P} \subseteq \mathsf{NC}^2$) [32].

We are able to overcome this barrier by exploiting features that are present in fungal automata but not in general circuits: *time* and *space*. Namely, we deliberately *retard* signals in the circuits we implement by extending the length of the wires that carry them. We show how this allows us to realize a primitive form of transistor. From this, in turn, we are able to construct a NAND gate, thus allowing both wire crossings and negations to be implemented.

Our construction is not subject to the limitations that apply to the two-dimensional case that were previously shown by Gajardo and Goles in [43] since the FA starting configuration is not a fixed point. The resulting construction is also significantly more complex than that of [53].

6.1.3. Overview of the construction

In the rest of the chapter we describe how to embed any Boolean circuit with description B and an assignment of values to the inputs into a configuration c of a fungal automaton in such a way that the following holds:

- "Running" the FA for a sufficient number of steps results in the "evaluation" of all simulated gates. In particular, after reaching a stable configuration, a specific cell of the FA is in state 1 or 0 if and only if the output of the circuit is 1 or 0, respectively.
- The initial configuration F of the FA is simple in the sense that, given the description of a circuit and an input to it, we can produce its embedding F using $O(\log n + \log|B|)$ space. Thus we have a log-space reduction from the CVP to the prediction problem for FA.

The construction consists of several layers:

Layer 0: The underlying model of fungal automata.

Layer 1: As a first abstraction we subdivide the space into *blocks* of 2×2 cells and always think of update *cycles* consisting of 4 steps of the CA, using the update sequence $(HV)^2$.

Layer 2: On top of that we will implement *polarized circuits* processing *polarized signals* that run along *wires*.

Layer 3: Polarized circuitry is then used to implement *Boolean circuits with delay*: *bits* are processed by *gates* connected by *cables*.[1]

Layer 4: Finally a given Boolean circuit (without delay) can be embedded in a fungal automaton (as a circuit with delay) in a systematic fashion that needs only logarithmic space to construct.

The rest of this chapter has a simple organization: Each layer i will be described separately in section $i + 2$.

6.2. Layer 0: The Fungal Automaton

Let $\mathbb{N}_+$ denote the set of positive integers and $\mathbb{Z}$ that of all integers. For $d \in \mathbb{N}_+$, a *d-dimensional CA* is a tuple (S, N, δ) where:

- S is a finite set of states
- N is a finite subset of $\mathbb{Z}^d$, called the *neighborhood*
- $\delta\colon S^N \to S$ is the *local transition function*

In the context of CA, the elements of $\mathbb{Z}^d$ are referred to as *cells*. The function δ induces a *global transition function* $\Delta\colon S^{\mathbb{Z}^d} \to S^{\mathbb{Z}^d}$ by applying δ to each cell simultaneously. In the following, we will be interested in the case $d = 2$ and the so-called *von Neumann neighborhood* $N = \{(a, b) \in \mathbb{Z}^2 \mid |a| + |b| \leq 1\}$ of radius 1.

Except for the updating of cells the fungal automaton is just a two-dimensional CA with the von Neumann neighborhood of radius 1 and $S = \{0, 1, \ldots, 7\}$ as the set of states.[2] A *configuration* is thus a mapping $c : \mathbb{Z}^2 \to S$.

Depending on the their states cells will be depicted as follows in diagrams:

- State 0 as $\boxed{}$
- State 1 as $\boxed{\cdot}$
- State $i \in S \setminus \{0, 1\}$ as $\boxed{i}$

[1] Here we slightly deviate from the standard terminology of Boolean circuits and reserve the term *wire* for the more primitive wires defined in layer 2.

[2] We use states as in [10]; however, the states 6 and 7 never occur in our construction.

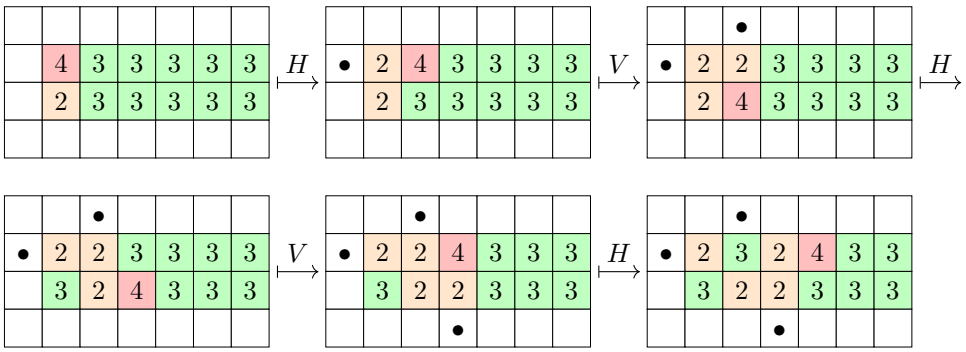


Figure 6.1.: Five transitions according to $HVHVH$

We will use colored background for cells in states 2, 3, and 4 since their presence determines the behavior of the polarized circuit. The state 1 is only a "side effect" of an empty cell receiving a grain of sand from some neighbors; hence it is represented as a dot. Cells that are not included in a figure are always assumed to be in state 0.

For a logical predicate P denote by $[P]$ the value 1 if P is true and the value 0 if P is false. For $i \in \mathbb{Z}^2$ denote by $h(i)$ the two horizontal neighbors of cell i and by $v(i)$ its two vertical neighbors. Cells are updated according to 2 functions H and V mapping from $S^{\mathbb{Z}^2}$ to $S^{\mathbb{Z}^2}$ where for each $i \in \mathbb{Z}^2$ the following holds:

$$H(c)(i) = c(i) - 2 \cdot [c(i) \geq 4] + \sum_{j \in h(i)} [c(j) \geq 4];$$
$$V(c)(i) = c(i) - 2 \cdot [c(i) \geq 4] + \sum_{j \in v(i)} [c(j) \geq 4].$$

The updates are similar to the sandpile model by Bak, Tang, and Wiesenfeld [10], but toppling cells only emit grains of sand either to their horizontal or their vertical neighbors. Therefore whenever a cell is non-zero, it stays non-zero forever.

The composition of these functions applying first H and then V is denoted HV. For the transitions of a fungal automaton with update sequence HV these functions are applied alternatingly, resulting in a computation

$$c, H(c), V(H(c)), H(V(H(c))), V(H(V(H(c)))),$$

and so on. In examples we will often skip three intermediate configurations and only show c, $HVHV(c)$, etc. Figure 6.1 shows a simple first example.

6.3. Layer 1: Coarse-Graining Space and Time

As a first abstraction from now on one should always think of the space as subdivided into *blocks* of 2×2 cells. Furthermore we will look at update *cycles* consisting of 4 steps of the CA, thus using the update sequence $HVHV$, which we will abbreviate as Z. As an example

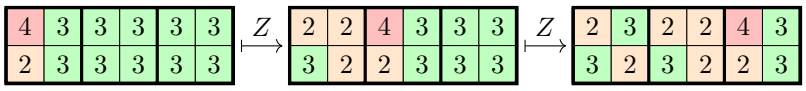


Figure 6.2.: Compact representation of two cycles

(a) ⊞ signals **(b)** ⊟ signals

Figure 6.3.: Representations of ⊞ and ⊟ signals

Figure 6.2 shows the same cycle as Figure 6.1 and the following cycle in a compact way. Block boundaries are indicated by thicker lines.

Cells outside the depicted area of a figure are assumed to be $\boxed{0}$ initially and they will never become critical and topple during the shown computation.

6.4. Layer 2: Polarized Components

We turn to the second lowest level of abstraction. Here we work with two types of signals, which we refer to as *positive* (denoted ⊞) and *negative* (denoted ⊟). Both types will have several representations as a block in the FA.

- All representations of a ⊞ signal have in common that the ***upper*** *left corner* of the block is a $\boxed{4}$ and the other cells are $\boxed{2}$ or $\boxed{3}$.
- All representations of a ⊟ signal have in common that the ***lower*** *left corner* of the block is a $\boxed{4}$ and the other cells are $\boxed{2}$ or $\boxed{3}$.

Not all representations will be appropriate in all situations as will be discussed in the next subsection.

The rules of fungal automata allow us to perform a few basic operations on these *polarized* signals (e.g., duplicating, merging, or crossing them under certain assumptions). The highlight here is that we can implement a (delay-sensitive) form of transistor that works with polarized signals, which we refer to as a *switch*.

As a convention, in the figures in this section, we write x and y for the inputs of a component and z, z_1, and z_2 for the outputs.

6.4.1. Polarized Signals and Wires

Representations of ⊞ and ⊟ signals are shown in Figure 6.3. We will refer to a block initially containing a ⊞ or ⊟ signal as a ⊞ or ⊟ *source*, respectively. (This will be used, for instance, to set the inputs to the embedded CVP instance.)

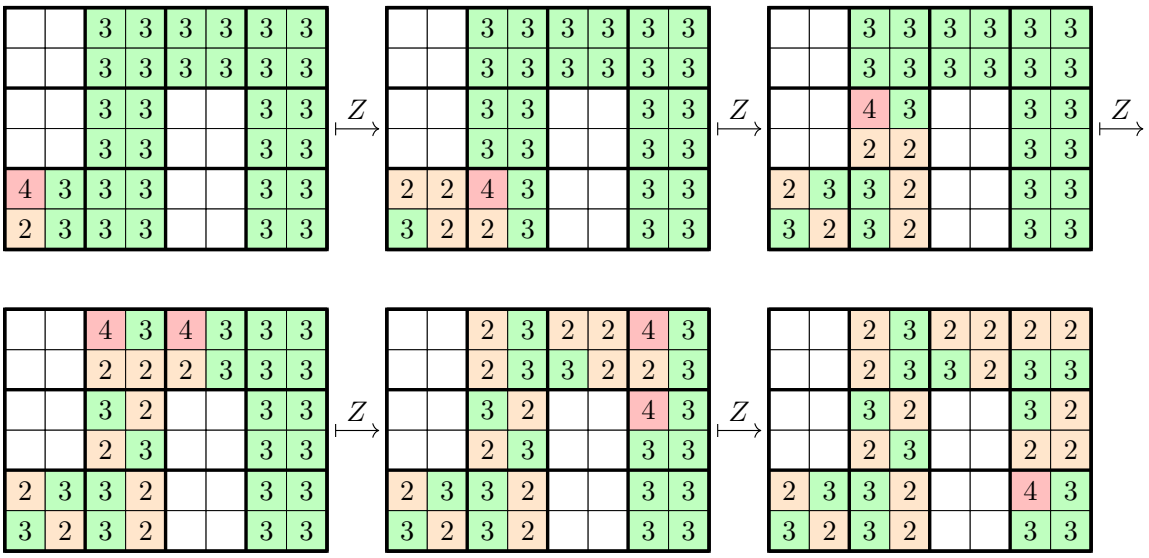


Figure 6.4.: A ⊞ signal moving along a wire with two right turns.

A comparison of Figure 6.2 and Figure 6.3a shows that in the former a ⊞ signal is "moving from left to right". In general we will use *wires* to propagate signals. Wires extending horizontally or vertically can be constructed by juxtaposing *wire blocks* consisting of 2×2 blocks of cells in state $\boxed{3}$.

While one can use the same wire blocks for both types of signals, each block is destroyed upon use and thus can only be used once. In particular, this means a wire will either be used by a ⊞ or a ⊟ signal. We refer to the respective wires as ⊞ and ⊟ *wires*, accordingly.

Every representation of a signal is restricted with respect to the possible directions it can move to along a wire. In our construction each signal will start at the left end of a horizontal wire. Figure 6.4 shows how a ⊞ signal first "turns left" once and then moves along a wire that "turns right" two times, changing its representation while meandering around. (The case of a ⊟ signal is similar and is not shown.)

Figure 6.5 can be seen as the continuation of Figure 6.4. The ⊞ signal moves further down, "turns left" twice, and then reaches the end of the wire. The composition of both parts can be seen in Figure 6.11 and will be used as the basic building unit for "retarders".

6.4.2. Diodes

Note that ⊞ and ⊟ signals do not encode any form of direction in them (regarding their propagation along a wire). In fact, a signal propagates in any direction a wire is placed in. In order for our components to operate correctly, it will be necessary to ensure a signal is propagated in a single direction. To realize this, we use *diodes*.

A diode is an element on a horizontal wire that only allows a signal to flow from left to right. A signal coming from right to left is not allowed through. As the other components, the diode is intended to be used only once. For the implementation, refer to Figure 6.6.

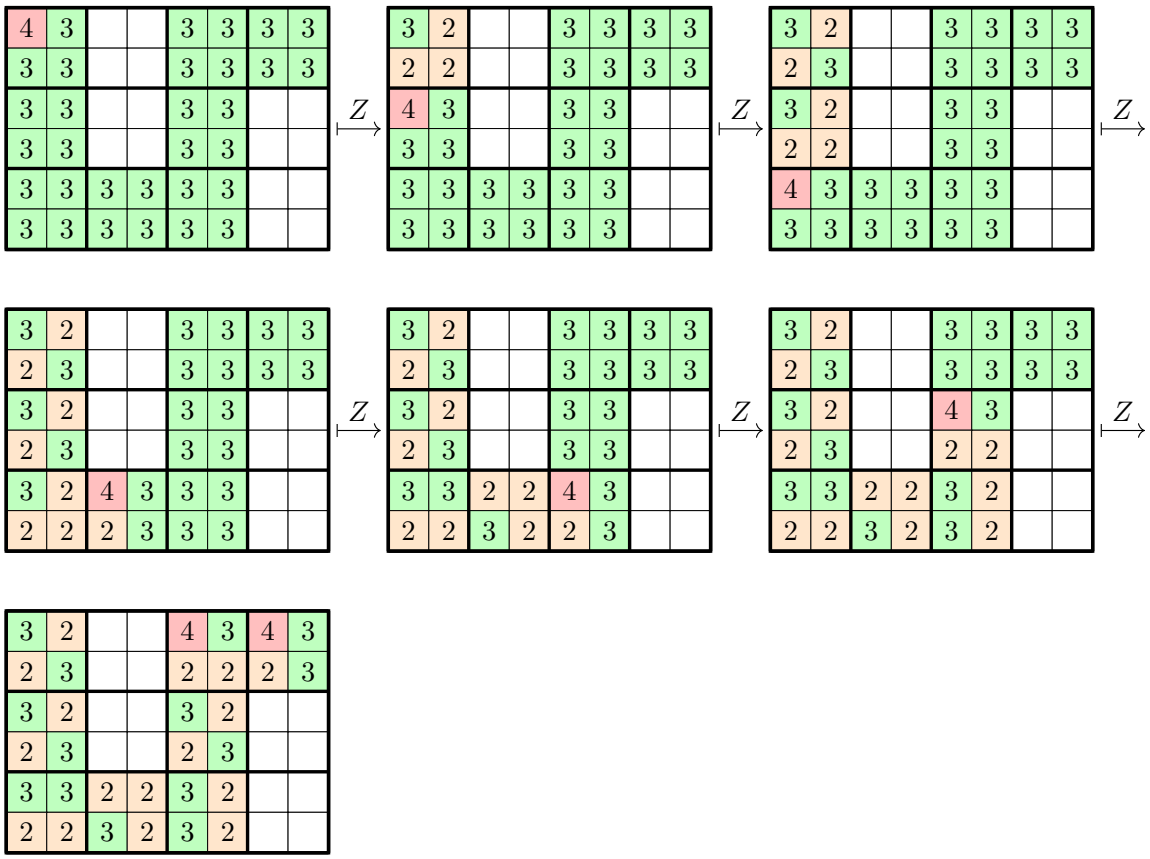
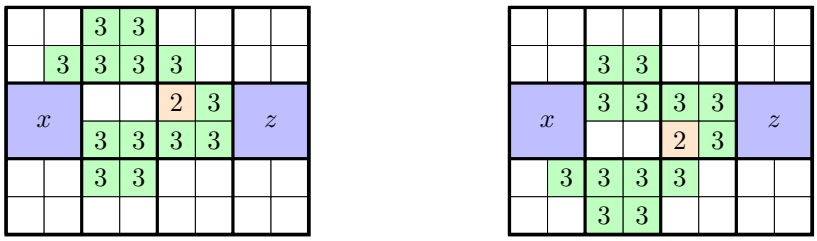


Figure 6.5.: A ⊞ signal moving along with two left turns (continuation of Figure 6.4)

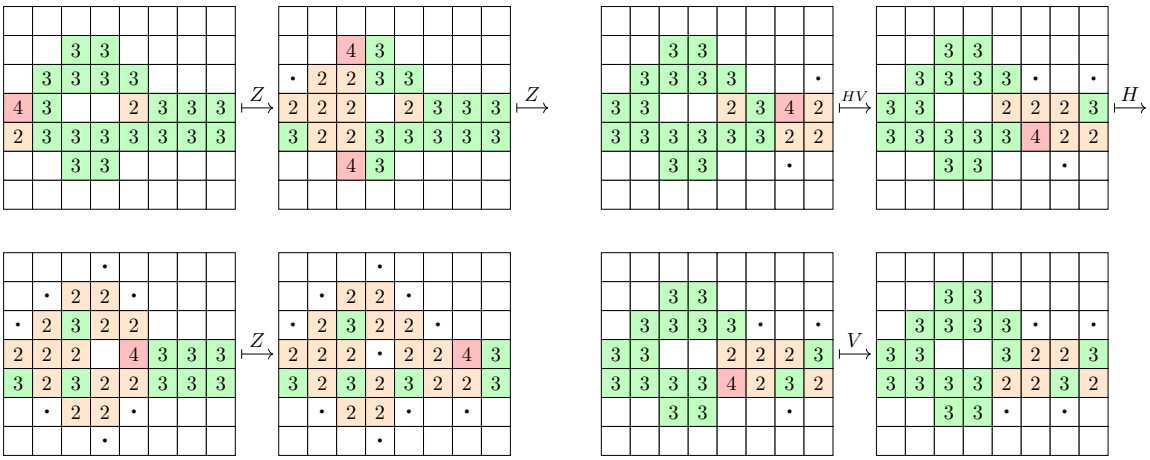


(a) ⊞ wires **(b)** ⊟ wires

Figure 6.6.: Diode implementations

(Recall that x denotes the component's input and z its output.) Figure 6.7 illustrates the operation of a diode for ⊞ signals. (The case of ⊟ signals is similar.)

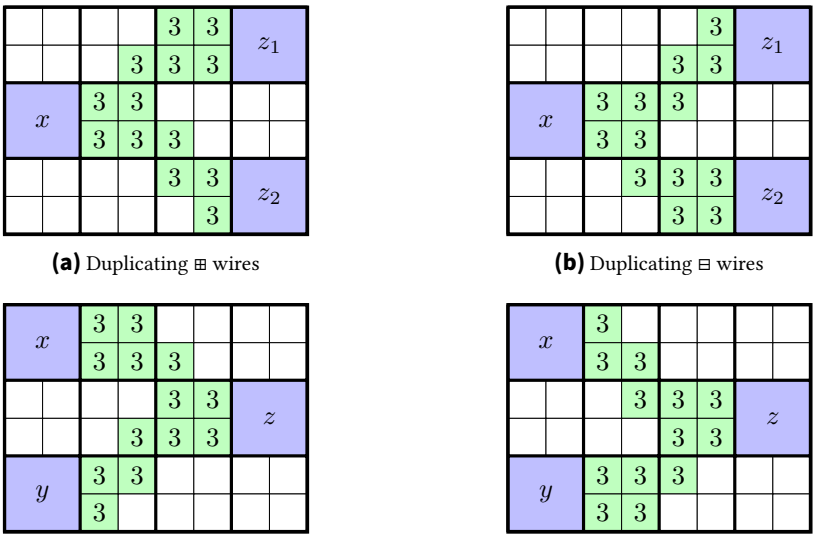
For all the remaining elements described in this section, we implicitly add diodes to their inputs and outputs. This ensures that the signals can only flow from left to right (as intended). This is probably not necessary for all elements, but doing so makes the construction simpler while the overhead is only a constant factor blowup in the size of the elements.



(a) ⊞ signal comes from the left

(b) ⊞ signal comes from the right

Figure 6.7.: Diode operation on ⊞ wires



(a) Duplicating ⊞ wires

(b) Duplicating ⊟ wires

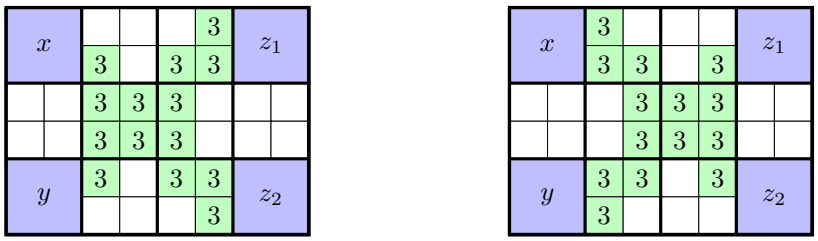
(c) Merging ⊞ wires

(d) Merging ⊟ wires

Figure 6.8.: Duplicating and merging wires

6.4.3. Duplicating, Merging, and Crossing Wires

Wires of the same polarity can be *duplicated* or *merged*. By duplicating a wire we mean we create two wires z_1 and z_2 from a single wire x in such a way that, if any signal arrives from x, then this signal is duplicated and propagated on both z_1 and z_2. (Equivalently, one might imagine that $x = z_1$ and z_2 is a wire copy of x.) In turn, a wire merge realizes in some sense the reverse operation: We have two wires x and y *of the same polarity* and create a wire z such that, if a signal arrives from x or y (or both), then a signal of the same polarity will emerge at z. (Hence one could say the wire merge realizes a polarized OR gate.) See Figure 6.8 for the implementations.



(a) ⊞ semicrossing **(b)** ⊟ semicrossing

Figure 6.9.: Semicrossing implementations

As discussed in the introduction, there is no straightforward realization of a wire crossing in fungal automata in the traditional sense. Nevertheless, it turns out we *can* cross wires under the following constraints:

1. The two wires being crossed are a ⊞ and a ⊟ wire.
2. The crossing is used only once and by a single input wire; that is, once a signal from either wire passes through the crossing, it is destroyed. (If two signals arrive from both wires at the same time, then the crossing is destroyed without allowing any signal to pass through.)

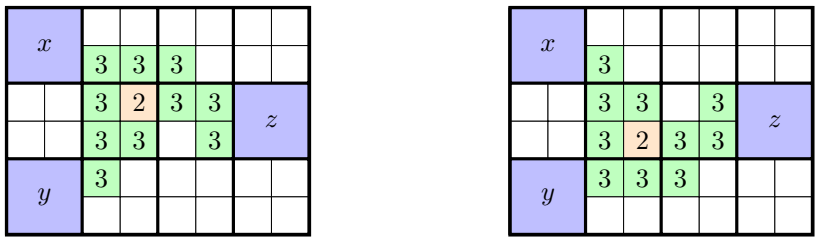
To elicit these limitations, we refer to such crossings as *semicrossings*.

We actually need two types of semicrossings, one for each choice of polarities for the two input wires. The semicrossings are named according to the polarity of the top input wire: A ⊞ semicrossing has a ⊞ wire as its top input (and a ⊟ wire as its bottom one) whereas a ⊟ semicrossing has a ⊟ wire at the top (and a ⊞ wire at the bottom). For the implementations, see Figure 6.9.

6.4.4. Switches

A *switch* is a rudimentary form of transistor. It has two inputs and one output. Adopting the terminology of field-effect transistors (FETs), we will refer to the two inputs as the *source* and *gate* and the output as the *drain*. In its initial state, the switch is *open* and does not allow source signals to pass through. If a signal arrives from the gate, then it turns the switch *closed*. A subsequent signal arriving from the source will then be propagated on to the drain. This means that switches are *delay-sensitive*: A signal arriving at the source only continues on to the drain if the gate signal has arrived beforehand (or simultaneously to the source).

Similar to semicrossings, our switches come in two flavors. In both cases the top input is a ⊞ wire and the bottom one a ⊟. The difference is that, in a ⊞ switch, the source (and thus also the drain) is the ⊞ input and the gate is the ⊟ input. Conversely, in a ⊟ switch the source and drain are ⊟ wires and the gate is a ⊞ wire. Refer to Figure 6.10 for the implementation of the two types of switches.



(a) ⊞ switch **(b)** ⊟ switch

Figure 6.10.: Switch implementations

6.4.5. Delays and Retarders

As mentioned in the introduction, the circuits we construct are sensitive to the time it takes for a signal to flow from one point to the other. To render this notion precise, we define for every component a *delay* that results from the time taken for a signal to pass through the component. This is defined as follows:

- The delay of a source is zero.
- The delay of a wire (including bends) at some block B is the delay of the wire's source S plus the length (in blocks) of the shortest contiguous path along the wire that leads from S to B according to the von Neumann neighborhood. We will refer to this length as the *wire distance* between S and B. For example, the wire distance between the inputs and outputs in all of Figures 6.6 and 6.8 to 6.10 is 4; similarly, the distance between x and z in Figure 6.11 (see below) is 15.
- The delay of a gate (i.e., a diode, wire duplication, wire merge, or semicrossing) is the maximum over the delays of its inputs plus the gate width (in blocks).

Notice our definition of wire distance may grossly estimate the actual number of steps a signal requires to propagate from S to B. This is fine for our purposes since we only need to reason about upper bounds later in Section 6.6.3.

Finally we will also need a *retarder* element, which is responsible for adding a variable amount of delay to a wire. Refer to Figure 6.11 for their realization. Retarders can have different dimensions. Evidently, one can ensure a delay of t with a retarder that is $O(\sqrt{t}) \times O(\sqrt{t})$ large. We are going to use retarders of delay at most D, where D depends on the CVP instance and is set later in Section 6.6.3. Hence, it is safe to assume all retarders in the same configuration are of the same size horizontally and vertically, but realize different delays. This allows one to use retarders of a *single* size for any fixed circuit, which simplifies the layout significantly (see also Sections 6.6.3 and 6.6.4).

6.5. Layer 3: Working With Bits

We will now use the elements from Section 6.4 (represented as in Figure 6.12) to construct planar delay-sensitive Boolean circuits. Our circuits will use NAND gates as their basis. We

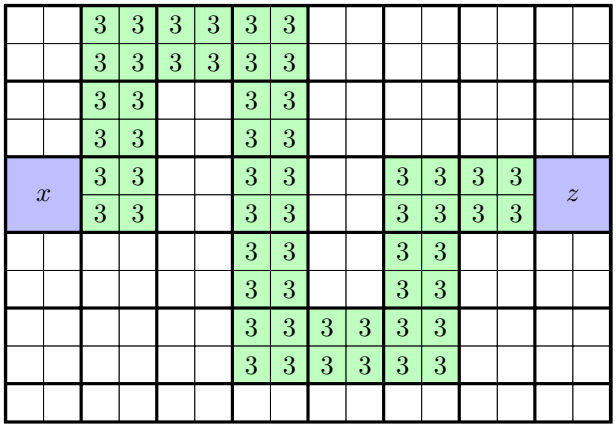


Figure 6.11.: Implementation of a basic retarder (for both ⊞ and ⊟ signals) that ensures a delay of ≥ 12 at z (relative to x). Retarders for greater delays can be realized by increasing (i) the height of the meanders, (ii) the number of up-down meanders, and (iii) the positions of the input and output.

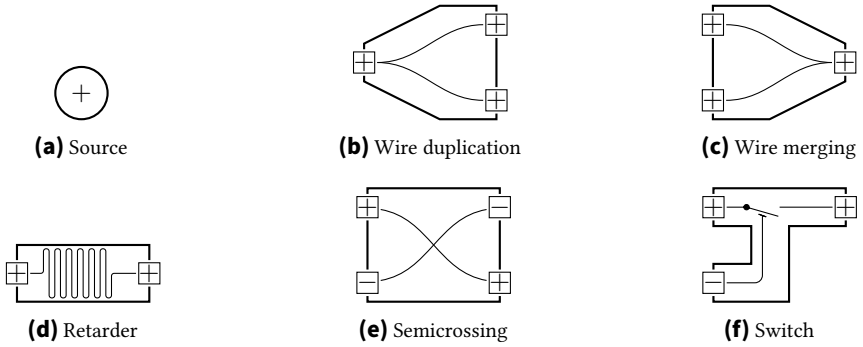


Figure 6.12.: Representations of the elements from abstraction layer 2 as used in layer 3. The polarities indicate whether the ⊞ or ⊟ version of the component is used.

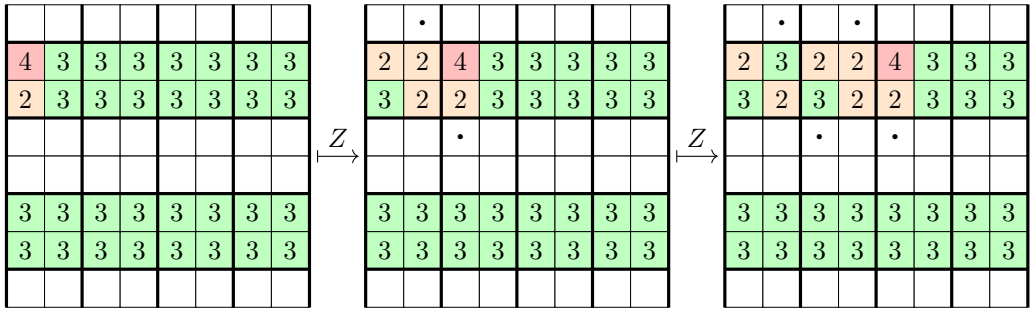
discuss how to overcome the planarity restriction in Section 6.5.4.

6.5.1. Representation of Bits

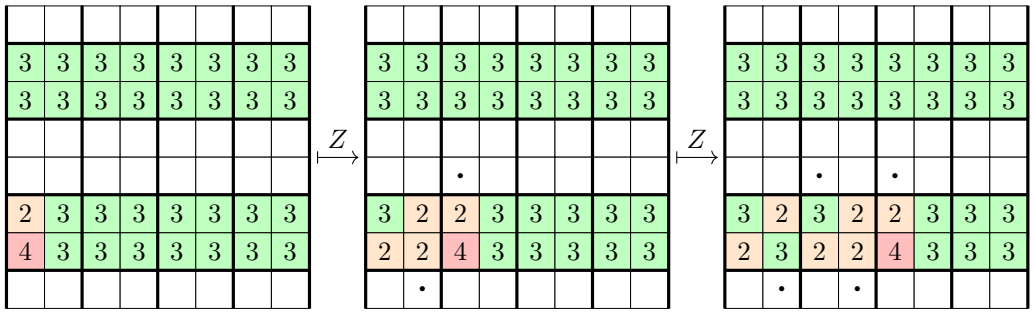
For the representation of a bit, we use a pair consisting of a polarized ⊞ wire and a polarized ⊟ wire. Such a pair of polarized wires is called a *cable*. As mentioned earlier, most of the time signals will travel from left to right. It is straightforward to generalize the notion of wire distance (see Section 6.4.5) to cables simply by setting it to the maximum of the respective wire distances.

A signal on a cable's ⊞ wire represents a binary `1`, and a signal on the ⊟ wire represents a binary `0`. By convention we will always draw the ⊞ wire "above" the ⊟ wire of the same cable. See Figure 6.13 for an example.

When referring to a gate's inputs and outputs, we indicate the ⊞ and ⊟ components of a cable with subscripts. For instance, for an input cable x, we write x_+ for its ⊞ and x_- for its ⊟ component.



(a) Binary `1`



(b) Binary `0`

Figure 6.13.: Binary representations traveling from left to right along a cable

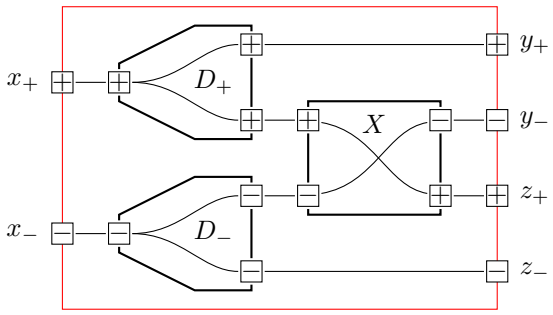


Figure 6.14.: Boolean branch

6.5.2. Bit Duplication

To duplicate a cable, we use the *Boolean branch* depicted in Figure 6.14. The circuit consists of two wire duplications (one of each polarity) and a crossing.

6.5.3. NAND Gates

As a matter of fact the NAND gate is inspired by the implementation of such a gate in CMOS technology[3]. Refer to Figure 6.15 for the implementation. Notice the usage of switches means these gates are *delay-sensitive*; that is, the gate only operates correctly (i.e.,

[3] e.g., `https://en.wikipedia.org/wiki/NAND_gate#/media/File:CMOS_NAND.svg`

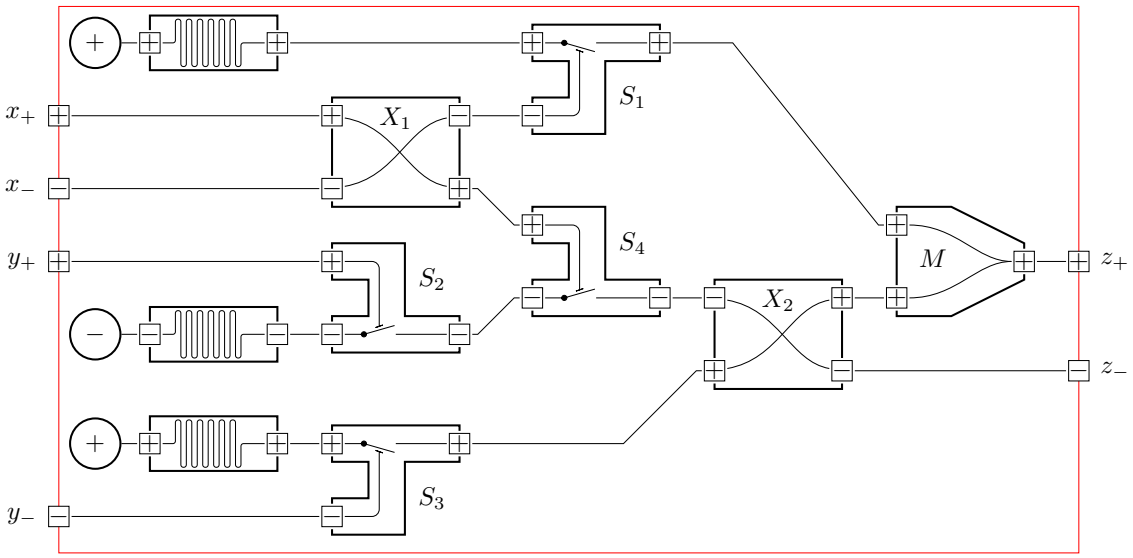


Figure 6.15.: NAND gate

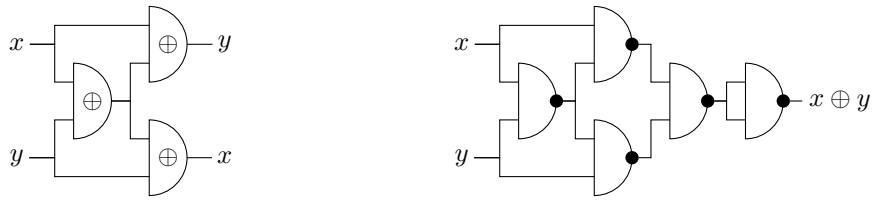
computing the NAND function) if the retarders have *strictly greater delays* than the inputs x and y. In fact, for our construction we will need to instantiate this same construction using *varying* values for the retarders' delays (but not their size as mentioned in Section 6.4.5). This seems necessary in order to chain NAND gates in succession (since each gate in a chain incurs a certain delay that must be compensated for in the next gate down the chain).

In addition, notice that in principle NAND gates have *variable* size as their dimension depends on that of the three retarders, As is the case for retarders, in the same embedding we insist on having all NAND gates be of the same size. We defer setting their dimensions to Section 6.6.3; for now, it suffices to keep in mind that NAND gates (and retarder elements) in the same embedding only vary in their delay (and not their size).

Claim. *Assuming the retarders have larger delay than the input cables x and y, the circuit on Figure 6.15 realizes a* NAND *gate.*

Proof. Consider first the case where both x_+ and y_+ are set. Since x_- is not set, X_1 is consumed by x_+, turning S_4 on. In addition, since y_+ is set, S_2 is also turned on. Hence, using the assumption on the delay of the inputs, the negative source flows through S_2, S_4, and X_2 on to z_-. Since both the switches S_1 and S_3 remain open, the z_+ output is never set. Notice the crossings X_1 and X_2 are each used exactly once.

Let now x_- or y_- (or both) be set. Then either S_2 or S_4 is open, which means z_- is never set. As a result, X_2 is used at most once (namely in case y_- is set). If x_- is set, then S_1 is opened, thus allowing the positive source to flow on to M. The same holds if y_- is set, in which case M receives the positive source arriving from S_3. Hence, at least one positive signal will flow to the M gate, causing z_+ to be set eventually. □



(a) Crossing two cables using XOR gates **(b)** Implementing XOR with NAND gates

Figure 6.16.: Implementing cable crossings as in [52]

6.5.4. Cable Crossings

There is a more or less well-known idea to cross to bits using three XOR gates that can be found in the paper by Goldschlager [52], for example. Figure 6.16 shows the idea. This construction can be used in FA. Because of the delays, there is not *the* crossing gate, but a whole family of them. Depending on the position in the whole circuit layout, each crossing needs NAND gates with specific builtin delays (that will be set in Section 6.6.3).

6.6. Layer 4: Layout of a Whole Circuit

Finally we describe one possibility to construct a finite rectangle of cells F of a FA containing the realization of a complete circuit, given its description B. The important point here is that, in order to produce F from B, the constructor only needs logarithmic space. (Therefore the simplicity of the layout has precedence over any form of "optimization".)

6.6.1. Arranging the Circuit in Tiles

Let C be the circuit that is to be embedded as an FA configuration F. Letting n be the length of inputs to C and m its number of gates, notice we have an upper bound of m on the circuit depth of C. Without restriction, we may assume $m \geq n$, which also implies an upper bound of $m + n = O(m)$ on the number of cables of C (since C has bounded fan-in). The logical gates of C are denoted by $G_1, \ldots, G_m$ and we assume that G_i has number $n + i$ in description B of C (recall Section 6.1.1).

In the configuration F we have cables $x_1, \ldots, x_n$ originating from the input gates as well as cables $g_1, \ldots, g_m$ coming from (the embedding of) the gates of C. The x_i and g_i flow in and out of equal-sized *tiles* $T_1, \ldots, T_m$, where in the i-th tile T_i we implement the i-th gate G_i of C. The inputs to T_i are $I_i = \{x_1, \ldots, x_n, g_1, \ldots, g_{i-1}\}$ and its outputs $O_i = I_i \cup \{g_i\}$; hence $I_{i+1} = O_i$.

Recall that, unlike standard circuits, the behavior of our layer 3 circuits is subject to spatial considerations, that is, to both gate placement and wire length. For the sake of simplicity, each tile is shaped as a square and all tiles are of the same size. In addition, the tiles are placed in ascending order from left to right and with no space in-between. The only objects

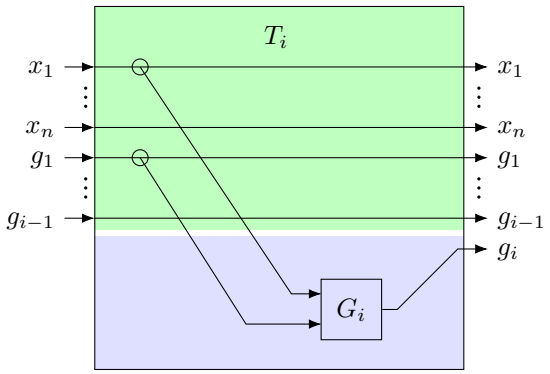


Figure 6.17.: Overview of the tile T_i. The *upper part* of the tile has green background, the *lower part* has blue background.

in F that lie outside the tiles are the inputs and output of C itself. The inputs are placed immediately next to corresponding cables that go into T_1 whereas the output is placed next to its corresponding wire g_m at the outgoing end of T_m.

6.6.2. Layout for Tile i

As depicted in Figure 6.17, each tile is subdivided into two areas. The *upper part* contains the wires that pass through it, while the *lower part* implements the gate G_i proper.

We give a broad overview of the process for constructing T_i. First determine the numbers y_1 and y_2 of the inputs to G_i. Then duplicate the bits on cables y_1 and y_2 (as in Section 6.5.2) and cross the copies over to the lower part of the tile. These crossings require setting adequate delays, which will be addressed in the next section. (In case $y_1 = y_2$, duplicate the cable twice and proceed as otherwise described.) Next instantiate G_i with a proper amount of delay (again, see the next section) and plug in y_1 and y_2 as inputs into G_i. Finally connect all inputs in I_i as well as the output wire g_i of G_i to their respective outputs. Notice the tile contains $O(m)$ crossings and thus also $O(m)$ NAND gates in total.

6.6.3. Choosing Suitable Delays for All Gates

The two details that remain are setting the dimensions and the delays for the retarders in all NAND gates. This requires certain care since we may otherwise end up running into a chicken-and-egg problem: The retarders' dimensions are determined by the required delays (in order to have enough space to realize them); in turn, the delays depend on the aforementioned dimensions (since the input wires in the NAND gates must be laid so as to "go around" the retarders).

The solution is to assume we already have an upper bound D on the maximum delay in F. This allows us to fix the size of the components as follows:

- The retarders and NAND gates have side length $O(\sqrt{D})$.

- Each tile has side length $O(m\sqrt{D})$.
- The support of F fits into a square with side length $O(m^2\sqrt{D})$.

With this in place, we determine upper bounds on the delays of the upper gates in a tile (i.e., the gates in the upper part of the tile), then of the lower gates G_i, then of the tiles themselves, and finally of the entire embedding of C. In the end we obtain an upper bound for the maximum possible delay in F. Simply setting D to be at least as large concludes the construction.

Upper gates. In order to set the delays of a NAND gate G in a tile T_i, we first need an upper bound d_{input} on the delays of the two inputs to G. Suppose the origins O_1 and O_2 of these inputs (i.e., either a NAND gate output or an input to T_i) have delay at most d_{origin}. Then certainly we have

$$d_{\text{input}} \leq d_{\text{origin}} + d_{\text{cable}},$$

where d_{cable} is the maximum of the cable distances between either one of O_1 and O_2 and the switches they are connected to inside G. Due to the layout of a tile and since a NAND gate has $O(\sqrt{D})$ side length, we know d_{cable} is at most $O(\sqrt{D})$. Hence, if G is in the j-th layer of T_i, then we may safely upper-bound its delay by $d_i + (j+1)d_{\text{cable}}$, where d_i is the maximum over the delays of the inputs to T_i.

Lower gates. Since there are $O(m)$ cables inside a tile, there are $O(m)$ cable crossings and thus $O(m)$ NAND gates realizing these crossings. Hence the inputs to the gate G_i in the lower part of T_i have delay at most

$$d_i + O(m) \cdot d_{\text{cable}} + O(m\sqrt{D}),$$

where the last factor is due to the side length of T (i.e., the maximum cable length needed to connect the last of the upper gates with G_i).

Tiles. Clearly the greatest delay among the output cables of T_i is that of g_i (since every other cable originates from a straight path across T_i). As we have determined in the last paragraph, at its output g_i has delay

$$d_{i+1} \leq d_i + O(m\sqrt{D}).$$

Since the side length of a tile is $O(m\sqrt{D})$, we may upper-bound the delays of the inputs of T_i by $i \cdot O(m\sqrt{D})$.

Support of F. Since there are m tiles in total, it suffices to choose a maximum delay D that satisfies $D \geq cm^2\sqrt{D}$ for some adequate constant c (that results from the considerations above). In particular, this means we may set $D = \Theta(m^4)$ independently of C.

6.6.4. Constructor

In this final section we describe how to realize a logspace constructor R that, given a CVP instance consisting of the description of a circuit C and an input x to it, reduces it to an instance as in Theorem 6.1. Due to the structure of F, this is relatively straightforward.

The constructor R outputs the description of F column for column. (Computing the coordinates of an element or wire is clearly feasible in logspace.) In the first few columns R sets the inputs to the embedded circuit according to x. Next R constructs F tile for tile. To construct tile T_i, R determines which cables are the inputs to G_i and constructs crossings accordingly. To estimate the delays of each wire, R uses the upper bounds we have determined in Section 6.6.3, which clearly are all computable in logspace (since the maximum delay D is polynomial in m).

Finally R also needs to produce y and T as in the statement of Theorem 6.1. Let c_i be the cable of T_m that corresponds to the output of the embedded circuit C. Then we let y be the index of the cell next to the $\boxplus$ wire of c_i at the output of T_m. (Hence y assumes a non-zero state if and only if c_i contains a 1, that is, $C(x) = 1$.) As for T, certainly setting it to the number of cells in F suffices (since a signal needs to visit every cell in F at most once).

Bibliography

[1] Andrew Adamatzky, Eric Goles Ch., Michail-Antisthenis I. Tsompanas, Genaro J. Martínez, Han A. B. Wosten, and Martin Tegelaar. "On Fungal Automata." In: *Automata and Complexity - Essays Presented to Eric Goles on the Occasion of His 70th Birthday*. Ed. by Andrew Adamatzky. Vol. 42. Springer, 2022, pp. 455–483.

[2] Eric Allender. "The New Complexity Landscape Around Circuit Minimization." In: *Language and Automata Theory and Applications - 14th International Conference, LATA 2020, Milan, Italy, March 4-6, 2020, Proceedings*. Ed. by Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron. Vol. 12038. Lecture Notes in Computer Science. Springer, 2020, pp. 3–16.

[3] Eric Allender and Michal Koucký. "Amplifying lower bounds by means of self-reducibility." In: *J. ACM* 57.3 (2010), 14:1–14:36.

[4] Roy Armoni. "On the Derandomization of Space-Bounded Computations." In: *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*. Ed. by Michael Luby, José D. P. Rolim, and Maria J. Serna. Vol. 1518. Lecture Notes in Computer Science. Springer, 1998, pp. 47–59.

[5] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge: Cambridge University Press, 2009.

[6] Pablo Arrighi, Renan Fargetton, Vincent Nesme, and Eric Thierry. "Applying Causality Principles to the Axiomatization of Probabilistic Cellular Automata." In: *Models of Computation in Context - 7th Conference on Computability in Europe, CiE 2011, Sofia, Bulgaria, June 27 - July 2, 2011. Proceedings*. Ed. by Benedikt Löwe, Dag Normann, Ivan N. Soskov, and Alexandra A. Soskova. Vol. 6735. Lecture Notes in Computer Science. Springer, 2011, pp. 1–10.

[7] Pablo Arrighi, Nicolas Schabanel, and Guillaume Theyssier. "Stochastic Cellular Automata: Correlations, Decidability and Simulations." In: *Fundam. Informaticae* 126.2-3 (2013), pp. 121–156.

[8] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. "Models and Issues in Data Stream Systems." In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. Ed. by Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis. ACM, 2002, pp. 1–16.

[9] Nicolas Bacquey, Etienne Grandjean, and Frédéric Olive. "Definability by Horn Formulas and Linear Time on Cellular Automata." In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*. Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 99:1–99:14.

[10] Per Bak, Chao Tang, and Kurt Wiesenfeld. "Self-organized criticality: An explanation of the $1/f$ noise." In: *Phys. Rev. Lett.* 59.4 (1987), pp. 381–384.

[11] Theodore P. Baker, John Gill, and Robert Solovay. "Relativizations of the P =? NP Question." In: *SIAM J. Comput.* 4.4 (1975), pp. 431–442.

[12] David A. Mix Barrington. "Extensions of an Idea of McNaughton." In: *Mathematical Systems Theory* 23.3 (1990), pp. 147–164.

[13] Danièle Beauquier and Jean-Eric Pin. "Factors of Words." In: *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*. 1989, pp. 63–79.

[14] Danièle Beauquier and Jean-Eric Pin. "Languages and Scanners." In: *Theor. Comput. Sci.* 84.1 (1991), pp. 3–21.

[15] Andrej Bogdanov, William M. Hoza, Gautam Prakriya, and Edward Pyne. "Hitting Sets for Regular Branching Programs." In: *37th Computational Complexity Conference, CCC 2022, July 20-23, 2022, Philadelphia, PA, USA*. Ed. by Shachar Lovett. Vol. 234. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 3:1–3:22.

[16] Vladimir Braverman. "Sliding Window Algorithms." In: *Encyclopedia of Algorithms*. 2016, pp. 2006–2011.

[17] Janusz A. Brzozowski and Imre Simon. "Characterizations of locally testable events." In: *Discrete Mathematics* 4.3 (1973), pp. 243–271.

[18] Hervé Caussinus, Pierre McKenzie, Denis Thérien, and Heribert Vollmer. "Nondeterministic NC^1 Computation." In: *J. Comput. Syst. Sci.* 57.2 (1998), pp. 200–212.

[19] Bernard Chazelle and Louis Monier. "A Model of Computation for VLSI with Related Complexity Results." In: *J. ACM* 32.3 (1985), pp. 573–588.

[20] Lijie Chen, Shuichi Hirahara, Igor Carboni Oliveira, Ján Pich, Ninad Rajgopal, and Rahul Santhanam. "Beyond Natural Proofs: Hardness Magnification and Locality." In: *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*. Ed. by Thomas Vidick. Vol. 151. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 70:1–70:48.

[21] Lijie Chen, Ce Jin, and R. Ryan Williams. "Hardness Magnification for all Sparse NP Languages." In: *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*. Ed. by David Zuckerman. IEEE Computer Society, 2019, pp. 1240–1255.

[22] Lijie Chen, Ce Jin, and R. Ryan Williams. "Sharp threshold results for computational complexity." In: *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*. Ed. by Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy. ACM, 2020, pp. 1335–1348.

[23] Lijie Chen, Dylan M. McKay, Cody D. Murray, and R. Ryan Williams. "Relations and Equivalences Between Circuit Lower Bounds and Karp-Lipton Theorems." In: *34th Computational Complexity Conference, CCC 2019, July 18-20, 2019, New Brunswick, NJ, USA*. Ed. by Amir Shpilka. Vol. 137. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 30:1–30:21.

[24] Lijie Chen and Roei Tell. "Bootstrapping results for threshold circuits "just beyond" known lower bounds." In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. Ed. by Moses Charikar and Edith Cohen. ACM, 2019, pp. 34–41.

[25] Mahdi Cheraghchi, Shuichi Hirahara, Dimitrios Myrisiotis, and Yuichi Yoshida. "One-Tape Turing Machine and Branching Program Lower Bounds for MCSP." In: *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*. Ed. by Markus Bläser and Benjamin Monmege. Vol. 187. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 23:1–23:19.

[26] Stephen A. Cook. "A Taxonomy of Problems with Fast Parallel Algorithms." In: *Information and Control* 64.1-3 (1985), pp. 2–21.

[27] Graham Cormode. "The continuous distributed monitoring model." In: *SIGMOD Rec.* 42.1 (2013), pp. 5–14.

[28] Stefano Crespi-Reghizzi, Dora Giammarresi, and Violetta Lonati. "Two-dimensional models." In: *Handbook of Automata Theory*. Ed. by Jean-Éric Pin. European Mathematical Society Publishing House, Zürich, Switzerland, 2021, pp. 303–333.

[29] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. "Maintaining Stream Statistics over Sliding Windows." In: *SIAM J. Comput.* 31.6 (2002), pp. 1794–1813.

[30] Marianne Delorme and Jacques Mazoyer, eds. *Cellular Automata. A Parallel Model*. Mathematics and Its Applications 460. Dordrecht: Springer Netherlands, 1999.

[31] Christoph Dürr, Ivan Rapaport, and Guillaume Theyssier. "Cellular automata and communication complexity." In: *Theor. Comput. Sci.* 322.2 (2004), pp. 355–368.

[32] Patrick W. Dymond and Stephen A. Cook. "Hardware Complexity and Parallel Computation (Preliminary Version)." In: *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*. IEEE Computer Society, 1980, pp. 360–372.

[33] Samuel Eilenberg. *Automata, Languages, and Machines*. Vol. B. Pure and applied mathematics. New York: Academic Press, 1976.

[34] Terry Farrelly. "A review of Quantum Cellular Automata." In: *Quantum* 4 (2020), p. 368.

[35] Laurent Feuilloley and Pierre Fraigniaud. "Survey of Distributed Decision." In: *Bull. EATCS* 119 (2016).

[36] Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. "Compact Distributed Certification of Planar Graphs." In: *Algorithmica* 83.7 (2021), pp. 2215–2244.

[37] Eldar Fischer. "The Art of Uninformed Decisions." In: *Bulletin of the EATCS* 75 (2001), p. 97.

[38] Michael A. Forbes and Zander Kelley. "Pseudorandom Generators for Read-Once Branching Programs, in Any Order." In: *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*. Ed. by Mikkel Thorup. IEEE Computer Society, 2018, pp. 946–955.

[39] Enrico Formenti and Kévin Perrot. "How Hard is it to Predict Sandpiles on Lattices? A Survey." In: *Fundam. Informaticae* 171.1-4 (2020), pp. 189–219.

[40] Pierre Fraigniaud, Pedro Montealegre, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. "On Distributed Merlin-Arthur Decision Protocols." In: *Structural Information and Communication Complexity - 26th International Colloquium, SIROCCO 2019, L'Aquila, Italy, July 1-4, 2019, Proceedings*. Ed. by Keren Censor-Hillel and Michele Flammini. Vol. 11639. Lecture Notes in Computer Science. Springer, 2019, pp. 230–245.

[41] Bin Fu. "Hardness of Sparse Sets and Minimal Circuit Size Problem." In: *Computing and Combinatorics - 26th International Conference, COCOON 2020, Atlanta, GA, USA, August 29-31, 2020, Proceedings*. Ed. by Donghyun Kim, R. N. Uma, Zhipeng Cai, and Dong Hoon Lee. Vol. 12273. Lecture Notes in Computer Science. Springer, 2020, pp. 484–495.

[42] Merrick L. Furst, James B. Saxe, and Michael Sipser. "Parity, Circuits, and the Polynomial-Time Hierarchy." In: *Mathematical Systems Theory* 17.1 (1984), pp. 13–27.

[43] Anahí Gajardo and Eric Goles. "Crossing information in two-dimensional Sandpiles." In: *Theor. Comput. Sci.* 369.1-3 (2006), pp. 463–469.

[44] Moses Ganardi, Danny Hucke, and Markus Lohrey. "Randomized Sliding Window Algorithms for Regular Languages." In: *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. Ed. by Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella. Vol. 107. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 127:1–127:13.

[45] Moses Ganardi, Danny Hucke, and Markus Lohrey. "Sliding Window Algorithms for Regular Languages." In: *Language and Automata Theory and Applications - 12th International Conference, LATA 2018, Ramat Gan, Israel, April 9-11, 2018, Proceedings*. Ed. by Shmuel Tomi Klein, Carlos Martín-Vide, and Dana Shapira. Vol. 10792. Lecture Notes in Computer Science. Springer, 2018, pp. 26–35.

[46] Moses Ganardi, Danny Hucke, and Markus Lohrey. "Derandomization for Sliding Window Algorithms with Strict Correctness*." In: *Theory Comput. Syst.* 65.3 (2021), pp. 1–18.

[47] Moses Ganardi, Artur Jez, and Markus Lohrey. "Sliding Windows over Context-Free Languages." In: *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*. Ed. by Igor Potapov, Paul G. Spirakis, and James Worrell. Vol. 117. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 15:1–15:15.

[48] Pedro García and José Ruiz. "Threshold Locally Testable Languages in Strict Sense." In: *Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology, and Back: Essays in Honour of Gheorghe Paun*. Ed. by Carlos Martín-Vide and Victor Mitrana. Vol. 9. Topics in Computer Mathematics. Taylor and Francis, 2003, pp. 243–252.

[49] Aurélien Garivier and Eric Moulines. "On Upper-Confidence Bound Policies for Switching Bandit Problems." In: *Algorithmic Learning Theory - 22nd International Conference, ALT 2011, Espoo, Finland, October 5-7, 2011. Proceedings*. Ed. by Jyrki Kivinen, Csaba Szepesvári, Esko Ukkonen, and Thomas Zeugmann. Vol. 6925. Lecture Notes in Computer Science. Springer, 2011, pp. 174–188.

[50] Oded Goldreich. *Computational Complexity: A Conceptional Perspective*. Cambridge: Cambridge University Press, 2008.

[51] Oded Goldreich and Avi Wigderson. "Tiny families of functions with random properties: A quality-size trade-off for hashing." In: *Random Struct. Algorithms* 11.4 (1997), pp. 315–343.

[52] Leslie M. Goldschlager. "The monotone and planar circuit value problems are log space complete for P." In: *SIGACT News* 9.2 (1977), pp. 25–29.

[53] Eric Goles, Michail-Antisthenis I. Tsompanas, Andrew Adamatzky, Martin Tegelaar, Han A. B. Wosten, and Genaro J. Martínez. "Computational universality of fungal sandpile automata." In: *Phys. Lett. A* 384.22 (2020), 126541:1–126541:8.

[54] Anaël Grandjean and Victor Poupet. "L-Convex Polyominoes Are Recognizable in Real Time by 2D Cellular Automata." In: *Cellular Automata and Discrete Complex Systems - 21st IFIP WG 1.5 International Workshop, AUTOMATA 2015, Turku, Finland, June 8-10, 2015. Proceedings*. Ed. by Jarkko Kari. Vol. 9099. Lecture Notes in Computer Science. Springer, 2015, pp. 127–140.

[55] Anaël Grandjean and Victor Poupet. "A Linear Acceleration Theorem for 2D Cellular Automata on All Complete Neighborhoods." In: *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*. Ed. by Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi. Vol. 55. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 115:1–115:12.

[56] Etienne Grandjean and Frédéric Olive. "A logical approach to locality in pictures languages." In: *J. Comput. Syst. Sci.* 82.6 (2016), pp. 959–1006.

[57] Juho Hirvonen and Jukka Suomela. *Distributed Algorithms 2020*. Available online. URL: `https://jukkasuomela.fi/da2020`.

[58] William M. Hoza, Edward Pyne, and Salil P. Vadhan. "Pseudorandom Generators for Unbounded-Width Permutation Branching Programs." In: *12th Innovations in Theoretical Computer Science Conference, ITCS 2021, January 6-8, 2021, Virtual Conference*. Ed. by James R. Lee. Vol. 185. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 7:1–7:20.

[59] William M. Hoza and David Zuckerman. "Simple Optimal Hitting Sets for Small-Success RL." In: *SIAM J. Comput.* 49.4 (2020), pp. 811–820.

[60] Oscar H. Ibarra, Michael A. Palis, and Sam M. Kim. "Fast Parallel Language Recognition by Cellular Automata." In: *Theor. Comput. Sci.* 41 (1985), pp. 231–246.

[61] Rahul Ilango, Bruno Loff, and Igor Carboni Oliveira. "NP-Hardness of Circuit Minimization for Multi-Output Functions." In: *35th Computational Complexity Conference, CCC 2020, July 28-31, 2020, Saarbrücken, Germany (Virtual Conference)*. Ed. by Shubhangi Saraf. Vol. 169. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 22:1–22:36.

[62] Neil Immerman. *Descriptive Complexity*. Graduate Texts in Computer Science. New York: Springer, 1999.

[63] Chuzo Iwamoto, Tomonobu Hatsuyama, Kenichi Morita, and Katsunobu Imai. "Constructible functions in cellular automata and their applications to hierarchy results." In: *Theor. Comput. Sci.* 270.1-2 (2002), pp. 797–809.

[64] Valentine Kabanets and Jin-yi Cai. "Circuit minimization problem." In: *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*. Ed. by F. Frances Yao and Eugene M. Luks. ACM, 2000, pp. 73–79.

[65] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. "Revisiting Norm Estimation in Data Streams." In: *CoRR* abs/0811.3648 (2008). arXiv: `0811.3648`.

[66] Sam Kim and Robert McCloskey. "A Characterization of Constant-Time Cellular Automata Computation." In: *Phys. D* 45.1-3 (1990), pp. 404–419.

[67] Kojiro Kobayashi. "On the structure of one-tape nondeterministic turing machine time hierarchy." In: *Theor. Comput. Sci.* 40 (1985), pp. 175–193.

[68] Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. "Interactive Distributed Proofs." In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*. Ed. by Calvin Newport and Idit Keidar. ACM, 2018, pp. 255–264.

[69] Martin Kutrib. "Cellular Automata and Language Theory." In: *Encyclopedia of Complexity and Systems Science*. 2009, pp. 800–823.

[70] Martin Kutrib. "Complexity of One-Way Cellular Automata." In: *Cellular Automata and Discrete Complex Systems - 20th International Workshop, AUTOMATA 2014, Himeji, Japan, July 7-9, 2014, Revised Selected Papers*. Ed. by Teijiro Isokawa, Katsunobu Imai, Nobuyuki Matsui, Ferdinand Peper, and Hiroshi Umeo. Vol. 8996. Lecture Notes in Computer Science. Springer, 2014, pp. 3–18.

[71] Martin Kutrib, Andreas Malcher, and Matthias Wendlandt. "Shrinking one-way cellular automata." In: *Nat. Comput.* 16.3 (2017), pp. 383–396.

[72] Nathan Linial. "Locality in Distributed Graph Algorithms." In: *SIAM J. Comput.* 21.1 (1992), pp. 193–201.

[73] Jean Mairesse and Irène Marcovici. "Around probabilistic cellular automata." In: *Theor. Comput. Sci.* 559 (2014), pp. 42–72.

[74] Andrew McGregor. "Graph stream algorithms: a survey." In: *SIGMOD Rec.* 43.1 (2014), pp. 9–20.

[75] Dylan M. McKay, Cody D. Murray, and R. Ryan Williams. "Weak lower bounds on resource-bounded compression imply strong separations of complexity classes." In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. Ed. by Moses Charikar and Edith Cohen. ACM, 2019, pp. 1215–1225.

[76] Robert McNaughton. "Algebraic Decision Procedures for Local Testability." In: *Mathematical Systems Theory* 8.1 (1974), pp. 60–76.

[77] Robert McNaughton and Seymour Papert. *Counter-Free Automata*. Cambridge, MA: The MIT Press, 1971.

[78] Peter Bro Miltersen. "The Computational Complexity of One-Dimensional Sandpiles." In: *Theory Comput. Syst.* 41.1 (2007), pp. 119–125.

[79] Augusto Modanese. "Shrinking and Expanding One-Dimensional Cellular Automata." Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2016.

[80] Augusto Modanese. "Complexity-Theoretic Aspects of Expanding Cellular Automata." In: *Cellular Automata and Discrete Complex Systems - 25th IFIP WG 1.5 International Workshop, AUTOMATA 2019, Guadalajara, Mexico, June 26-28, 2019, Proceedings*. Ed. by Alonso Castillo-Ramirez and Pedro Paulo Balbi de Oliveira. Vol. 11525. Lecture Notes in Computer Science. Springer, 2019, pp. 20–34.

[81] Augusto Modanese. "Sublinear-Time Language Recognition and Decision by One-Dimensional Cellular Automata." In: *Developments in Language Theory - 24th International Conference, DLT 2020, Tampa, FL, USA, May 11-15, 2020, Proceedings*. Ed. by Natasa Jonoska and Dmytro Savchuk. Vol. 12086. Lecture Notes in Computer Science. Springer, 2020, pp. 251–265.

[82] Augusto Modanese. "Lower Bounds and Hardness Magnification for Sublinear-Time Shrinking Cellular Automata." In: *Computer Science - Theory and Applications - 16th International Computer Science Symposium in Russia, CSR 2021, Sochi, Russia, June 28 - July 2, 2021, Proceedings*. Ed. by Rahul Santhanam and Daniil Musatov. Vol. 12730. Lecture Notes in Computer Science. Springer, 2021, pp. 296–320.

[83] Augusto Modanese. "Sublinear-Time Language Recognition and Decision by One-Dimensional Cellular Automata." In: *Int. J. Found. Comput. Sci.* 32.6 (2021), pp. 713–731.

[84] Augusto Modanese. "Sublinear-Time Probabilistic Cellular Automata." In: *CoRR* abs/2203.14614 (2022). arXiv: 2203.14614.

[85] Augusto Modanese and Thomas Worsch. "Shrinking and Expanding Cellular Automata." In: *Cellular Automata and Discrete Complex Systems - 22nd IFIP WG 1.5 International Workshop, AUTOMATA 2016, Zurich, Switzerland, June 15-17, 2016, Proceedings*. Ed. by Matthew Cook and Turlough Neary. Vol. 9664. Lecture Notes in Computer Science. Springer, 2016, pp. 159–169.

[86] Augusto Modanese and Thomas Worsch. "Embedding arbitrary Boolean circuits into fungal automata." In: *LATIN 2022: Theoretical Informatics - 15th Latin American Symposium, Guanajuato, Mexico, November 7-11, 2022, Proceedings*. Lecture Notes in Computer Science. Springer, 2022. Forthcoming.

[87] Pedro Montealegre, Diego Ramírez-Romero, and Ivan Rapaport. "Shared vs Private Randomness in Distributed Interactive Proofs." In: *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*. Ed. by Yixin Cao, Siu-Wing Cheng, and Minming Li. Vol. 181. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 51:1–51:13.

[88] Pedro Montealegre, Diego Ramírez-Romero, and Ivan Rapaport. "Compact Distributed Interactive Proofs for the Recognition of Cographs and Distance-Hereditary Graphs." In: *Stabilization, Safety, and Security of Distributed Systems - 23rd International Symposium, SSS 2021, Virtual Event, November 17-20, 2021, Proceedings*. Ed. by Colette Johnen, Elad Michael Schiller, and Stefan Schmid. Vol. 13046. Lecture Notes in Computer Science. Springer, 2021, pp. 395–409.

[89] Cristopher Moore and Martin Nilsson. "The computational complexity of sandpiles." In: *Journal of Statistical Physics* 96.1 (1999), pp. 205–224.

[90] Cody D. Murray and R. Ryan Williams. "On the (Non) NP-Hardness of Computing Circuit Complexity." In: *Theory of Computing* 13.1 (2017), pp. 1–22.

[91] Noam Nisan. "Pseudorandom generators for space-bounded computation." In: *Comb.* 12.4 (1992), pp. 449–461.

[92] Igor Carboni Oliveira, Ján Pich, and Rahul Santhanam. "Hardness Magnification Near State-of-the-Art Lower Bounds." In: *Theory Comput.* 17 (2021), pp. 1–38.

[93] Igor Carboni Oliveira and Rahul Santhanam. "Hardness Magnification for Natural Problems." In: *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*. Ed. by Mikkel Thorup. IEEE Computer Society, 2018, pp. 65–76.

[94] Maciej Pacut, Mahmoud Parham, Joel Rybicki, Stefan Schmid, Jukka Suomela, and Aleksandr Tereshchenko. "Brief Announcement: Temporal Locality in Online Algorithms." In: *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*. Ed. by Christian Scheideler. Vol. 246. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 52:1–52:3.

[95] Victor Poupet. "A Padding Technique on Cellular Automata to Transfer Inclusions of Complexity Classes." In: *Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings*. Ed. by Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov. Vol. 4649. Lecture Notes in Computer Science. Springer, 2007, pp. 337–348.

[96] Daniel Průša. "Complexity of Matching Sets of Two-Dimensional Patterns by Two-Dimensional On-Line Tessellation Automaton." In: *Int. J. Found. Comput. Sci.* 28.5 (2017), p. 623.

[97] Michael O. Rabin. "Probabilistic Automata." In: *Inf. Control.* 6.3 (1963), pp. 230–245.

[98] Azriel Rosenfeld. *Picture Languages: Formal Models for Picture Recognition*. New York: Academic Press, 1979.

[99] Azriel Rosenfeld and Angela Y. Wu. "Reconfigurable Cellular Computers." In: *Information and Control* 50.1 (1981), pp. 60–84.

[100] Azriel Rosenfeld, Angela Y. Wu, and Tsvi Dubitzki. "Fast language acceptance by shrinking cellular automata." In: *Inf. Sci.* 30.1 (1983), pp. 47–53.

[101] Ronitt Rubinfeld and Asaf Shapira. "Sublinear Time Algorithms." In: *SIAM J. Discrete Math.* 25.4 (2011), pp. 1562–1588.

[102] José Ruiz, Salvador España Boquera, and Pedro García. "Locally Threshold Testable Languages in Strict Sense: Application to the Inference Problem." In: *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings*. 1998, pp. 150–161.

[103] Walter L. Ruzzo. "On Uniform Circuit Complexity." In: *J. Comput. Syst. Sci.* 22.3 (1981), pp. 365–383.

[104] Juan C. Seck-Tuoh-Mora and Genaro J. Martínez. "Graphs Related to Reversibility and Complexity in Cellular Automata." In: *Cellular Automata*. Ed. by Andrew Adamatzky. Encyclopedia of Complexity and Systems Science. Springer, 2018.

[105] Rudolph Sommerhalder and S. Christian van Westrhenen. "Parallel Language Recognition in Constant Time by Cellular Automata." In: *Acta Inf.* 19 (1983), pp. 397–407.

[106] Michael Stratmann and Thomas Worsch. "Leader election in d-dimensional CA in time diam log(diam)." In: *Future Gener. Comput. Syst.* 18.7 (2002), pp. 939–950.

[107] Howard Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Progress in Theoretical Computer Science. Boston, MA: Birkhäuser, 1994.

[108] Madhu Sudan. "Probabilistically checkable proofs." In: *Commun. ACM* 52.3 (2009), pp. 76–84.

[109] Jukka Suomela. "Survey of local algorithms." In: *ACM Comput. Surv.* 45.2 (2013), 24:1–24:40.

[110] Klaus Sutner. "De Bruijn Graphs and Linear Cellular Automata." In: *Complex Syst.* 5.1 (1991).

[111] Véronique Terrier. "Language Recognition by Cellular Automata." In: *Handbook of Natural Computing*. 2012, pp. 123–158.

[112] Clark David Thompson. "A Complexity Theory for VLSI." PhD thesis. Department of Computer Science, Carnegie-Mellon University, Aug. 1980.

[113] Salil P. Vadhan. "Pseudorandomness." In: *Found. Trends Theor. Comput. Sci.* 7.1-3 (2012), pp. 1–336.

[114] Roland Vollmar. "On two modified problems of synchronization in cellular automata." In: *Acta Cybern.* 3.4 (1977), pp. 293–300.

[115] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Texts in Theoretical Computer Science. An EATCS Series. Berlin: Springer, 1999.

[116] Andrew Chi-Chih Yao. "Circuits and Local Computation." In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA*. Ed. by David S. Johnson. ACM, 1989, pp. 186–196.