The Trichotomy of Regular Property Testing

Gabriel Bathie
LaBRI, Université de Bordeaux, France
DIENS, Paris, France

Nathanaël Fijalkow
LaBRI, CNRS, Université de Bordeaux, France

Corto Mascle
LaBRI, Université de Bordeaux, France
MPI-SWS, Kaiserslautern, Germany

Abstract

Property testing is concerned with the design of algorithms making a sublinear number of queries to distinguish whether the input satisfies a given property or is far from having this property. A seminal paper of Alon, Krivelevich, Newman, and Szegedy in 2001 introduced property testing of formal languages: the goal is to determine whether an input word belongs to a given language, or is far from any word in that language. They constructed the first property testing algorithm for the class of all regular languages. This opened a line of work with improved complexity results and applications to streaming algorithms. In this work, we show a trichotomy result: the class of regular languages can be divided into three classes, each associated with an optimal query complexity. Our analysis yields effective characterizations for all three classes using so-called minimal blocking sequences, reasoning directly and combinatorially on automata.

2012 ACM Subject Classification Theory of computation → Regular languages

Keywords and phrases property testing, regular languages

Digital Object Identifier 10.4230/LIPIcs.ICALP.2025.141

Category Track B: Automata, Logic, Semantics, and Theory of Programming

Related Version *Full Version*: `https://arxiv.org/abs/2504.19152`

Funding *Gabriel Bathie*: Partially funded by the grant ANR-20-CE48-0001 from the French National Research Agency.

1 Introduction

Property testing was introduced by Goldreich, Goldwasser and Ron [18] in 1998: it is the study of randomized approximate decision procedures that must distinguishing objects that have a given property from those that are *far* from having it. Because of this relaxation on the specification, the field focuses on very efficient decision procedures, typically with sublinear (or even constant) running time – in particular, the algorithm does not even have the time to read the whole input.

In a seminal paper, Alon et al. [4] introduced property testing of formal languages: given a language L of finite words, the goal is to determine whether an input word u belongs to the language or is ε-far[1] from it, where ε is the precision parameter. The model assumes random access to the input word: a *query* specifies a position in the word and asks for the letter at that position, and the *query complexity* of the algorithm is the worst-case number of queries it makes to the input. Alon et al. [4] showed a surprising result: under the Hamming distance,

[1] Informally, u is ε-far from L means that even by changing an ε-fraction of the letters of u, we cannot obtain a word in L. See Section 2 for a formal definition.

© Gabriel Bathie, Nathanaël Fijalkow, and Corto Mascle;
licensed under Creative Commons License CC-BY 4.0

all regular languages are testable with $\mathcal{O}(\log^3(\varepsilon^{-1})/\varepsilon)$ queries, where the $\mathcal{O}(\cdot)$ notation hides constants that depend on the language, but, crucially, not on the length of the input word. In that paper, they also identified the class of *trivial* regular languages, those for which the answer is always *yes* or always *no* for sufficiently large n, e.g. finite languages or the set of words starting with an a, and showed that testing membership in a *non-trivial* regular language requires $\Omega(1/\varepsilon)$ queries.

The results of Alon et al. [4] leave a multiplicative gap of $\mathcal{O}(\log^3(1/\varepsilon))$ between the best upper and lower bounds. We set out to improve our understanding of property testing of regular languages by closing this gap. Bathie and Starikovskaya obtained in 2021 [8] the first improvement over the result of Alon et al. [4] in more than 20 years:

▶ **Fact 1.1** (From [8, Theorem 5]). *Under the edit distance, every regular language can be tested with $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

Testers under the edit distance are weaker than testers under the Hamming distance, hence this result does not exactly improve the result of Alon et al. [4]. We overcome this shortcoming later in this article: Theorem 4.15 extends the above result to the case of the Hamming distance.

Bathie and Starikovskaya [8] also showed that this upper bound is tight, in the sense that there is a regular language L_0 for which this complexity cannot be further improved, thereby closing the query complexity gap.

▶ **Fact 1.2** (From [8, Theorem 15]). *There is a regular language L_0 with query complexity $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ under the edit distance[2], for all small enough $\varepsilon > 0$.*

Furthermore, it is easy to find specific non-trivial regular languages for which there is an algorithm using only $\mathcal{O}(1/\varepsilon)$ queries, e.g. $L = a^*$ over the alphabet $\{a, b\}$, $L = (ab)^*$ or $L = (aa + bb)^*$.

Hence, these results combined with those of Alon et al. [4] show that there exist trivial languages (that require 0 queries for large enough n), *easy* languages (with query complexity $\Theta(1/\varepsilon)$) and *hard* languages (with query complexity $\Theta(\log(\varepsilon^{-1})/\varepsilon)$). This raises the question of whether there exist languages with a different query complexity (e.g. $\Theta(\log\log(\varepsilon^{-1})/\varepsilon)$), or if every regular is either trivial, easy or hard. This further asks the question of giving a characterization of the languages that belong to each class, inspired by the recent success of exact characterizations of the complexity of sliding window [15] recognition and dynamic membership [6] of regular languages.

In this article, we answer both questions: we show a trichotomy of the complexity of testing regular languages under the Hamming distance[3], showing that there are only the three aforementioned complexity classes (trivial, easy and hard), we give a characterization of all three classes using a combinatorial object called *blocking sequences*, and show that this characterization can be decided in polynomial space (and that it is complete for PSPACE). This trichotomy theorem closes a line of work on improving query complexity for property testers and identifying easier subclasses of regular languages.

[2] Note that, as opposed to testers, lower bounds for the edit distance are stronger than lower bounds of the Hamming distance.

[3] We consider one-sided error testers, also called testing with perfect completeness, see definitions below.

1.1 Related work

A very active branch of property testing focuses on graph properties, for instance one can test whether a given graph appears as a subgraph [2] or as an induced subgraph [3], and more generally every monotone graph property can be tested with one-sided error [5]. Other families of objects heavily studied under this algorithmic paradigm include probabilistic distributions [23, 10] combined with privacy constraints [1], numerical functions [9, 26], and programs [12, 11]. We refer to the book of Goldreich [17] for an overview of the field of property testing.

Testing formal languages. Building upon the seminal work of Alon et al. [4], Magniez et al. [21] gave a tester using $\mathcal{O}(\log^2(\varepsilon^{-1})/\varepsilon)$ queries for regular languages under the edit distance with moves, and François et al. [14] gave a tester using $\mathcal{O}(1/\varepsilon^2)$ queries for the case of the weighted edit distance. Alon et al. [4] also show that context-free languages cannot be tested with a constant number of queries, and subsequent work has considered testing specific context-free languages such as the Dyck languages [24, 13] or regular tree languages [21]. Property testing of formal languages has been investigated in other settings: Ganardi et al. [16] studied the question of testing regular languages in the so-called "sliding window model", while others considered property testing for subclasses of context-free languages in the streaming model: Visibly Pushdown languages [14], Dyck languages [19, 20, 22] or DLIN and LL(k) [7]. A recent application of property testing of regular languages was to detect race conditions in execution traces of distributed systems [27].

1.2 Main result and overview of the paper

We start with a high-level presentation of the approach, main result, and key ideas. In this section we assume familiarity with standard notions such as finite automata; we will detail notations in Section 2.

Let us start with the notion of a property tester for a language L: the goal is to determine whether an input word u belongs to the language L, or whether it is ε-far from it. We say that u of length n is *ε-far from L* with respect to a metric d over words if all words $v \in L$ satisfy $d(u, v) \geq \varepsilon n$, written $d(u, L) \geq \varepsilon n$. Throughout this work and unless explicitly stated otherwise, we will consider the case where d is the Hamming distance, defined for two words u and v as the number of positions at which they differ if they have the same length, and as $+\infty$ otherwise. In that case, $d(u, L) \geq \varepsilon n$ means that one cannot change a proportion ε of the letters in u to obtain a word in L. We assume random access to the input word: a query specifies a position in the word and asks for the letter in this position. A *ε-property tester* (or for short, simply a *tester*) T for a language L is a randomized algorithm that, given an input word u of length n, always answers "yes" if $u \in L$ and answers "no" with probability bounded away from 0 when u is ε-far from L. As in previous works on this topic, we measure the complexity of a tester by its *query complexity*. It is the maximum number of queries that T makes on an input of length n, as a function of n and ε, in the worst case over all words of length n and all possible random choices.

We can now formally define the classes of *trivial, easy* and *hard* regular languages.

▶ **Definition 1.3** (Hard, easy and trivial languages). *Let L be a regular language. We say that:*

- *L is* hard *if the optimal query complexity for a property tester for L is $\Theta(\log(\varepsilon^{-1})/\varepsilon)$.*
- *L is* easy *if the optimal query complexity for a property tester for L is $\Theta(1/\varepsilon)$.*
- *L is* trivial *if there exists $\varepsilon_0 > 0$ such that for all positive $\varepsilon < \varepsilon_0$, there is a property tester and some $n \in \mathbb{N}$ such that the tester makes 0 queries on words of length $\geq n$.*

▶ Remark 1.4. If L is finite, then it is trivial: since there is a bound B on the lengths of its words, a tester can reject words of length at least $n_0 = B + 1$ without querying them. For that reason, we only consider *infinite* languages in the rest of the article.

Our characterization of those three classes uses the notion of *blocking sequence* of a language L. Intuitively, they are sequences of words such that finding those words as factors of a word w proves that w is not in L. We also define a partial order on them, which gives us a notion of *minimal* blocking sequence.

▶ **Theorem 1.5.** *Let L be an infinite regular language recognized by an NFA $\mathcal{A}$ and let $\mathsf{MBS}(\mathcal{A})$ denote the set of minimal blocking sequences of $\mathcal{A}$. The complexity of testing L is characterized by $\mathsf{MBS}(\mathcal{A})$ as follows:*

1. *L is trivial if and only if $\mathsf{MBS}(\mathcal{A})$ is empty;*
2. *L is easy if and only if $\mathsf{MBS}(\mathcal{A})$ is finite and nonempty;*
3. *L is hard if and only if $\mathsf{MBS}(\mathcal{A})$ is infinite.*

In the case where L is recognised by a strongly connected automaton, blocking sequences can be replaced by *blocking factors*. A blocking factor is a single word that is not a factor of any word in L.

Section 2 defines the necessary terms and notations. The rest of the paper is structured as follows. In Sections 3 and 4, we delimit the set of hard languages, that is, the ones that require $\Theta(\log(\varepsilon^{-1})/\varepsilon)$ queries. More precisely, Section 3 focuses on the subcase of languages defined by *strongly connected automata*. First, we combine the ideas of Alon et al. [4] with those that Bathie and Starikovskaya presented in [8] to obtain a property tester that uses $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries for any language with a strongly connected automaton, under the Hamming distance. Second, we show that if the language of a strongly connected automaton has infinitely many blocking factors then it requires $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries. This result generalizes the result of Bathie and Starikovskaya [8], which was for a single language, to all regular languages with infinitely many minimal blocking factors. We use Yao's minimax principle [28]: this involves constructing a hard distribution over inputs, and showing that any deterministic property testing algorithms cannot distinguish between positive and negative instances against this distribution.

In Section 4, we extends those results to all automata. The interplay with the previous section is different for the upper and the lower bound.For the upper bound of $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries, we use a natural but technical extension of the proof in the strongly connected case. Note that this result is an improvement over the result of Bathie and Starikovskaya [8], which works under the edit distance, and testers for the Hamming distance are also testers for the edit distance. For the lower bound of $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries for languages with infinitely many minimal blocking sequences, we reduce to the strongly connected case. The main difficulty is that it is not enough to consider strongly connected components in isolation: there exists finite automata that contain a strongly connected component that induces a hard language, yet the language of the whole automaton is easy. We solve this difficulty by carefully defining the notion of minimality for a blocking sequence.

Section 5 completes the trichotomy, by characterising the easy and trivial languages. We show that languages of automata with finitely many blocking sequences can be tested with $\mathcal{O}(1/\varepsilon)$ queries. We also prove that if an automaton has at least one blocking sequence, then it requires $\Omega(1/\varepsilon)$ queries to be tested, by showing that the languages that our notion of trivial language coincides with the one given by Alon et al. [4]. By contrast, we show that automata without blocking sequences recognize trivial languages.

Once we have the trichotomy, it is natural to ask whether it is effective: given an automaton $\mathcal{A}$, can we determine if its language is trivial, easy or hard? The answer is yes, and we show in Section 6 that all three decision problems are PSPACE-complete, even for strongly connected automata.

2 Preliminaries

Words and automata. We write Σ^* (resp. Σ^+) for the set of finite words (resp. non-empty words) over the alphabet Σ. The length of a word u is denoted $|u|$, and its ith letter is denoted $u[i]$. The empty word is denoted γ. Given $u \in \Sigma^*$ and $0 \leq i, j \leq |u| - 1$, define $u[i..j]$ as the word $u[i]u[i+1]\dots u[j]$ if $i \leq j$ and γ otherwise. Further, $u[i..j)$ denotes the word $u[i..j-1]$. A word w is a *factor* (resp. *prefix*, *suffix*) of u is there exist indices i, j such that $w = u[i..j]$ (resp. with $i = 0, j = |u| - 1$). We use $w \preccurlyeq u$ to denote "w is a factor of u". Furthermore, if w is a factor of u and $w \neq u$, we say that w is a *proper factor* of u.

A *nondeterministic finite automaton* (NFA) $\mathcal{A}$ is a transition system defined by a tuple $(Q, \Sigma, \delta, q_0, F)$, with Q a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ the transition function, $q_0 \in Q$ the initial state and $F \subseteq Q$ the set of final states. The semantics is as usual [25]. When there is a path from a state p to a state q in $\mathcal{A}$, we say that q is reachable from p and that p is co-reachable from q. In this work, we assume w.l.o.g. that all NFA $\mathcal{A}$ are *trim*, i.e., every state is reachable from the initial state and co-reachable from some final state.

Property testing.

▶ **Definition 2.1.** *Let L be a language, let u be a word of length n, let $\varepsilon > 0$ be a precision parameter and let $d : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{+\infty\}$ be a metric. We say that the word u is* ε-far from L w.r.t. d *if $d(u, L) \geq \varepsilon n$, where*

$$d(u, L) := \inf_{v \in L} d(u, v).$$

We assume random access to the input word: a query specifies a position in the word and asks for the letter in this position.

Throughout this work and unless explicitly stated otherwise, we will consider the case where d is the Hamming distance, defined for two words u and v as the number of positions at which they differ if they have the same length, and as $+\infty$ otherwise. In that case, $d(u, L) \geq \varepsilon n$ means that one cannot change an ε-fraction of the letters in u to obtain a word in L.

A *ε-property tester* (or simply a *tester*) T for a language L is a randomized algorithm that, given an input word u, always answers "yes" if $u \in L$ and answers "no" with probability bounded away from 0 when u is ε-far from L.

▶ **Definition 2.2.** *A property tester for the language L with precision $\varepsilon > 0$ is a randomized algorithm T that, for any input u of length n, given random access to u, satisfies the following properties:*

$$\text{if } u \in L, \text{ then } T(u) = 1,$$
$$\text{if } u \text{ is } \varepsilon\text{-far from } L, \text{ then } \mathbb{P}(T(u) = 0) \geq 2/3.$$

The query complexity of T is a function of n and ε that counts the maximum number of queries that T makes over all inputs of length n and over all possible random choices.

We measure the complexity of a tester by its *query complexity*. Let us emphasize that throughout this article we focus on so-called "testers with perfect completeness", or "one-sided error": if a word is in the language, the tester answers positively (with probability 1). In particular our characterization applies to this class. Because they are based on the notion of blocking factors that we will discuss below, all known testers for regular languages [4, 21, 14, 8] have perfect completeness.

In this paper, we assume that the automaton $\mathcal{A}$ that describes the tested language L is *fixed*, and not part of the input. Therefore, we consider its number of states m as a constant.

Graphs and periodicity. We now recall tools introduced by Alon et al. [4] to deal with periodicity in finite automata.

Let $G = (V, E)$ with $E \subseteq V^2$ be a directed graph. A *strongly connected component* (or SCC) of G is a maximal set of vertices that are all reachable from each other. It is *trivial* if it contains a single state with no self-loop on it. We say that G is *strongly connected* if its entire set of vertices is an SCC.

The period $\lambda = \lambda(G)$ of a non-trivial strongly connected graph G is the greatest common divisor of the length of the cycles in G. Following the work of Alon et al. [4], we will use the following property of directed graphs.

▶ **Fact 2.3** (From [4, Lemma 2.3]). *Let $G = (V, E)$ be a non-empty, non-trivial, strongly connected graph with finite period $\lambda = \lambda(G)$. Then there exists a partition $V = Q_0 \sqcup \ldots \sqcup Q_{\lambda-1}$ and a reachability constant $\rho = \rho(G)$ that does not exceed $3|V|^2$ such that:*

1. *For every $0 \leq i, j \leq \lambda - 1$ and for every $s \in Q_i, t \in Q_j$, the length of any directed path from s to t in G is equal to $(j - i) \mod \lambda$.*
2. *For every $0 \leq i, j \leq \lambda - 1$, for every $s \in Q_i, t \in Q_j$ and for every integer $r \geq \rho$, if $r = (j - i) \pmod \lambda$, then there exists a directed path from u to v in G of length r.*

The sets $(Q_i : i = 0, \ldots, \lambda - 1)$ are the *periodicity classes* of G. In what follows, we will slightly abuse notation and use Q_i even when $i \geq \lambda$ to mean $Q_{i \pmod \lambda}$.

An automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ defines an underlying graph $G = (Q, E)$ where $E = \{(p, q) \in Q^2 \mid \exists a \in \Sigma : q \in \delta(p, a)\}$. In what follows, we naturally extend the notions defined above to finite automata through this graph G. Note that the numbering of the periodicity classes is defined up to a shift mod λ: we can thus always assume that Q_0 is the class that contains the initial state q_0. The period of $\mathcal{A}$ is written $\lambda(\mathcal{A})$.

Positional words and positional languages. Consider the language $L_3 = (ab)^*$. The word $v = ab$ can appear as a factor of a word $u \in L_3$ if v occurs at an even position (e.g. position 0, 2, etc.) in u. However, if v occurs at an *odd* position in u, then u cannot be in L_3. Therefore, v can be used to witness that u is not in L_3, but only if we find it at an odd position. This example leads us to introducing *p-positional words*, which additionally encode information about the index of each letter modulo an integer p.

More generally, we will associate to each regular language a period λ, and working with λ-positional words will allow us to define blocking factors in a position-dependent way without explicitly considering the index at which the factor occurs.

▶ **Definition 2.4** (Positional words). *Let p be a positive integer. A* p-positional word *is a word over the alphabet $\mathbb{Z}/p\mathbb{Z} \times \Sigma$ of the form $(n \pmod p, a_0)((n+1) \pmod p, a_1) \cdots ((n+\ell) \pmod p, a_\ell)$ for some non-negative integer n. If $u = a_0 \cdots a_\ell$, we write $(n : u)$ to denote this word.*

With this definition, if $u = abcd$ and we consider the 2-positional word $\tau = (0 : u)$, the factor bc appears at position 1 in u and is mapped to the factor $\mu = (1, a)(0, b)$. In this case, even when taking factors of μ, we still retain the (congruence classes of the) indices in the original word τ.

Any strongly connected finite automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ can naturally be extended into an automaton $\widehat{\mathcal{A}}$ over $\lambda(\mathcal{A})$-positional words with $\lambda(\mathcal{A})|Q|$ states. It suffices to keep track in the states of the current state of $\mathcal{A}$ and the number of letters read modulo $\lambda(\mathcal{A})$.

We call the language recognized by $\widehat{\mathcal{A}}$ the *positional language of* $\mathcal{A}$, and denote it $\mathcal{TL}(\mathcal{A})$. This definition is motivated by the following property:

▶ **Property 2.5.** *For any word $u \in \Sigma^*$, we have $u \in \mathcal{L}(\mathcal{A})$ if and only if $(0 : u) \in \mathcal{TL}(\mathcal{A})$.*

Positional words make it easier to manipulate factors with positional information, hence we phrase our property testing results in terms of positional languages. Notice that a property tester for $\mathcal{TL}(\mathcal{A})$ immediately gives a property tester for $\mathcal{L}(\mathcal{A})$, as one can simulate queries to $(0 : u)$ with queries to u by simply pairing the index of the query modulo $\lambda(\mathcal{A})$ with its result.

3 Hard Languages for Strongly Connected NFAs

Before considering the case of arbitrary NFAs, we first study the case of strongly connected NFAs, which are NFAs such that for any pair of states $p, q \in Q$, there exists a word w such that $p \xrightarrow{w} q$. We will later generalize the results of this section to all NFAs.

We show that the query complexity of the language of such an NFA $\mathcal{A}$ can be characterized by the cardinality of the set of *minimal blocking factors* of $\mathcal{A}$, which are factor-minimal $\lambda(\mathcal{A})$-positional words that witness the fact that a word does not belong to $\mathcal{TL}(\mathcal{A})$. In this section, we consider a fixed NFA $\mathcal{A}$ and simply use "positional words" to refer to λ-positional words, where $\lambda = \lambda(\mathcal{A})$ is the period of $\mathcal{A}$.

▶ **Definition 3.1** (Blocking factors). *Let $\mathcal{A}$ be a strongly connected NFA. A positional word τ is a* blocking factor *of $\mathcal{A}$ if for any other positional word μ we have $\tau \preccurlyeq \mu \Rightarrow \mu \notin \mathcal{TL}(\mathcal{A})$.*

Further, we say that τ is a minimal *blocking factor of $\mathcal{A}$ if no proper factor of τ is a blocking factor of $\mathcal{A}$. We use $\mathsf{MBF}(\mathcal{A})$ to denote the set of all minimal blocking factors of $\mathcal{A}$.*

Intuitively and in terms of automata, the positional word $(i : u)$ is blocking for $\mathcal{A}$ if it does not label any transition in $\mathcal{A}$ labeled by u starting from a state of Q_i. (This property is formally established later in Lemma 3.3.) The main result of this section is the following:

▶ **Theorem 3.2.** *Let L be an infinite language recognised by a strongly connected NFA $\mathcal{A}$. If $\mathsf{MBF}(\mathcal{A})$ is infinite, then L is hard, i.e., it has query complexity $\Theta(\log(\varepsilon^{-1})/\varepsilon)$*

This result gives both an upper bound of $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ and a lower bound of $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ on the query complexity of a tester for L: we prove the upper bound in Section 3.2 and the lower bound in Section 3.3.

3.1 Positional words, blocking factors and strongly connected NFAs

We formalize the intuition we gave earlier about blocking factors. In Section 3.2, we highlight the connection between property testing and blocking factors in strongly connected NFAs.

▶ **Lemma 3.3.** *A positional word $\tau = (i : u)$ is a blocking factor for $\mathcal{A}$ iff for every states $p \in Q_i, q \in Q$, we have $p \not\xrightarrow{u} q$.*

The above claim allows us to interchangeably use the statements "u is ε-far from $\mathcal{L}(\mathcal{A})$" and "$(0 : u)$ is ε-far from $\mathcal{TL}(\mathcal{A})$".

3.2 An efficient property tester for strongly connected NFAs

In this section, we show that for any strongly connected NFA $\mathcal{A}$, there exists an ε-property tester for $\mathcal{L}(\mathcal{A})$ that uses $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries.

▶ **Theorem 3.4.** *Let $\mathcal{A}$ be a strongly connected NFA. For any $\varepsilon > 0$, there exists an ε-property tester for $\mathcal{L}(\mathcal{A})$ that uses $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

Our proof is similar to the one given in [8], with one notable technical improvement: we use a new method for sampling factors in u, which greatly simplifies the correctness analysis.

3.2.1 An efficient sampling algorithm

We first introduce a sampling algorithm (Algorithm 1) that uses few queries and has a large probability of finding at least one factor from a large set $\mathcal{S}$ of disjoint "special" factors. Using this algorithm on a large set of disjoint blocking factors gives us an efficient property tester for strongly connected NFAs. We will re-use this sampling procedure later in the case of general NFAs (Theorem 4.15).

The procedure is fairly simple: the algorithm samples factors of various lengths in u at random. On the other hand, the correctness of the tester is far from trivial. The lengths and the number of factors of each length are chosen so that the number of queries is minimized and the probability of finding a "special" factor is maximized, regardless of their repartition in u. (In what follows, the "special" factors are blocking factors.)

Algorithm 1 Efficient generic sampling algorithm.

```
1:  function OneSample(u, ℓ)
2:      i ← uniform(0, n − 1)
3:      l ← max(i − ℓ, 0), r ← min(i + ℓ, n − 1)
4:      return u[l..r]
5:  function Sampler(u, N, L)
6:      n ← |u|
7:      β ← n/N
8:      T ← ⌈log(L)⌉
9:      F ← ∅
10:     for t = 0 to T do
11:         ℓ_t ← 2^t, r_t ← ⌈2 ln(3)β/ℓ_t⌉
12:         for i = 0 to r_t do
13:             F ← F ∪ {OneSample(u, ℓ_t)}
14:     return F
```

▷ Claim 3.5. A call to Sampler(u, N, L) (Algorithm 1) makes $\mathcal{O}(n \log(L)/N)$ queries to u.

▶ **Lemma 3.6.** *Let u be a word of length n, and consider a set $\mathcal{S}$ containing at least N disjoint factors of u, each of length at most L. A call to the function Sampler(u, N, L) (Algorithm 1) returns a set $\mathcal{F}$ of factors of u such that there exists a word of $\mathcal{S}$ that is a factor of some word of $\mathcal{F}$, with probability at least $2/3$.*

3.2.2 The tester

The algorithm for Theorem 3.4 is given in Algorithm 2.

Algorithm 2 Generic ε-property tester that uses $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries.

```
function Tester(u, ε)
    n ← |u|, m ← |Q|
    L ← 12m²/ε
    if L(A) ∩ Σ^n = ∅ then
        Reject
    else if n < L then
        Query all of u and run A on it
        Accept if and only if A accepts
    else
        F ← Sampler((0 : u), n/L, L)
        Reject if and only if F contains a blocking factor for A.
```

We now show that Algorithm 2 is a property tester for $\mathcal{L}(\mathcal{A})$ that uses $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries. In what follows, we use n to denote the length of the input word u and m to denote the number of states of $\mathcal{A}$.

▷ Claim 3.7. The tester given in Algorithm 2 makes $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries to u.

Alon et al. [4, Lemma 2.6] first noticed that if a word u is ε-far from $\mathcal{L}(\mathcal{A})$, then it contains $\Omega(\varepsilon n)$ short factors that witness the fact that u is not in $\mathcal{L}(\mathcal{A})$. We start by translating the lemma of Alon et al. on "short witnesses" to the framework of blocking factors. More precisely, we show that if u is ε-far from $\mathcal{L}(\mathcal{A})$, then $(0 : u)$ contains many disjoint (i.e. non-overlapping) blocking factors.

▶ **Lemma 3.8.** *Let $\varepsilon > 0$, let u be a word of length $n \geq 6m^2/\varepsilon$ and assume that $\mathcal{L}(\mathcal{A})$ contains at least one word of length n. If $\tau = (0 : u)$ is ε-far from $\mathcal{TL}(\mathcal{A})$, then τ contains at least $\varepsilon n/(6m^2)$ disjoint blocking factors.*

Next, we show that if u is ε-far from $\mathcal{L}(\mathcal{A})$, then $(0 : u)$ contains $\Omega(\varepsilon n)$ blocking factors, each of length $\mathcal{O}(1/\varepsilon)$.

▶ **Lemma 3.9.** *Let $\varepsilon > 0$, let u be a word of length $n \geq 6m^2/\varepsilon$ and assume that $\mathcal{L}(\mathcal{A})$ contains at least one word of length n. If u is ε-far from $\mathcal{L}(\mathcal{A})$, then the positional word $(0 : u)$ contains at least $\varepsilon n/(12m^2)$ disjoint blocking factors of length at most $12m^2/\varepsilon$.*

Proof. Let $u, \mathcal{A}$ be a word and an automaton satisfying the above hypotheses. By Lemma 3.8, $(0 : u)$ contains at least $\varepsilon n/(6m^2)$ *disjoint* blocking factors. As these factors are disjoint, at most half of them (that is, $\varepsilon n/(12m^2)$ of them) can have length greater than $12m^2/\varepsilon$, as the sum of their lengths cannot exceed n. ◀

Proof of Theorem 3.4. First, note that if $u \in \mathcal{L}(\mathcal{A})$, $(0 : u)$ cannot contain a blocking factor for $\mathcal{A}$, hence Algorithm 2 always accepts u. Next, if $\mathcal{L}(\mathcal{A}) \cap \Sigma^n$ is empty or if $|u| \leq L = 12m^2/\varepsilon$, the tester has the same output as $\mathcal{A}$, hence it is correct.

In the remaining case, u is long enough and ε-far from $\mathcal{L}(\mathcal{A})$, hence Lemma 3.9 gives us a large set of short blocking factors in $(0 : u)$: this is exactly what the Sampler function needs to find at least one factor containing a blocking factor with probability at least $2/3$. More precisely, by Lemma 3.9, $(0 : u)$ contains at least $\varepsilon n/(12m^2) = n/L$ blocking factors of length at most $L = 12m^2/\varepsilon$, hence the conditions of Lemma 3.6 are satisfied.

As a factor containing a blocking factor is also a blocking factor, the set $\mathcal{F}$ computed on line 10 of Algorithm 2 contains at least one blocking factor with probability at least 2/3, and Algorithm 2 satisfies Definition 2.2. ◀

3.3 Lower bound from infinitely many minimal blocking factors

We now show that languages with infinitely many *minimal* blocking factors are hard, i.e. any tester for such a language requires $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries.

Let us first give an example that will motivate our construction. Consider the parity language P consisting of words that contain an even number of b's, over the alphabet $\{a, b\}$. Distinguishing $u \in P$ from $u \notin P$ requires $\Omega(n)$ queries, as changing the letter at single position can change membership in P. However, P is trivial to test, as any word is at distance at most 1 from P, for the same reason. Now, consider language L_2 consisting of words over $\{a, b, c, d\}$ such that between a c and the next d, there is a word in P. Intuitively, this language encodes multiple instances of P, hence we can construct words ε-far from L_2, and each instance is hard to recognize for property testers, hence the whole language is. In [8, Theorem 15], Bathie and Starikovskaya proved a lower bound of $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ on the query complexity of any property tester for L_2, matching the upper bound in the same paper.

The minimal blocking factors of L_2 include all words for the form cvd where $v \notin P$: there are infinitely many such words. This is no coincidence: we show that this lower bound can be lifted to any language with infinitely many minimal blocking factors.

▶ **Theorem 3.10.** *Let $\mathcal{A}$ be a strongly connected NFA. If $\mathsf{MBF}(\mathcal{A})$ is infinite, then there exists a constant ε_0 such that for any $\varepsilon < \varepsilon_0$, any ε-property tester for $L = \mathcal{L}(\mathcal{A})$ uses $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

The proof of this result is full generalization of the lower bound against the "repeated Parity" example given above.

Our proof is based on (a consequence of) Yao's Minimax Principle [28]: if there is a distribution $\mathcal{D}$ over inputs such that any *deterministic* algorithm that makes at most q queries errs on $u \sim \mathcal{D}$ with probability at least p, then any *randomized* algorithm with q queries errs with probability at least p on some input u.

To prove Theorem 3.10, we first exhibit such a distribution $\mathcal{D}$ for $q = \Theta(\log(\varepsilon^{-1})/\varepsilon)$. We take the following steps:

1. we show that with high probability, an input u sampled w.r.t. $\mathcal{D}$ is either in or ε-far from L (Lemma 3.11),
2. we show that with high probability, any deterministic tester that makes fewer than $c \cdot \log(\varepsilon^{-1})/\varepsilon$ queries (for a suitable constant c) cannot distinguish whether the instance u is positive or ε-far, hence it errs with large probability.
3. combine the above two results to prove Theorem 3.10 via Yao's Minimax principle.

Constructing a Hard Distribution $\mathcal{D}$

To sample an input w.r.t. $\mathcal{D}$, we first sample a uniformly random bit π: if $\pi = 1$, we construct a word u that belongs to L, and if $\pi = 0$, u will be far from L with high probability. The input word is then conceptually divided into εn disjoint intervals. For the j-th interval, we sample a random variable $\kappa_j \in \{0, 1, \ldots, \log(1/\varepsilon)\}$: this random variable describes the content of the interval. If $\pi = 1$, we fill the interval with *non-blocking* factors of length $\mathcal{O}(2^{\kappa_j})$, and if $\pi = 0$, we instead fill the interval with *blocking* factors of the same length, chosen to be very similar the non-blocking factors, making it hard to distinguish between the two cases with few queries. By carefully choosing the distribution for κ_j, we can ensure that when $\pi = 0$, the resulting instance u is ε-far from L with high probability.

▶ **Lemma 3.11.** *Conditioned on $\pi = 0$, the probability of the event $\mathcal{F} = \{u$ is ε-far from $\mathcal{TL}(\mathcal{A})\}$ goes to 1 as n goes to infinity.*

▶ **Corollary 3.12.** *For large enough n, we have $\mathbb{P}(\mathcal{F}) \geq 5/12$.*

Intuitively, our distribution is hard to test because positive and negative instance are very similar. Therefore, a tester with few queries will likely not be able to tell them apart: the perfect completeness constraint forces the tester to accept in that case. Below, we establish this result formally.

▶ **Lemma 3.13.** *Let T be a deterministic tester with perfect completeness (i.e. one sided error, always accepts $\tau \in \mathcal{TL}(\mathcal{A})$) and let q_j denote the number of queries that it makes in the j-th interval. Conditioned on the event $\mathcal{M} = \{\forall j$ s.t. $\kappa_j > 0, q_j < 2^{\kappa_j}\}$, the probability that T accepts u is 1.*

Next, we show that if a tester makes few queries, then the event $\mathcal{M}$ has large probability.

▶ **Lemma 3.14.** *Let T be a deterministic tester, let q_j denote the number of queries that it makes in the j-th interval, and assume that T makes at most $\frac{1}{72} \cdot \log(\varepsilon^{-1})/\varepsilon$ queries, i.e. $\sum_j q_j \leq \frac{1}{72} \cdot \log(\varepsilon^{-1})/\varepsilon$. The probability of the event $\mathcal{M} = \{\forall j$ s.t. $\kappa_j > 0, q_j < 2^{\kappa_j}\}$ is at least 11/12.*

We are now ready to prove Theorem 3.10.

Proof of Theorem 3.10. We want to show that any tester with perfect completeness for $\mathcal{L}(\mathcal{A})$ requires at least $\frac{1}{72} \cdot \log(\varepsilon^{-1})/\varepsilon$ queries, by showing that any tester with fewer queries errs with probability at least 1/3. We show that any **deterministic** algorithm T with perfect completeness that makes less than $\frac{1}{72} \cdot \log(\varepsilon^{-1})/\varepsilon$ queries errs on u when $(0 : u) \sim \mathcal{D}$ with probability at least 1/3, and conclude using Yao's Minmax principle.

Consider such an algorithm T. The probability that T makes an error on u is lower-bounded by the probability that u is ε-far from $\mathcal{L}(\mathcal{A})$ and T accepts, which in turn is larger than the probability of $\mathcal{M} \cap \mathcal{F}$. By Corollary 3.12, we have $\mathbb{P}(\mathcal{F}) \geq 5/12$, and by Lemma 3.14, $\mathbb{P}(\mathcal{M})$ is at least 11/12. Therefore, we have

$$\mathbb{P}(T \text{ errs}) \geq \mathbb{P}(\mathcal{M} \cap \mathcal{F}) \geq 1 - 7/12 - 1/12 = 4/12 = 1/3.$$

This concludes the proof of Theorem 3.10, and consequently of Theorem 3.2. ◀

Next, we show that if a tester makes few queries, then the event $\mathcal{M}$ has large probability.

4 Characterisation of Hard Languages for All NFAs

In this section we extend the results of the previous section to all finite automata. This extension is based on a generalization of blocking factors: we introduce *blocking sequences*, which are sequences of factors that witness the fact that we cannot take any path through the strongly connected components of the automaton. For the lower bound, we define a suitable partial order on blocking sequences, which extends the factor relation on words to those sequences, and allows us to define *minimal* blocking sequences.

4.1 Blocking sequences

4.1.1 Examples motivating blocking sequences

Before presenting the technical part of the proof, let us go through two examples, which motivate the notions that we introduce and illustrate some of the main difficulties.

▶ **Example 4.1.** Consider the automaton $\mathcal{A}_1$ depicted in Figure 1: it recognizes the language L_1 of words in which all c's appear before the first b, over the alphabet $\{a, b, c\}$.

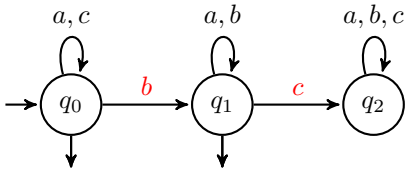


Figure 1 An automaton $\mathcal{A}_1$ that recognizes the language $L_1 = (a + c)^*(a + b)^*$.

The set of minimal blocking factors of $\mathcal{A}_1$ is infinite: it is the language ba^*c. Yet, L_1 is easy to test: we sample $\mathcal{O}(1/\varepsilon)$ letters at random, answer "no" if the sample contains a c occurring after a b, and "yes" otherwise. To prove that this yields a property tester, we rely on the following property:

▶ **Property 4.2.** *If u is ε-far from L_1, then it can be decomposed into $u = u_1u_2$ where u_1 contains $\Omega(\varepsilon n)$ letters b and u_2 contains $\Omega(\varepsilon n)$ letters c.*

The pair of factors (b, c) is an example of blocking sequence: a word that contains an occurrence of the first followed by an occurrence of the second cannot be in L_1. We can also show that a word ε-far from L_1 must contains many disjoint blocking sequences – this property (Lemma 4.17) underpins the algorithm for general regular languages.

What this example shows is that blocking factors are not enough: we need to consider sequences of factors, yielding the notion of *blocking sequences*. Intuitively, a blocking sequence for L is a sequence $\sigma = (v_1, \dots, v_k)$ of (positional) words such that if each word of the sequence appears in u, with the occurrences of the v_i's ordered as in σ, then u is not in L.[4] While L_1 has infinitely many minimal blocking factors, it has a single minimal blocking sequence $\sigma = (b, c)$.

Notice that the (unique) blocking sequence (b, c) can be visualized on Figure 1: it is composed of the red letters that label transitions between the different SCCs. This is no coincidence: in many simple cases, blocking sequences are exactly sequences that contain one blocking factors for each SCC. This fact could lead one to believe that the set of minimal blocking sequences is exactly the set of sequences of minimal blocking factors, one for each SCC. In particular, this would imply that as soon as one SCC has infinitely many minimal blocking factors, the language of the whole automaton is hard to test. We show in the next example that this is not always the case, because SCCs might share minimal blocking factors.

▶ **Example 4.3.** Consider the automaton in Figure 2: it has two SCCs and a sink state. The minimal blocking factors of the first SCC are given by $B_1 = be^*c + a$, and $B_2 = \{a\}$ for the second SCC. This automaton is easy to test: intuitively, a word that is ε-far from this

[4] This is not quite the definition, but it conveys the right intuition.

language has to contain many a's, as otherwise we can make it accepted by deleting all a's, thanks to the second SCC. However, a is also a blocking factor of the first SCC, therefore, as soon as we find two a's in the word, we know that it is not in L_2.

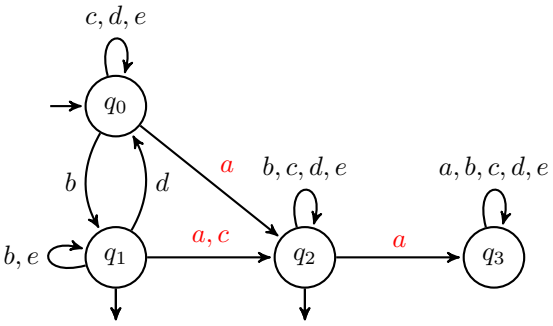


Figure 2 An automaton $\mathcal{A}_2$ that recognizes the language $L_2 = [((c+d+e)^*b(b+e)^*d)^*a](b+c+d+e)^*$.

The crucial facts here are that the set B_2 of minimal blocking factors of the second SCC is finite and it is a subset of B_1: the infinite nature of B_1 is made irrelevant because any word far from the language contains many a's. Therefore, $\mathcal{A}_2$ has a *single* minimal blocking sequence, $\sigma = (a)$.

4.1.2 Portals and SCC-paths

Intuitively, blocking sequences are sequences of blocking factors of successive strongly connected components. To formalize this intuition, we use *portals*, which describe how a run in the automaton interacts with a strongly connected component, and *SCC-paths*, that describe a succession of portals.

In what follows, we fix an NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, \{q_f\})$. We assume w.l.o.g. that $\mathcal{A}$ has a unique final state q_f. Let $\mathscr{S}$ be the set of SCCs of $\mathcal{A}$. We define the partial order relation $\leq_{\mathcal{A}}$ on $\mathscr{S}$ by $S \leq_{\mathcal{A}} T$ if and only if T is reachable from S. We write $<_{\mathcal{A}}$ for its strict part $\leq_{\mathcal{A}} \setminus \geq_{\mathcal{A}}$. These relations can be naturally extended to states through their SCC: if $s \in S$ and $t \in T$, then $s \leq_{\mathcal{A}} t$ if and only if $S \leq_{\mathcal{A}} T$.

We define p as the least common multiple of the lengths of all simple cycles of $\mathcal{A}$. Given a number $k \in \mathbb{Z}/p\mathbb{Z}$, we say that a state t is k-reachable from a state s if there is a path from s to t of length k modulo p. In what follows, we use "positional words" for p-positional words with this value of p.

▶ **Definition 4.4** (Portal). *A* portal *is a 4-tuple $P = s, x \rightsquigarrow t, y \in (Q \times \mathbb{Z}/p\mathbb{Z})^2$, such that s and t are in the same SCC. It describes the first and last states visited by a path in an SCC, and the positions x, y (modulo p) at which it first and lasts visits that SCC.*

The positional language of a portal is the set

$$\mathcal{L}(s, x \rightsquigarrow t, y) = \{(x : w) \mid s \xrightarrow{w} t \wedge x + |w| = y \pmod p\}.$$

Portals were already defined by Alon et al. [4], in a slightly different way. Our definition will allow us to express blocking sequences more naturally.

▶ **Definition 4.5.** *A positional word $(n : u)$ is blocking for a portal P if it is not a factor of any word of $\mathcal{L}(P)$. In other words, there is no path that starts in s and ends in t, of length $y - x$ modulo p, which reads u after $n - x$ steps modulo p.*

The above definition matches the definition of blocking factors for strongly connected automata. This is no coincidence: we show in the next lemma that the language of a portal has a strongly connected automaton.

▶ **Lemma 4.6.** *Let $\mathcal{A}$ be an automaton and P a portal of $\mathcal{A}$. There is a strongly connected NFA with at most $p|\mathcal{A}|$ states that recognizes $L' = \mathcal{L}(P)$.*

Portals describe the behavior of a run inside a single strongly connected component of the automaton. Next, we introduce SCC-paths, which describe the interaction of a run with multiple SCCs and between two successive SCCs.

▶ **Definition 4.7** (SCC-path)**.** *An* SCC-path π *of $\mathcal{A}$ is a sequence of portals linked by single-letter transitions $\pi = s_0, x_0 \rightsquigarrow t_0, y_0 \xrightarrow{a_1} s_1, x_1 \rightsquigarrow t_1, y_1 \cdots \xrightarrow{a_k} s_k, x_k \rightsquigarrow t_k, y_k$, such that for all $i \in \{1, \dots, k\}$, $x_i = y_{i-1} + 1 \pmod p$, $t_{i-1} \xrightarrow{a_i} s_i$, and $t_{i-1} <_{\mathcal{A}} s_i$.*

Intuitively, an SCC-path is a description of the states and positions at which a path through the automaton enters and leaves each SCC.

▶ **Definition 4.8.** *The language $\mathcal{L}(\pi)$ of an SCC-path $\pi = P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots P_k$ is the set*

$$\mathcal{L}(\pi) = \mathcal{L}(P_0)a_1\mathcal{L}(P_2)a_2 \cdots \mathcal{L}(P_k).$$

We say that π is *accepting* if $P_0 = s_0, x_0 \rightsquigarrow t_0, y_0$, $P_k = s_k, x_k \rightsquigarrow t_k, y_k$ with $x_0 = 0$, $s_0 = q_0$, $t_k = q_f$ and $\mathcal{L}(\pi)$ is non-empty.

▶ **Lemma 4.9.** *We have $\mathcal{TL}(\mathcal{A}) = \bigcup_{\pi \text{ accepting}} \mathcal{L}(\pi)$.*

As a consequence the distance between a word μ and the (positional) language of $\mathcal{A}$ is equal to the minimum of the distances between μ and the languages of the SCC-paths of $\mathcal{A}$.

▶ **Corollary 4.10.** *For any positional word μ, we have*

$$d(\mu, \mathcal{TL}(\mathcal{A})) = \min_{\pi \text{ accepting}} d(\mu, \mathcal{L}(\pi)).$$

Decomposing $\mathcal{A}$ as a union of SCC-paths allows us to use them as an intermediate step to define blocking sequences. We earlier defined blocking factors for portals: we now generalize this definition to blocking sequences for SCC-paths, to finally define blocking sequence of automata.

▶ **Definition 4.11** ((Strongly) Blocking Sequences for SCC-paths)**.** *We say that a sequence $\sigma = (\mu_1, \dots, \mu_\ell)$ of positional factors is blocking for an SCC-path $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ if there is a sequence of indices $i_0 \le i_1 \le \cdots \le i_k$ such that for every j, μ_{i_j} is blocking for P_j.*

Furthermore, if there is a sequence of indices $i_0 < i_1 < \cdots < i_k$ with the same property, then σ is said to be strongly blocking *for π.*

Note that, crucially, in the definition of blocking sequences, consecutive indices i_j and i_{j+1} can be equal, i.e. a single factor of the sequence may be blocking for multiple consecutive SCCs in the SCC-path. This choice is motivated by Example 4.3, where the language is easy because consecutive SCCs share blocking factors.

We say that two occurrences of blocking sequences in a word μ are *disjoint* if the occurrences of their factors appear on disjoint sets of positions in μ.

In the strongly connected case, we had the property that if μ contains an occurrence of a factor blocking for $\mathcal{A}$, then μ is not in the language of $\mathcal{A}$. The following lemma gives an extension of this result to *strongly* blocking sequences and the language of an SCC-path.

▶ **Lemma 4.12.** *Let π be an SCC-path. If μ contains a strongly blocking sequence for π, then $\mu \notin \mathcal{L}(\pi)$.*

We can now define sequences that are blocking for an automaton: they are sequences that are blocking for *every* accepting SCC-path of the automaton.

▶ **Definition 4.13** (Blocking sequence for $\mathcal{A}$). *Let $\sigma = (\mu_1, \dots, \mu_\ell)$ be a sequence of positional words. We say that σ is blocking for $\mathcal{A}$ if it is blocking for all accepting SCC-paths of $\mathcal{A}$.*

As an example, observe that the sequences $((0 : ab), (1 : ab))$ and $((0 : aa), (0 : b))$ are both blocking for the automaton displayed in Figure 3 (see Example 4.14).

▶ **Example 4.14.** Consider the automaton displayed in Figure 3. The lcm of the lengths of its simple cycles is $p = 2$. This automaton has six accepting SCC-paths, for instance

$$\pi_1 = q_0, 0 \rightsquigarrow q_0, 0 \xrightarrow{a} q_1, 1 \rightsquigarrow q_1, 1 \xrightarrow{a} q_3, 0 \rightsquigarrow q_3, 0 \xrightarrow{b} q_4, 1 \rightsquigarrow q_4, 1$$

The language of the portal π_1 is $a(ba)^*a(a^2)^*b$. A blocking sequence for this SCC-path is $((0 : aa), (0 : b))$, which is in fact blocking for all of the SCC-paths.

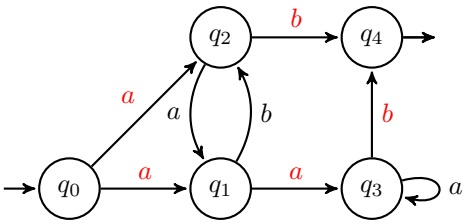


Figure 3 Automaton used for Example 4.14.

On the other hand, $((0 : ab))$ is not blocking for π_1, as $(0 : ab)$ is not a blocking factor for the portal $q_1, 1 \rightsquigarrow q_1, 1$. It is, however, a blocking sequence for π_2. This is because if we enter the SCC $\{q_1, q_2\}$ through q_1, a factor ab can only appear after an even number of steps, while if we enter through q_2, it can only appear after an odd number of steps.

4.2 An efficient property tester

In this section, we show that for any regular language L and any small enough $\varepsilon > 0$, there is an ε-property tester for L that uses $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries.

▶ **Theorem 4.15.** *For any NFA $\mathcal{A}$ and any small enough $\varepsilon > 0$, there exists an ε-property tester for $\mathcal{L}(\mathcal{A})$ that uses $\mathcal{O}(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

As mentioned in the overview, this result supersedes the one given by Bathie and Starikovskaya [8]: while both testers use the same number of queries, the tester in [8] works under the edit distance, while that of Theorem 4.15 is designed for the Hamming distance. As the edit distance never exceeds the Hamming distance, the set of words that are ε-far with respect to the former is contained in the set of words ε-far for the latter. Therefore, an ε-tester for the Hamming distance is also an ε-tester for the edit distance.

The property tester behind Theorem 4.15 uses the property tester for strongly connected NFAs as a subroutine, and its correctness is based on an extension of Lemma 3.9 to blocking sequences. We show that we can reduce property testing of $\mathcal{L}(\mathcal{A})$ to a search for blocking sequences in the word, in the following sense:

- If μ contains a strongly blocking sequence for each of the SCC-paths of $\mathcal{A}$, then it is not in the language and we can answer no (Corollary 4.16).
- If μ is ε-far from the language, then for each accepting SCC-path π of $\mathcal{A}$, μ is far from for the language of π and contains many disjoint strongly blocking sequences for π (Lemma 4.17), hence random sampling is likely to find at least one of them, and we reject μ with constant probability.

▶ **Corollary 4.16.** *If μ contains a strongly blocking sequence for each SCC-path of $\mathcal{A}$, then $\mu \notin \mathcal{TL}(\mathcal{A})$.*

Proof. This follows from Lemma 4.9. ◀

The next lemma expresses a partial converse to Corollary 4.16 and generalizes Lemma 3.8 from the strongly connected case: if a word is far from the language, then it contains many strongly blocking sequences for any SCC-path.

▶ **Lemma 4.17.** *Let $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ be an SCC-path, let $L = \mathcal{L}(\pi)$, and let μ be a positional word of length n such that $d(\mu, L)$ is finite. There is a constant C such that if $n \geq C/\varepsilon$ and μ is ε-far from L, then μ can be partitioned into $\mu = \mu_0\mu_1 \cdots \mu_k$ such that for every $i = 0, \ldots, k$, μ_i contains at least $\frac{\varepsilon n}{C}$ disjoint blocking factors for P_i, each of length at most $\mathcal{O}(1/\varepsilon)$.*

▶ **Lemma 4.18.** *Let $L = \mathcal{TL}(\mathcal{A})$ and let μ be a positional word of length n. If L contains a word of length n and μ is ε-far from L, then μ contains $\Omega(\varepsilon n)$ disjoint blocking sequences for $\mathcal{A}$.*

4.3 Lower bound

In order to characterize hard languages for all automata, we define a partial order $\trianglelefteq$ on sequences of positional factors. It is an extension of the factor partial order on blocking factors. It will let us define *minimal blocking sequences*, which we use to characterize the complexity of testing a language.

▶ **Definition 4.19** (Minimal blocking sequence)**.** *Let $\sigma = (\mu_1, \mu_2, \ldots, \mu_k)$ and $\sigma' = (\mu'_1, \ldots, \mu'_t)$ be sequences of positional words. We have $\sigma \trianglelefteq \sigma'$ if there exists a sequence of indices $i_1 \leq i_2 \leq \ldots \leq i_k$ such that μ_j is a factor of μ'_{i_j} for all $j = 1, \ldots, k$.*

A blocking sequence σ of $\mathcal{A}$ (resp. π) is minimal *if it is a minimal element of $\trianglelefteq$ among blocking sequences of $\mathcal{A}$ (resp. π). The set of minimal blocking sequences of $\mathcal{A}$ (resp. π) is written $\mathsf{MBS}(\mathcal{A})$ (resp. $\mathsf{MBS}(\pi)$).*

▶ Remark 4.20. If $\sigma \trianglelefteq \sigma'$ and σ is a blocking sequence for an SCC-path π then σ' is also a blocking sequence for π.

To prove a lower bound on the number of queries necessary to test a language when $\mathsf{MBS}(\mathcal{A})$ is infinite, we present a reduction to the strongly connected case. Under the assumption that $\mathcal{A}$ has infinitely many minimal blocking sequences, we exhibit a portal P of $\mathcal{A}$ with infinitely many minimal blocking factors and "isolate it" by constructing two sequences of positional factors σ_l and σ_r such that for all μ, $\sigma_l, (\mu), \sigma_r$ is blocking for $\mathcal{A}$ if and only if μ is a blocking factor of P. Then we reduce the problem of testing the language of this portal to the problem of testing the language of P.

To define "isolating P" formally, we define the left (and right) effect of a sequence on an SCC-path. Informally, the left effect of a sequence σ on an SCC-path π is related to the index of the first portal in π where a run can be after reading σ, because all previous portals

have been blocked. The right effect represents the same in reverse, starting from the end of the run. More formally, the *left effect* of a sequence σ on an SCC-path $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ is the largest index i such that the sequence is blocking for $P_0 \xrightarrow{a_1} \cdots P_i$ (-1 if there is no such i). We denote it by $(\sigma \gg \pi)$. Similarly, the *right effect* of a sequence on π is the smallest index i such that the sequence is blocking for $P_i \xrightarrow{a_{i+1}} \cdots P_k$ ($k+1$ if there is no such i); we denote it by $(\pi \ll \sigma)$.

▶ Remark 4.21. A sequence σ is blocking for an SCC-path $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ if and only if $(\sigma \gg \pi) = k$, if and only if $(\pi \ll \sigma) = 0$.

Also, given two sequences σ_l, σ_r, the sequence $\sigma_l\sigma_r$ is blocking for π if and only if $(\sigma_l \gg \pi) \geq (\pi \ll \sigma_r)$.

For the next lemma we define a partial order on portals: $P \preceq P'$ if all blocking factors of P' are also blocking factors of P. We write $\succeq$ for the reverse relation, $\simeq$ for the equivalence relation $\preceq \cap \succeq$ and $\not\simeq$ for the complement relation of $\simeq$.

Additionally, given an SCC-path $\pi = P_0 \xrightarrow{x_1} \dots P_k$ and two sequences of positional words σ_l, σ_r, we say that the portal P_i *survives* (σ_l, σ_r) *in* π if $(\sigma_l \gg \pi) < i < (\pi \ll \sigma_r)$.

▶ **Definition 4.22.** *Let P be a portal and σ_l and σ_r sequences of positional words.*
We define three properties that those objects may have:

P1) *$\sigma_l\sigma_r$ is not blocking for $\mathcal{A}$*

P2) *P has infinitely many minimal blocking factors*

P3) *for any accepting SCC-path π in $\mathcal{A}$, every portal in π which survives (σ_l, σ_r) is $\simeq$-equivalent to P.*

▶ **Lemma 4.23.** *If $\mathcal{A}$ has infinitely many minimal blocking sequences, then there exist a portal P and sequences σ_l and σ_r satisfying properties P1, P2 and P3.*

▶ **Lemma 4.24.** *If there exist a portal P and σ_l, σ_r satisfying properties P1, P2 and P3 then $\mathcal{L}(\mathcal{A})$ is hard.*

▶ **Proposition 4.25.** *If $\mathcal{A}$ has infinitely many minimal blocking sequences, then $\mathcal{L}(\mathcal{A})$ is hard.*

Proof. We combine Lemmas 4.23 and 4.24. ◀

5 Trivial and Easy languages

5.1 Upper bound for easy languages

We first establish that an automaton with finitely many minimal blocking sequences is easy (or trivial) to test, using a similar algorithm as in Theorem 4.15.

▶ **Lemma 5.1.** *Let $\mathcal{A}$ be an NFA with a finite number of minimal blocking sequences, let $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ be an SCC-path of $\mathcal{A}$, let $L = \mathcal{L}(\pi)$, and let μ be a positional word of length n such that $d(\mu, L)$ is finite. There are constants B, D such that if $n \geq 2D/\varepsilon$ and μ is ε-far from L, then μ can be partitioned into $\mu = \tau_0\tau_1 \cdots \tau_k$ such that for every $i = 0, \dots, k$, τ_i contains at least $\frac{\varepsilon n}{D}$ disjoint blocking factors for P_i, each of length at most B.*

▶ **Proposition 5.2.** *If $\mathcal{A}$ has finitely many minimal blocking sequences, then there is a tester for $\mathcal{L}(\mathcal{A})$ that uses $\mathcal{O}(1/\varepsilon)$ queries.*

Proof. We use the same algorithm that for Theorem 4.15, except that we use the factors given by Lemma 5.1, therefore, in the call to the SAMPLER function (Algorithm 1), the upper bound on the length of the factors is B instead of $\mathcal{O}(1/\varepsilon)$. In that case, the query complexity becomes $\mathcal{O}(\log(B)/\varepsilon) = \mathcal{O}(1/\varepsilon)$. ◀

This already gives us a clear dichotomy: all languages either require $\Theta(\log(\varepsilon^{-1})/\varepsilon)$ queries to be tested, or can be tested with $\mathcal{O}(1/\varepsilon)$ queries.

5.2 Separation between trivial and easy languages

It remains to show that languages that can be tested with $\mathcal{O}(1/\varepsilon)$ queries have query complexity either $\Theta(1/\varepsilon)$, or 0 for large enough n. Our proof uses the class of *trivial* regular languages identified by Alon et al. [4], which we revisit next.

An example of a trivial language is L_2 consisting of words containing at least one a over the alphabet $\{a, b\}$. For any word u, replacing any letter by a yields a word in L_2, hence $d(u, L_2) \leq 1$. Therefore, for $n > 1/\varepsilon$, no word of length n is ε-far from L_2, and the trivial property tester that answers "yes" without sampling any letter is correct.

Alon et al. [4] define non-trivial languages as follows.

▶ **Definition 5.3** ([4, Definition 3.1])**.** *A language L is non-trivial if there exists a constant $\varepsilon_0 > 0$, so that for infinitely many values of n the set $L \cap \Sigma^n$ is non-empty, and there exists a word $w \in \Sigma^n$ so that $d(w, L) \geq \varepsilon_0 n$.*

It is easy to see that if a language is trivial in the above sense (i.e. not non-trivial), then for large enough input length n, the answer to testing membership in L only depends n, and the algorithm does not need to query the input. Alon et al. [4, Property 2] show that if a language is non-trivial, then testing it requires $\Omega(1/\varepsilon)$ queries for small enough $\varepsilon > 0$.

To obtain our characterization of trivial languages, we show that $\mathsf{MBS}(\mathcal{A})$ is non-empty if and only if $\mathcal{L}(\mathcal{A})$ is non-trivial (in the above sense). It follows that if $\mathsf{MBS}(\mathcal{A})$ is empty, then testing $\mathcal{L}(\mathcal{A})$ requires 0 queries for large enough n. Furthermore, by the result of Alon et al. [4], if $\mathsf{MBS}(\mathcal{A})$ is non-empty, then testing $\mathcal{L}(\mathcal{A})$ requires $\Omega(1/\varepsilon)$ queries.

Recall that we focus on infinite languages, since we know that all finite ones are trivial (Remark 1.4).

▶ **Lemma 5.4.** *$\mathsf{MBS}(\mathcal{A})$ is empty if and only if $L = \mathcal{L}(\mathcal{A})$ is trivial.*

It is easy to see that if a language is trivial in the above sense, then for large enough input length n, membership in L only depends n, and the algorithm does not need to query the input. Alon et al. [4] show that if a language is non-trivial, then testing it requires $\Omega(1/\varepsilon)$ queries for small enough $\varepsilon > 0$. As a corollary of that lower bound, we obtain that if $\mathsf{MBS}(\mathcal{A})$ is non-empty, then testing $\mathcal{L}(\mathcal{A})$ requires $\Omega(1/\varepsilon)$ queries.

6 Hardness of classifying

In the previous sections, we have shown that testing some regular languages (*easy* ones) that requires fewer queries than testing others (*hard* ones). Therefore, given the task of testing a word for membership in $\mathcal{L}(\mathcal{A})$, it is natural to first try to determine if the language of $\mathcal{A}$ is easy, and if this is the case, run the appropriate ε-tester, that uses fewer queries. In this section, we investigate the computational complexity of checking which class of the trichotomy the language of a given automaton belongs to. We formalize this question as the following decision problems:

▶ **Problem 6.1** (Triviality problem). Given an finite automaton $\mathcal{A}$, is $\mathcal{L}(\mathcal{A})$ trivial?

▶ **Problem 6.2** (Easiness problem). Given an finite automaton $\mathcal{A}$, is $\mathcal{L}(\mathcal{A})$ easy?

▶ **Problem 6.3** (Hardness problem). Given an finite automaton $\mathcal{A}$, is $\mathcal{L}(\mathcal{A})$ hard?

In these problems, the automaton $\mathcal{A}$ is the input and is no longer fixed. We show that, our combinatorial characterization based on minimal blocking sequences is effective, in the sense that all three problems are decidable. However, it does not lead to efficient algorithms, as both problems are PSPACE-complete.

▶ **Theorem 6.4.** *The triviality and easiness problems are both PSPACE-complete, even for strongly connected NFAs.*

7 Conclusion

We presented an effective classification of regular languages in three classes, each associated with an optimal query complexity for property testing. We thus close a line of research aiming to determine the optimal complexity of regular languages. All our results are with respect to the Hamming distance. We conjecture that they can be adapted to the edit distance. We use non-deterministic automata to represent regular languages. A natural open question is the complexity of classifying languages represented by *deterministic* automata.

References

1 Maryam Aliakbarpour, Ilias Diakonikolas, and Ronitt Rubinfeld. Differentially private identity and equivalence testing of discrete distributions. In *International Conference on Machine Learning, ICML*, 2018. URL: https://proceedings.mlr.press/v80/aliakbarpour18a.html.

2 Noga Alon, Richard A. Duke, Hanno Lefmann, Vojtech Rödl, and Raphael Yuster. The algorithmic aspects of the regularity lemma. *Journal of Algorithms*, 16(1):80–109, 1994. doi:10.1006/JAGM.1994.1005.

3 Noga Alon, Eldar Fischer, Michael Krivelevich, and Mario Szegedy. Efficient testing of large graphs. *Combinatorica*, 20(4):451–476, 2000. doi:10.1007/s004930070001.

4 Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing*, 30(6):1842–1862, 2001. doi:10.1109/SFFCS.1999.814641.

5 Noga Alon and Asaf Shapira. Every monotone graph property is testable. *SIAM Journal of Computing*, 38(2):505–522, 2008. doi:10.1137/050633445.

6 Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic membership for regular languages. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 116:1–116:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.116.

7 Ajesh Babu, Nutan Limaye, Jaikumar Radhakrishnan, and Girish Varma. Streaming algorithms for language recognition problems. *Theoretical Computer Science*, 494:13–23, 2013. doi:10.1016/J.TCS.2012.12.028.

8 Gabriel Bathie and Tatiana Starikovskaya. Property testing of regular languages with applications to streaming property testing of visibly pushdown languages. In *International Colloquium on Automata, Languages, and Programming, ICALP*, 2021. doi:10.4230/LIPIcs.ICALP.2021.119.

9 Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993. doi:10.1016/0022-0000(93)90044-W.

10 Ilias Diakonikolas and Daniel M Kane. A new approach for testing properties of discrete distributions. In *IEEE Symposium on Foundations of Computer Science, FOCS*, pages 685–694. IEEE, 2016. doi:10.1109/FOCS.2016.78.

11 Yevgeniy Dodis, Oded Goldreich, Eric Lehman, Sofya Raskhodnikova, Dana Ron, and Alex Samorodnitsky. Improved testing algorithms for monotonicity. In *International Workshop on Randomization and Approximation Techniques in Computer Science, RANDOM*, pages 97–108. Springer, 1999. doi:10.1007/978-3-540-48413-4_10.

12 Funda Ergün, Sampath Kannan, S.Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. *Journal of Computer and System Sciences*, 60(3):717–751, 2000. doi:10.1006/jcss.1999.1692.

13 Eldar Fischer, Frédéric Magniez, and Tatiana Starikovskaya. Improved bounds for testing Dyck languages. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1529–1544. SIAM, 2018. doi:10.1137/1.9781611975031.100.

14 Nathanaël François, Frédéric Magniez, Michel de Rougemont, and Olivier Serre. Streaming property testing of visibly pushdown languages. In *Proceedings of the 24th Annual European Symposium on Algorithms*, volume 57 of *LIPIcs*, pages 43:1–43:17, 2016. doi:10.4230/LIPIcs.ESA.2016.43.

15 Moses Ganardi, Danny Hucke, and Markus Lohrey. Randomized sliding window algorithms for regular languages. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming*, volume 107 of *LIPIcs*, pages 127:1–127:13, 2018. doi:10.4230/LIPIcs.ICALP.2018.127.

16 Moses Ganardi, Danny Hucke, Markus Lohrey, and Tatiana Starikovskaya. Sliding Window Property Testing for Regular Languages. In Pinyan Lu and Guochuan Zhang, editors, *30th International Symposium on Algorithms and Computation (ISAAC 2019)*, volume 149 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ISAAC.2019.6.

17 Oded Goldreich. *Introduction to property testing.* Cambridge University Press, 2017. doi:10.1017/9781108135252.

18 Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45(4):653–750, July 1998. doi:10.1145/285055.285060.

19 Rahul Jain and Ashwin Nayak. The space complexity of recognizing well-parenthesized expressions in the streaming model: The index function revisited. *IEEE Transactions on Information Theory*, 60(10):6646–6668, October 2014. doi:10.1109/TIT.2014.2339859.

20 Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. In *Proc. of MFCS 2011*, volume 6907 of *LNCS*, pages 412–423. Springer, 2011. doi:10.1007/978-3-642-22993-0_38.

21 Frédéric Magniez and Michel de Rougemont. Property testing of regular tree languages. *Algorithmica*, 49(2):127–146, 2007. doi:10.1007/s00453-007-9028-3.

22 Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM J. Comput.*, 43(6):1880–1905, 2014. doi:10.1137/130926122.

23 Liam Paninski. A coincidence-based test for uniformity given very sparsely sampled discrete data. *IEEE Transactions on Information Theory*, 54(10):4750–4755, 2008. doi:10.1109/TIT.2008.928987.

24 Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing membership in parenthesis languages. *Random Struct. Algorithms*, 22(1):98–138, 2003. doi:10.1002/rsa.10067.

25 Jean-Éric Pin, editor. *Handbook of Automata Theory.* European Mathematical Society Publishing House, Zürich, Switzerland, 2021. doi:10.4171/AUTOMATA.

26 Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing*, 25(2):252–271, 1996. doi:10.1137/S0097539793255151.

27 Mosaad Al Thokair, Minjian Zhang, Umang Mathur, and Mahesh Viswanathan. Dynamic race detection with O(1) samples. *Proc. ACM Program. Lang.*, 7(POPL):1308–1337, 2023. doi:10.1145/3571238.

28 Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *Symposium on Foundations of Computer Science, SFCS*, pages 222–227. IEEE, 1977. doi:10.1109/SFCS.1977.24.