Constant-Space, Constant-Randomness Verifiers with Arbitrarily Small Error✩

M. Utkan Gezer*, A. C. Cem Say

Boğaziçi University, Bebek, İstanbul, Turkey

Abstract

We study the capabilities of probabilistic finite-state machines that act as verifiers for certificates of language membership for input strings, in the regime where the verifiers are restricted to toss some fixed nonzero number of coins regardless of the input size. Say and Yakarıyılmaz showed that the class of languages that could be verified by these machines within an error bound strictly less than $1/2$ is precisely NL, but their construction yields verifiers with error bounds that are very close to $1/2$ for most languages in that class. We characterize a subset of NL for which verification with arbitrarily low error is possible by these extremely weak machines. It turns out that, for any $\varepsilon > 0$, one can construct a constant-coin, constant-space verifier operating within error $\varepsilon$ for every language that is recognizable by a linear-time multi-head finite automaton ($2\text{nf}_a(k)$). We discuss why it is difficult to generalize this method to all of NL, and give a reasonably tight way to relate the power of linear-time $2\text{nf}_a(k)$’s to simultaneous time-space complexity classes defined in terms of Turing machines.

Keywords: Interactive Proof Systems, Multi-head finite automata, Probabilistic finite automata

1. Introduction

The classification of languages in terms of the resources required for verifying proofs (“certificates”) of membership in them is a main concern of computational complexity theory. Major results in this area have demonstrated important tradeoffs among different types of resources such as time, space, and randomness: The power of deterministic polynomial-time, polynomial-space bounded verifiers, characterized by the class NP, has, for instance, been shown to be identical to that of probabilistic bounded-error polynomial-time logarithmic-space verifiers that toss only logarithmically many coins in terms of the input

✩This paper is a substantially improved version of [1].

*Corresponding author

Email addresses: utkan.gezer@boun.edu.tr (M. Utkan Gezer), say@boun.edu.tr (A. C. Cem Say)
size \[2\]. More recently, Say and Yakaryılmaz initiated the study of the power of finite-state verifiers that are restricted to toss some fixed nonzero number of coins regardless of the input size, and proved \[3\] that the class of languages that could be verified by these machines within an error bound strictly less than \(1/2\) is precisely NL, i.e. languages with deterministic logarithmic-space verifiers.

The construction given in \[3\] could exhibit a constant-randomness verifier operating within error \(\varepsilon\) for some \(\varepsilon < 1/2\) for any language in NL, however, it did not provide a method for reducing this error to more desirable smaller values. Indeed, for many languages in NL, the constructed verifier’s error bound is uncomfortably close to \(1/2\), raising the question of whether the class of languages for which it is possible to obtain verifiers with arbitrarily small positive error bounds is a proper subset of NL or not.

In this paper, we characterize a subset of NL for which verification with arbitrarily low error is possible by these extremely weak machines. It turns out that, for any \(\varepsilon > 0\), one can construct a constant-coin, constant-space verifier operating within error \(\varepsilon\) for every language that is recognizable by a linear-time multi-head finite automaton (\(2\text{nfa}(k)\)). We discuss why it is difficult to generalize this method to all of NL, and give a reasonably tight way to relate the power of linear-time \(2\text{nfa}(k)\)'s to simultaneous time-space complexity classes defined in terms of Turing machines. We conclude with a list of open questions.

2. Preliminaries

The reader is assumed to be familiar with the standard concepts of automata theory, Turing machines, and basic complexity classes \[4\].

The following notation will be used throughout this paper:

- \(\sigma_i\) is the \(i^{th}\) element of the sequence \(\sigma\)
- \(\sigma\tau\) is the sequences \(\sigma\) and \(\tau\) concatenated
- \(\langle O_1, \ldots, O_k \rangle\) is the encoding of objects \(O_i\) in the alphabet of context
- \(A \sqcup B\) is the union of sets \(A\) and \(B\), also asserting that \(A\) and \(B\) are disjoint

2.1. Multihead finite automata

A \(k\)-head nondeterministic finite automaton, denoted \(2\text{nfa}(k)\), is a 6-tuple consisting of

1. a finite set of states \(Q\),
2. an input alphabet \(\Sigma\),
3. a transition function \(\delta: Q \times \Gamma^k \rightarrow \mathcal{P}(Q \times \Delta^k)\), where:
   - \(\Gamma = \Sigma \sqcup \{\rhd, \ldr\}\) is the tape alphabet, where \(\rhd\) and \(\ldr\) are respectively the left and right end markers

\footnote{\textsuperscript{1}The 2 in \(2\text{nfa}(k)\) is to indicate that they can move their heads in both directions.}
• \( \Delta = \{-1, 0, 1\} \) is the set of head movements, where \(-1\) and \(1\) respectively indicate moving left and right, and \(0\) indicates staying put,

4. an initial state \( q_0 \),
5. an accept state \( q_{\text{accept}} \), and
6. a reject state \( q_{\text{reject}} \).

A 2nfa\((k)\) \(M = (Q, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\) initially starts from \(q_0\), and with \(\triangleright x \triangleleft\) written on its single read-only tape, where \(x \in \Sigma^*\) is the input string. All \(k\) tape heads are initially on the \(\triangleright\) symbol. The function \(\delta\) maps the current state and the \(k\) symbols under the tape heads to a set of alternative steps \(M\) can take. By picking an alternative \((q, d)\), \(M\) transitions into the state \(q\), and moves its \(i^{th}\) head by \(d_i\).

The configuration of a 2nfa\((k)\) \(M\) at a step of its execution is the \((k+1)\)-tuple consisting of its state and its head positions at that moment. The initial configuration of \(M\) is \((q_0, 0^k)\).

Starting from its initial configuration, and following different alternatives offered by \(\delta\), a 2nfa\((k)\) \(M\) may have several computational paths on the same string. A computational path of \(M\) halts if it reaches \(q_{\text{accept}}\) or \(q_{\text{reject}}\), or if \(\delta\) does not offer any steps for \(M\) to follow. \(M\) accepts an input string \(x\), if there is a computational path of \(M\) running on \(x\) that halts on \(q_{\text{accept}}\). \(M\) rejects an input string \(x\), if \(M\) running on \(x\) halts on a state other than \(q_{\text{accept}}\) on every computational path. The language recognized by \(M\) is the set of all strings accepted by \(M\).

Given an input string \(x\), \(M\) may have computational paths that never halt. In the special case that \(M\) given any input string halts on every computational path, \(M\) is said to be an always halting 2nfa\((k)\).

For a particular \(k > 0\), let \(\mathcal{L}(\text{2nfa}(k))\) denote the class of languages recognized by a 2nfa\((k)\). For when \(k\) is unspecified, let \(\mathcal{L}(\text{2nfa}(k))\) denote the class of languages that has a 2nfa\((k)\) recognizing them for some \(k\). In other words;

\[
\mathcal{L}(\text{2nfa}(k)) = \bigcup_{k > 0} \mathcal{L}(\text{2nfa}(k))
\]

\(\mathcal{L}(\text{2nfa}(1))\) is the class of regular languages.

For any growth function \(f(n)\), NSPACE\((f(n))\) denotes the class of languages recognized by nondeterministic Turing machines which are allowed to use \(O(f(n))\) space for inputs of length \(n\). The class NSPACE\((\log n)\) is commonly denoted as NL.

**Lemma 1.** Nondeterministic multi-head finite automata are equivalent to nondeterministic logarithmic space TMs in terms of language recognition power [2]. Put formally:

\[
\mathcal{L}(\text{2nfa}(k)) = \text{NL}
\]
Lemma 2. The languages in NL are organized in a hierarchy based on the number of heads of the nondeterministic automata recognizing them. Formally, the following is true for any $k > 0$:

$$L(2nfa(k)) \subset L(2nfa(k + 1))$$

For any given $k$, let $L(2nfa(k), f(n))$ denote the class of languages that are recognized by a $2nfa(k)$ running for $O(f(n))$ steps on every alternative computational path on any input of length $n$. Clearly, those machines are also always halting. Let $L(2nfa(k), f(n))$ denote the class of languages that are recognized by a nondeterministic multi-head finite automata with any number of heads, and running in $O(f(n))$ time. We use linear-time designation instead of $f(n) = n$.

Lemma 3. The following is true for any $k > 0$:

$$L(2nfa(k)) \subset L(2nfa(2k), n^k)$$

Proof. Let $M$ be any $2nfa(k)$ recognizing $A$ with $Q$ as its set of states. Running on an input string of length $n$, $M$ can have $T = |Q| \cdot (n + 2)^k$ different configurations. If $M$ executes for more than $T$ steps, then it must have repeated a configuration, and be in a loop. Therefore, for every input string in $A$, $M$ should have an accepting computation path of at most $T$ steps.

With the help of $k$ additional counter heads, the $2nfa(2k)$ $M'$ can simulate $M$ while imposing it a runtime limit of $T$ steps. Machine $M'$ can count up to $T$ as follows: Let $c_1, \ldots, c_k$ denote the counter heads. Head $c_1$ moves right every $|Q|^{1/\delta}$ step of $M$’s simulation. For $i < k$, whenever the head $c_i$ reaches the right end marker, it rewinds back to the left end, and head $c_{i+1}$ moves once to right. If $c_k$ attempts to move past the right end, $M'$ rejects.

The $2nfa(2k)$ $M'$ recognizes the same language as $M$, but within the time limit of $O(n^k)$. $\square$

Lemmas 2 and 3 can be combined into the following useful fact.

Corollary 4. For every $A \in \text{NL}$, there is a minimum number $k_A$, such that there exists an always halting $2nfa(k_A)$ recognizing $A$, but not an always halting $2nfa(h)$ with $h < k_A$.

Proof. By Lemma 2, $A \in L(2nfa(k))$, but $A \notin L(2nfa(k - 1))$ for some $k$. The existence of an always halting $2nfa(k_A)$ recognizing $A$ for a minimum $k_A$ between $k$ and $2k$ is guaranteed by Lemmas 2 and 3 respectively. $\square$

Lemma 5. $\text{HALTING}_{2nfa} = \{ \langle M \rangle | M \text{ is an always halting } 2nfa \}$ is decidable.

Proof. The two-way alternating finite automaton, denoted $2afa$, is a generalization of the $2nfa$ model. The state set of a $2afa$ is partitioned into universal

\footnote{We thank Neal E. Young, who introduced us the algorithm for this proof.}
and existential states. A 2afa accepts a string $x$, if and only if starting from the initial state, every alternative transition from the universal states, and at least one of the alternative transitions from the existential states leads to acceptance. Thus, a 2nfa is a 2afa with only existential states.

A one-way finite automaton, denoted 1dfa, is a 2dfa that cannot move its head to left. A 1nfa is a nondeterministic 1dfa.

Consider the following algorithm to recognize $HALTING_{2nfa}$:

$D = \text{“On input } \langle M \rangle, \text{ where } M \text{ is an } 2nfa, \text{ and } \Sigma \text{ is its alphabet:}$$

1. Construct a 2afa $M'_{2afa}$ by modifying $M$ to accept whenever it halts and designating every state as universal.
2. Convert $M'_{2afa}$ to an equivalent 1dfa $M'_{1dfa}$.
3. Check whether $M'_{1dfa}$ recognizes $\Sigma^*$. If it does, accept. Otherwise, reject.”

By its construction, $M'_{2afa}$ recognizes $\Sigma^*$ if and only if $M$ halts in every computation path, running on every possible input string, i.e. it is always halting. Stage 2 can be implemented by the algorithms given in [7] and [4]. The final check in stage 3 also known as the universality problem, has a well-known algorithm. So the algorithm $D$ decides whether a given 2nfa $M$ is always halting.

2.2. Probabilistic Turing machines and finite automata

A probabilistic Turing machine (PTM) is a Turing machine equipped with a randomization device. In its designated coin-tossing states, a PTM obtains a random bit using the device, and proceeds by its value. The language of a PTM is the set of strings that it accepts with a probability greater than $1/2$.

A probabilistic finite automaton (2pfa) is a restricted PTM with a single read-only tape. This model can also be viewed as an extension of a 2dfa with designated coin-tossing states. A 2pfa tosses a hypothetical coin whenever it is in one of those states, and proceeds by its random outcome. Formally, a 2pfa consists of the following:

1. a finite set of states $Q = Q_d \cup Q_r$, where:
   • $Q_d$ is the set of deterministic states\(^3\) and
   • $Q_r$ is the set of coin-tossing states\(^3\)
2. an input alphabet $\Sigma$,
3. a transition function overloaded as deterministic $\delta_d$ and coin-tossing $\delta_r$:
   • $\delta_d : Q_d \times \Gamma \to Q \times \Delta$, where $\Gamma$ and $\Delta$ are as defined for the 2nfa($k$)’s, and
   • $\delta_r : Q_r \times \Gamma \times R \to Q \times \Delta$, where $R = \{0, 1\}$ is a random bit provided by a “coin toss”\(^3\).

\(^3\)The letters $d$ and $r$ stand for deterministic and random, respectively.
4. an initial state $q_0$,
5. an accept state $q_{\text{accept}}$, and
6. a reject state $q_{\text{reject}}$.

The language of a 2pfa is similarly the set of strings which are accepted with a probability greater than $\frac{1}{2}$.

Due to its probabilistic nature, a PTM may occasionally err, and disagree with its language. In this paper, we will be concerned about the following types of error:

1. *Failing to accept* – rejecting or looping indefinitely given a member input
2. *Failing to reject* – accepting or looping indefinitely given a non-member input

2.3. Interactive proof systems

Our definitions of interactive proof systems (IPSecs) are based on [8]. We will only be interested with a single variant, namely, the private-coin one-way IPS.

An IPS consists of a verifier and a prover. The verifier is a PTM vested with the task of recognizing an input string’s membership, and the prover is a function providing the purported proof of membership.

In private-coin one-way IPS, the prover $P$ can be viewed as a certificate function $c: \Sigma^* \rightarrow \Lambda^\infty$ that maps input strings to infinitely long certificates, where $\Sigma$ and $\Lambda$ are respectively the input and certificate alphabets. The verifier $V$, in turn, can be thought of as having an additional certificate tape (with a head that cannot move left) to read from. Given an input string $x \in \Sigma^*$, $V$ executes on it as usual, and with $c(x)$ written on its certificate tape.

In this paper, the term “PTM verifier in a private-coin one-way IPS” will be abbreviated as “PTM verifier”.

The language $A$ of PTM verifier $V$ is the set of strings that $V$, paired with some $c(x)$, accepts with a probability greater than $\frac{1}{2}$. The error bound$^4$ of $V$, denoted $\varepsilon_V$, is then defined as the minimum value satisfying both of the following, given any such input string $x$:

- $V$, paired with some $c(x)$, accepts $x \in A$ with a probability at least $1 - \varepsilon_V$.
- $V$, paired with any $c(x)$, rejects $x \notin A$ with a probability at least $1 - \varepsilon_V$.

Let $1\text{IP}_\varepsilon(t(n),s(n),r(n))$ be the class of languages that have verifiers with an error at most $\varepsilon$ ($\varepsilon < 1/2$) using $O(s(n))$ space, and $O(r(n))$ amount of coins in the worst case, and with an expected runtime in $O(t(n))$, where $n$ denotes the length of the input string. Instead of a function of $n$, and when appropriate, we write simply $\text{cons}$, $\log$, $\text{poly}$, or $\text{exp}$ to describe a constant, logarithmic, polynomial, or exponential limit, respectively, in terms of the input length.

$^4$Our definition of the error bound corresponds to the “strong” version of the IPS definition in [8].
We write $0$ and $\infty$ to describe that a resource is unavailable and unlimited, respectively. Furthermore, let:

$$1\text{IP}(t(n), s(n), r(n)) = \bigcup_{\varepsilon < \frac{1}{2}} 1\text{IP}_{\varepsilon}(t(n), s(n), r(n))$$

$$1\text{IP}_{\ast}(t(n), s(n), r(n)) = \bigcap_{\varepsilon > 0} 1\text{IP}_{\varepsilon}(t(n), s(n), r(n))$$

We note the following known results: $[9, 2, 3]$

$$\text{NP} = 1\text{IP}(\text{poly, poly, 0}) = 1\text{IP}(\text{poly, log, log}) = 1\text{IP}(\text{poly, log, poly})$$

$$\text{NL} = 1\text{IP}(\text{poly, log, 0}) = 1\text{IP}(\infty, \text{cons, cons})$$

For polynomial-time verifiers with the ability to use at least logarithmic space, the class $1\text{IP}_{\ast}(t(n), s(n), r(n))$ is identical to the corresponding class $1\text{IP}(t(n), s(n), r(n))$, since such an amount of memory can be used to time one’s own execution and reject computations that exceed the time limit, enabling the verifier to run through several consecutively appended copies of certificates for the same string, and deciding according to the majority of the results of the individual controls. For constant-space verifiers, this procedure is not possible, and the question of whether $1\text{IP}_{\ast}(\infty, \text{cons, cons})$ equals $1\text{IP}(\infty, \text{cons, cons})$ is nontrivial, as we will examine in the following sections.

3. Linear-time 2nfa($k$)’s and verification with small error

In $[3]$, Say and Yakaryılmaz showed that membership in any language in NL may be checked by a 2pfa verifier using some constant number of random bits. They also showed how the weak error of the verifier can be made arbitrarily small.

We will now describe their approach, which forms the basis of our own work.

The method, which we will name $\mu_1$, for producing a constant-randomness 2pfa verifier, given any language $A \in \text{NL}$, takes an always halting 2nfa($k$) $M_A$ recognizing $A$ (for some $k$), which exists by Lemmas $\text{[11]}$ and $\text{[13]}$ as its starting point. The constructed verifier $\mu_1(A)$ will attempt to simulate $M_A$, relying on the certificate and its private coins to compensate for the fact that it has $k - 1$ fewer input heads than $M_A$. Given any input string $x$, $\mu_1(A)$ expects a certificate $c(x)$ to provide the following information for each transition of $M_A$ en route to purported acceptance: the symbols read by the $k$ heads, and the nondeterministic branch taken. $\mu_1(A)$ tracks the described computational path of $M_A$ according to $c(x)$, until either the path reaches a halting state, or $\mu_1(A)$ catches a “lie” in the certificate, in which case it rejects. If a nondeterministic

\[5\] In contrast to the (strong) error definition we use in this paper, the weak error definition (also by $\text{[3]}$) does not regard the verifier looping forever on an input which is not a member of the language as an error.
branching that \( c(x) \) reports turns out to be unavailable with the given readings, or the simulation arrives at the reject state, \( \mu_1(A) \) rejects. At the beginning of \( M_A \)'s simulation, \( \mu_1(A) \) chooses a head at random using \( \lceil \log k \rceil \) coins. Throughout the simulation, \( \mu_1(A) \) mimics the movements of this chosen head, verifying \( c(x) \)'s claims about what is being scanned by that head at any step, while leaving the claims about the remaining \( k - 1 \) heads unverified. If this simulation can repeated for \( m \) rounds, all of which end with the described computational path of \( M_A \) reaching acceptance without any lies being caught, \( \mu_1(A) \) finally accepts.

For any language in \( A \in \text{NL} \) which can be recognized by an always halting \( 2\text{nfa}(k) \) \( M_A \), the verifier of \( \mu_1 \) simulating \( M_A \) for \( m \) rounds uses a total of \( m \cdot \lceil \log k \rceil \) coins, which is a constant with respect to the input length.

Paired with the proper certificate \( c(x) \), \( \mu_1(A) \) accepts all strings \( x \in A \) with probability 1. As mentioned earlier, the “weak error” of \( \mu_1(A) \) therefore depends only on its worst-case probability of accepting some \( x \notin A \).

For \( x \notin A \), there does not exist an accepting computation of \( M_A \) on \( x \). Still, a certificate may describe a fictional computational path of \( M_A \) to acceptance, by reporting inaccurate values for the symbols read by at least one of the heads. Since \( \mu_1(A) \) can not check many of the actual readings, it may fail to notice those inaccuracies. However, since \( \mu_1(A) \) chooses a head to verify in random, there is a non-zero chance that \( \mu_1(A) \) detects any such lie.

The likelihood that \( \mu_1(A) \) chooses the same head to verify as the certificate is inaccurate about is at least \( \frac{1}{k} \). Therefore, the weak error of \( \mu_1(A) \) is at most \( \left( \frac{k-1}{k} \right)^m \). This upper bound for weak error can be made as close to 0 as one desires by increasing \( m \), the number of rounds to simulate.

Although the underlying \( 2\text{nfa}(k) \) \( M_A \) recognizing \( A \in \text{NL} \) is an always halting machine, the verifier \( \mu_1(A) \) can still be wound up in an infinite loop by some certificate: \( M_A \) might be relying on the joint effort of its many heads to ensure that it always halts. Since \( \mu_1(A) \) validates only a single head’s readings, inaccuracies on what others read may tamper this joint effort, and lead \( \mu_1(A) \) into a loop. A malicious certificate might lead \( \mu_1(A) \) in a loop due to being inaccurate about one head alone. This might happen during the first round, and then, there would not be any more rounds for \( \mu_1(A) \), as it would be in a loop. The (strong) error \( \varepsilon_{\mu_1(A)} \) of \( \mu_1(A) \) is therefore at most \( \frac{(k-1)}{k} \). This upper bound to \( \varepsilon_{\mu_1(A)} \) cannot be reduced to less than \( \frac{(k_A-1)}{k_A} \), where \( k_A \) is the minimum number of heads required in an always halting machine to recognize \( A \), by Corollary 4.

Say and Yakarýlmaz also propose the method \( \mu_2 \), a slightly modified version of \( \mu_1 \) that produces verifiers with errors less than \( \frac{1}{2} \), albeit barely so. Let \( A \in \text{NL} \), and \( M_A \) be an always halting \( 2\text{nfa}(k) \) recognizing \( A \), for some \( k \). Regardless of the input string, the verifier \( \mu_2(A) \) rejects at the very beginning with a probability \( \frac{(k-1)}{2k} \), using \( \lceil \log k \rceil + 1 \) coins. Then, it continues just like

\(^6\)The error in the approximation \( k \approx 2^{\lceil \log k \rceil} \) used in this analysis does not affect the end result, and simplifies the explanation.
\(\mu_1(A)\). The bounds for the error \(\varepsilon_{\mu_2(A)}\) of \(\mu_2(A)\) are as follows:

\[
\frac{k - 1}{2k} \leq \varepsilon_{\mu_2(A)} \leq \frac{k^2 - 1}{2k^2}
\]

### 3.1. Safe and risky heads

How much of NL may yet fit into \(1IP^*_\infty(\infty, \text{cons, cons})\)? Method \(\mu_1\) was our starting point in working towards a lower bound for \(1IP^*_\infty(\infty, \text{cons, cons})\).

Let \(M_A\) be the 2nfa\((k)\) that \(\mu_1(A)\) uses to verify \(A \in NL\). The cause for \(\mu_1(A)\)'s high strong error turns out to be a decidable characteristic of \(M_A\)'s heads. We call such undependable heads risky heads.

**Definition 1 (Safe and risky heads).** Let \(M\) be a 2nfa\((k)\) with the transition function \(\delta: Q \times \Gamma^k \to \mathcal{P}(Q \times \Delta^k)\). For \(i\) between 1 and \(k\), let \(M_i\) be a 2nfa with the transition function \(\delta_i: Q \times \Gamma \to \mathcal{P}(Q \times \Delta)\) defined as follows:

\[
\delta_i(q, x) = \bigcup_{y \in \Gamma^k \atop y_i = x} \{ (r, d_i) \mid (r, d) \in \delta(q, y) \}
\]

If \(M_i\) is always halting, then the \(i^{th}\) head of \(M\) is a safe head. Otherwise, it is a risky head.

The execution of each 2nfa \(M_i\) in Definition 1 is designed to correspond to the \(i^{th}\)-head-only simulation of the 2nfa\((k)\) \(M\) by the verifier of \(\mu_1\). Just like the verifier of \(\mu_1\), \(M_i\) can make any of the transitions that \(M\)'s transition function allows, chooses one by the certificate, but making sure that the \(i^{th}\) symbol fed to \(M\)'s transition function is the same as the symbol it is reading itself. Crucially, if a certificate can wind the verifier of \(\mu_1\) into a loop during the one-headed simulation of \(M\), then the 2nfa \(M_i\) has a branch of computation that loops with an analogous certificate. The converse is also true. Therefore, the verifier of \(\mu_1\) can be wound up in a loop during a round of verification, if and only if it has chosen a risky head to verify.

**Lemma 6.** Being safe or risky is a decidable property of a 2nfa\((k)\)'s heads.

**Proof.** To decide whether the \(i^{th}\) head of a 2nfa\((k)\) \(M\) is safe, an algorithm can construct the 2nfa \(M_i\) described in Definition 1 and then test whether \(M_i \in \text{HALTING}_{2\text{nfa}}\) by the algorithm in Lemma 3.

Consider a language \(A \in NL\) that is recognized by a 2nfa\((k)\) \(M_A\) that always halts, and has safe heads only. The verifier \(\mu_1(A)\) using \(M_A\) cannot choose a risky head, and therefore can never loop. Thus, it verifies \(A\) with \(\varepsilon_{\mu_1(A)} \leq (k - 1)/k\).

### 3.1. Safe and risky heads

...
3.2. 2nfa\((k)\)’s with a safe head and small-error verification

The distinction of safe and risky heads has been the key to our improvement to the method \(\mu_1\). Method \(\mu_3\), to be introduced in the proof of the following lemma, is able to produce verifiers with an error bound equaling any desired non-zero constant, for a subset of languages in NL.

**Lemma 7.** Let \(A \in \text{NL}\). If there exists an always halting 2nfa\((k)\) with at least one safe head recognizing \(A\), then \(A \in \text{1IP}_* (\infty, \text{cons}, \text{cons})\).

**Proof idea.** The method \(\mu_3\) in the proof will construct verifiers similar to those of \(\mu_1\), except for a key difference. Given a language \(A \in \text{NL}\) recognized by an always halting 2nfa\((k)\) \(M_A\) that has at least one safe head, every head of \(M_A\) has essentially the same probability of getting chosen by \(\mu_1(A)\). In contrast, \(\mu_3(A)\) will be more likely to choose safe heads than the risky heads. Since \(\mu_3(A)\) cannot loop while tracking a safe head, one can reduce the probability of \(\mu_3(A)\) looping in any round to any non-zero constant \(P_R\) by increasing its bias towards the safe heads.

The (redeemable) disadvantage of \(\mu_3(A)\) for having a bias towards the safe heads is that it will be less likely to choose any risky head. So a certificate’s lies about the risky heads will be less likely to get detected. However, the probability \(p\) of choosing any head is still non-zero, as long as the bias is not absolute. Thus, the chances of repeatedly missing lies for \(m\) rounds will be at most \((1 − p)^m\), which can also be lowered to any non-zero value by increasing \(m\).

**Proof.** Let \(A \in \text{NL}\), and \(M_A = (Q, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\) be an always halting 2nfa\((k)\) recognizing \(A\) with at least one safe head.

Let \(s = [\log k]\). Regarding the risky heads, let \(H_R\) be the set of their indices, \(k_R\) be their count, and if \(k_R > 0\), let \(\nu_R: \{0, 1\}^s \to H_R\) be any total function, such that each \(i\) in \(H_R\) is mapped to by \(\nu_R\) exactly \([2^s / k_s]\) or \([2^s / k_s]\) many times. Regarding the safe heads, let \(H_S\), \(k_S\), and \(\nu_S\) be defined analogously.

The following parameters will be controlling the error of the verifier:

- \(m\) as the number of rounds to simulate
- \(P_R < 1\) as the probability that the selected head is a risky head, which must be finitely representable in binary, and 0 if and only if \(k_R\) is zero

Let \(r\) be the minimum number of fractional digits to represent \(P_R\) in binary. Then, the algorithm for \(\mu_3(A)\) is as follows:

\(\mu_3(A) = \text{“On input } x:\)"

1. Repeat \(m\) times:
   1. Move the tape head to its original position.
   2. Choose \(i\) from \(\{1, \ldots, k\}\) randomly with bias, as follows:
      1. Flip \(r\) coins for a uniformly random binary probability value \(t\) with \(r\) fractional digits.
      2. Flip \(s\) more coins. Let \(u \in \{0, 1\}^s\) be the outcomes.
      3. Choose \(i\) as \(\nu_R(u)\) if \(t < P_R\), and as \(\nu_S(u)\) otherwise.


7. Let \( q = q_0 \). Repeat the following until \( q = q_{\text{accept}} \):

8. Read \( y \in \Sigma^k \) from the certificate. If \( y_i \) differs from the symbol under the tape head, reject.

9. Read \( (q', d) \in Q \times \Delta^k \) from the certificate. If \( (q', d) \notin \delta(q, y) \), or \( q' = q_{\text{reject}} \), reject.

10. Set \( q = q' \). Move the tape head by \( d_i \).

11. Accept.”

An iteration of stage 1 is called a round. The string of symbols read from the certificate during a round is called a round of certificate. Running on a non-member input string, \( \mu_3(A) \) false accepts for a round, when that round ends without rejecting. Similarly, \( \mu_3(A) \) loops on a round, when that round does not end.

Verifier \( \mu_3(A) \) keeps track of \( M_A \)'s state, starting from \( q_0 \), and advancing it by \( \delta \) and the reports of the certificate. At any given round, \( \mu_3(A) \) can either be led to the state \( q_{\text{accept}} \) and pass, to the state \( q_{\text{reject}} \) and reject, to follow a loop of transitions availed by \( \delta \) and run indefinitely, or to a verification failure and again reject. Since these are events of distinct premises, a certificate may not lead \( \mu_3(A) \) to any combination of those at the same time, regardless of \( \mu_3(A) \)'s random choice of head to verify.

Verifier \( \mu_3(A) \) running on an input string \( x \in A \) always accepts, if paired with a proper certificate that provides \( m \) rounds of certificate, each logging an accepting execution path of \( M_A \).

Given an input \( x \notin A \), every execution path of the always halting 2nfa(\( k \)) \( M_A \) recognizing \( A \) rejects eventually. For \( \mu_3(A) \) to accept \( x \), or loop on it, a certificate \( c(x) \) must be reporting an execution path that is possible by \( \delta \), however impossible for \( M \) running on \( x \). The weak point of \( \mu_3(A) \)'s verification is the fact that it overlooks \( k - 1 \) symbols in stage 1. Hence, \( c(x) \) must lie about those overlooked symbols. Since, however, \( \mu_3(A) \) chooses a head to verify randomly and in private, lies about a head in \( c(x) \) have just as much chance of being detected as how often that head gets selected.

Let \( p \) be the probability of \( \mu_3(A) \) choosing the least likely head of \( M_A \). By the restrictions on \( P_\mu \), and the definition of \( \nu_s \) and \( \nu_r \), every head of \( M_A \) has a non-zero chance of being chosen, and therefore \( p > 0 \). If \( c(x) \) has an inaccuracy, then \( p \) is also the minimum probability of it being detected.

Falsely accepting a string \( x \) is possible for \( \mu_3(A) \), only if \( x \) is not a member of \( A \), \( c(x) \) is an inaccurate certificate with more than \( m \) rounds, and \( \mu_3(A) \) fails to detect the inaccuracies in each round. The probability of this event is at most

\[
(1 - p)^m.
\]

Looping on a string \( x \notin A \) is possible for \( \mu_3(A) \), only if \( c(x) \) is an inaccurate certificate with \( m' \leq m \) rounds, \( \mu_3(A) \) fails to detect the inaccuracies in each round, and \( \mu_3(A) \) chooses a risky head on the final and infinite round. The probability of this event is at most

\[
(1 - p)^{m' - 1} \cdot P_\mu.
\]
The probability that $\mu_3(A)$ falsely accepts (Equation (1)) can be reduced arbitrarily to any non-zero value by increasing $m$. The probability that it loops on a non-member input (Equation (2)) can also be reduced to any positive value by reducing $P_R$ if $k_R > 0$, and is necessarily 0 otherwise.

Verifier $\mu_3(A)$ uses $m \cdot (r + s)$ coins in its execution; a constant amount that does not depend on the input string.

In summary, given any language $A \in \mathsf{NL}$ that can be recognized by a $2\mathsf{nfa}(k)$ with at least one safe head, and for any error bound $\varepsilon > 0$, $\mu_3(A)$ can verify memberships to $A$ within that bound. The amount of coins $\mu_3(A)$ uses depends only on $\varepsilon$, and is constant with respect to the input string.

3.3. Linear-time $2\mathsf{nfa}(k)$’s and safe heads

Lemma 8. Given a language $A$, the following statements are equivalent:

1. $A \in \mathsf{L}(2\mathsf{nfa}(k), \text{linear-time})$.

2. $A$ is recognized by a $2\mathsf{nfa}(k)$ with at least one safe head.

The proof of Lemma 8 will be in two parts.

Proof of (1) $\iff$ (2) Given $A \in \mathsf{L}(2\mathsf{nfa}(k), \text{linear-time})$, for some $k$, there exists a $2\mathsf{nfa}(k)$ $M$ recognizing $A$ together with a constant $c$, such that given any input string $x$, $M$ halts in at most $c \cdot |x|$ steps. Consider the $2\mathsf{nfa}(k+1)$ $M'$, which operates its first $k$ heads by $M$’s algorithm, and uses its last head $T$ as a timer that moves to the next cell on the input tape every $c^{th}$ step of the execution. Head $T$ times out when it reaches the end of the string, and $M'$ rejects in that case.

Note that $M'$ recognizes indeed the same language as $M$, given that $M$, as well as $M'$, runs for at most $c \cdot |x|$ steps for any given input string $x$, and therefore $T$ cannot ever reach the end of $x$. Moreover, head $T$ in $M'$ is a safe head.

Proof of (2) $\iff$ (1) Let $M$ be a $2\mathsf{nfa}(k)$ recognizing $A$, such that its $i^{th}$ head is a safe head. Let $\delta : Q \times \Gamma^k \rightarrow \mathcal{P}(Q \times \Delta^k)$ be the transition function of $M$. Let $M_i$ be the $2\mathsf{nfa}$ with the following transition function as in Definition 1

$$
\delta_i(q, x) = \bigcup_{y \in \Gamma^k} \{ (r, d_i) \mid (r, d) \in \delta(q, y) \}
$$

Note the relationship between the computational paths (sequences of configurations) of $M$ and $M_i$ running on the same input string. These machines have the same state set, but $M_i$ is running a program which has been obtained from the program of $M$ by removing all constraints provided by all the other $k - 1$ heads. If one looks at any possible computational path of $M$ through “filters” that only show the current state and the present position of the $i^{th}$ head, and hide the rest of the information in $M$’s configurations, one will only see legitimate computational paths of $M_i$. 

Since the \(i\)th head is safe, \(M_i\) is always halting, and \(\delta_i\) does not allow \(M_i\) to ever repeat its configuration in a computation. But this means that \(M\) is also unable to loop forever, since the two components of its configuration (the state and the position of its \(i\)th head) can never be in the same combination of values at two different steps. So \(M\) can not run for more than \(|Q| \cdot (n + 2)\) steps, where \(n\) is the length of the input string.

We have proven the following theorem.

**Theorem 9.** \(L(2nfa(k), \text{linear-time}) \subseteq 1IP_*(\infty, \text{cons, cons})\).

Note that the following nonregular languages, among others, have linear-time \(2nfa(k)\)’s, and can therefore be verified with arbitrarily small error by constant-randomness, constant-space verifiers:

\[
\begin{align*}
\text{EQ1} & = \{ 0^i1^i \mid i \in \mathbb{N} \} \\
\text{PAL} & = \{ x \mid x \text{ is the reverse of itself} \} \\
\text{EQ2} & = \{ x \mid x \in \{0, 1\}^* \text{ and contains equally many 0’s and 1’s} \} \\
\text{CERT} & = \{ x_1 \cdots x_l \# x_l^+ \cdots x_1^+ \mid l > 0 \text{ and } x_1, \ldots, x_l \in \{0, 1\} \}
\end{align*}
\]

There are \(2\text{dfa}(2)\)’s without risky heads recognizing the languages \(\text{EQ1}\) and \(\text{PAL}\). We have not been able to find \(2\text{dfa}(k)\)’s without risky heads that recognize the languages \(\text{EQ2}\) and \(\text{CERT}\).

4. Discussion and conclusion

Having determined that \(L(2nfa(k), \text{linear-time}) \subseteq 1IP_*(\infty, \text{cons, cons}) \subseteq \text{NL}\), it is natural to ask if any one of these subset relationships can be replaced by equalities. Let us review the evidence we have at hand in this matter.

One approach to prove the claim that constant-space, constant-randomness verifiers can be constructed for every desired positive error bound (i.e. that \(1IP_*(\infty, \text{cons, cons}) = 1IP(\infty, \text{cons, cons})\)) would be to show that \(\text{NL} = L(2nfa(k), \text{linear-time})\), i.e. that any \(2nfa(k)\) has a linear-time counterpart recognizing the same language. This, however, is a difficult open question \[3\]. As a matter of fact, there are several examples of famous languages in \(\text{NL}\), e.g. \(\text{PATH} = \{ (G, s, t) \mid \text{there is a path from node } s \text{ to node } t \text{ on the graph } G \}\), for which we have not been able to construct \(2nfa(k)\)’s with a safe head, and we conjecture that \(L(2nfa(k), \text{linear-time}) \neq \text{NL}\).

We will now show that \(L(2nfa(k), \text{linear-time})\) is contained in a subset of \(\text{NL}\) corresponding to a tighter time restriction of \(O(n^2/\log(n))\) on the underlying nondeterministic Turing machine.\[4\] \(\text{NTISP}(f(n), g(n))\) denotes the class of languages that can be verified by a \(\text{TM}\) that uses \(O(f(n))\) time and \(O(g(n))\) space, simultaneously.

\[4\]Recall that logarithmic-space \(\text{TM}\)’s require \(\Omega(n^2/\log(n))\) time for recognizing the palindromes language \[10, 11, 12\] which is easily recognized by a linear-time \(2\text{dfa}(2)\).
Theorem 10. \( \mathcal{L}(2\text{nf}\alpha(k), \text{linear-time}) \subseteq \text{NTISP}(n^2/\log(n), \log n) \)

**Proof idea.** Given a 2nf\(\alpha(k) \) \(M\) that runs in linear time, an NTM \(N\) can simulate it in \(O(n^2/\log(n))\) steps. One such \(N\) uses \(k\) counters for keeping the head positions of \(M\), and \(k\) caches for a faster access to the symbols in the vicinity of each head, on a tape with \(2k\) tracks. \(N\) initializes its caches with a \(\triangleright\) symbol, followed by the first \(\log(n)\) symbols of the input, and puts a mark on \(\triangleright\) symbols to indicate the position of each simulated head. Counters are initialized as 0 for yet another indication of the head positions.

To mimic \(M\) reading its tape, \(N\) reads the marked symbols on its caches. To move the simulated heads, \(N\) both moves the marks on the caches, and adjusts the counters. If a mark reaches the end of its cache, \(N\) re-caches by copying the \(\log(n)\) symbols centered around the corresponding head from the input to that cache. Counters provide the means for \(N\) to locate these symbols on the input.

As the analysis will show, the algorithm described for \(N\) runs within the promised time and space bounds. In the following proof, \(N\) will have an additional track that has a mark on its \(\log(n)/2^{th}\) cell to indicate the middle of the caches.

**Proof.** Let \(M = (Q, \Sigma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\) be a 2nf\(\alpha(k) \) that runs in linear time. An NTM \(N\) can simulate \(M\), by using \(2k + 1\) tracks on its tape to have:

- \(k\) \(\log(n)\) digit binary counters, \(\kappa_1, \ldots, \kappa_k\), with their least significant digit on their left end,
- \(k\) caches of input excerpts of \(\log(n)\) length, \(\eta_1, \ldots, \eta_k\), and
- a mark on the \(\log(n)/2^{th}\) cell to indicate the middle.

The work tape alphabet \(\Gamma = \Gamma_\kappa \times \Gamma_\eta \times \{\triangleright, \triangleleft\}\) allows \(N\) to encode those information, where:

- \(\Gamma_\kappa = \{0, 1, \_\}\) to represent each \(\kappa_i\)
- \(\Gamma_\eta = \Sigma_\circ \cup \hat{\Sigma}_\circ\) to represent each cache, where:
  - \(\Sigma_\circ = \Sigma \cup \{\triangleright, \triangleleft, \#, \_\}\), and
  - \(\hat{\Sigma}_\circ\) is a clone of \(\Sigma_\circ\), containing “marked” versions of all \(\Sigma_\circ\)’s symbols.

Initially, all cells of the work tape contain the symbol \(\_\) \(2k+1\). The algorithm of \(N\) is as follows:

\(N = \text{"On input } x \text{ of length } n:\)

1. Write 0 to each \(\kappa_i\).
2. Write \(\#\triangleright x_1 \ldots x_{\log(n)}\#\) on to each \(\eta_i\).
3. Write \(\triangleleft\) to the \(\log(n)/2^{th}\) cell of the last track.
4. Let \(q = q_0\). Repeat the following until \(q = q_{\text{accept}}:\)
5. Scan the caches. Note the marked symbol in $\eta_i$ as $y_i$.
6. Guess a $(r, d) \in \delta(q, y_1 \cdots y_k)$. Reject if the set is empty, or $r = q_{\text{reject}}$.
7. For all $i$, adjust $\kappa_i$, and move the mark on $\eta_i$ by $d_i$.
8. Re-cache each $\eta_i$ that has a $\#$ symbol, as follows:
   a. Clear the mark on $\#$ of $\eta_i$.
   b. Go to $\kappa_i$'th cell on the input.
   c. Go to middle of $\eta_i$ on the work tape.
   d. Move both tape heads left, until the left end of $\eta_i$ is reached.
   e. Copy $\log(n)$ symbols from the input to between the $\#$ symbols of $\eta_i$.
   f. Move both tape heads left, until the middle of $\eta_i$ is reached.
   g. Mark the middle symbol on $\eta_i$.
   h. Set $\kappa_i$ to the input head’s position index.
9. Update $q$ as $r$.
10. Accept."

$N$ should carefully prepend/append the left/right end marker to a cache when copying the beginning/end of the input in stage 13, respectively. $N$ should also skip stage 7 for an $i$, if the corresponding movement is done while reading an end marker and attempting a movement beyond it. These details have been omitted from the algorithm to reduce clutter.

Counting up to $n$ in binary is a common task across this algorithm, and takes linear time, by a standard result of amortized analysis. Only the stages that take a constant amount of steps are omitted from the following analysis.

Stage 2 takes $O(n)$ time as it involves counting up to $n$ in binary to find and marking the $\log(n)$th cell on the caches. After putting $\#$ on both ends, copying $x_1 \cdots x_{\log(n)}$ in takes $\log(n)$ more steps. Stage 3 can be performed in $O(\log^2 n)$ steps, by copying $\#$ symbols over from a cache, and moving them towards the center one by one until they meet.

Given that $M$ runs in linear time, the loop of stage 4 is repeated for at most $O(n)$ many times. Stages 5 and 7 take logarithmic time.

The re-caching in stage 8 is to shift the window of input on a cache by $\log(n)/2$, so that the mark will be centered on that cache. Stages 9 and 10 are the most time consuming sub-stages of a re-cache, involving decrementing of $\kappa_i$ down to 1, and setting it back to its original value, respectively. They both take $O(n)$ time, for that they count down from or up to $n$ at most. Every other sub-stage of a re-cache takes $O(\log n)$ time. As a result, each re-cache takes $O(n)$ time.

Re-caches are prohibitively slow. Luckily, since the head marker is shifted to the middle with each re-cache, a subsequent re-cache will not happen on the same cache for at least another $\log(n)/2$ steps of the simulation. Moreover, since the number of steps that $M$ runs is in $O(n)$, the number of times a cache can be re-cached is in $O(n/\log(n))$ for the entire simulation. Hence, stage 8's time

---

\[9\text{This is done using the states of } N, \text{ and does not use work tape.}\]
cost to \(N\) is \(O(n^2/\log(n))\).

Caches and counters occupy \(O(\log n)\) cells on \(N\)’s tape. Since every stage of \(N\) runs in \(O(n^2/\log(n))\) time, so does \(N\). 

It is not known whether NL contains any languages that are not members of NTISP\((n^2/\log(n), \log(n))\).

If 1IP\((\infty, \text{cons, cons})\) is indeed a proper subset of \(1\Pi(\infty, \text{cons, cons})\), studying the effects of imposing an additional time-related bound on the verifier may be worthwhile in the search for a characterization. We conclude by noting the following relationship between runtime, the amount of randomness used, and the probability of being fooled by a certificate to run forever in our setup:

**Lemma 11.** Let \(V\) be a 2pfa verifier flipping \(r\) coins at most in a private-coin one-way IPS, and recognizing the language \(A\). If \(V\) running on some string \(x \notin A\) of length \(n\) paired with some certificate \(c(x)\) always takes \(\omega(n^{2r-1})\) steps, then its error is at least \(1/2\).

**Proof.** Let \(V\) be a 2pfa as described above, recognizing \(A\). By an idea introduced in [3], we will construct a verifier equivalent to \(V\). For \(z \in \{0, 1\}^r\), let \(V_z\) be the 2dfa verifier that is based on \(V\), but hard-wired to assume that its \(i\)th “coin flip” has the outcome \(z_i\). Construct the 2pfa verifier \(V'\) that flips \(r\) coins at the beginning of its execution, and obtains the \(r\)-bit random string \(z\). Then, \(V'\) passes the execution to \(V_z\).

Verifiers \(V\) and \(V'\) have the same control, whenever their random bits are the same. Therefore, they are equivalent.

Each \(V_z\) has \(O(n)\) different configurations, where \(n\) denotes the length of the input string. Similarly, any collection of \(2^{r-1}\) distinct \(V_z\) has \(O(n^{2r-1})\) different collective configurations. Let \(\mathcal{V}\) be any one of those collections.

Let \(x\) and \(c(x)\) be a nonmember string and its certificate satisfying the premise of the statement. Then, each \(V_z\) paired with \(c(x)\) also runs on \(x\) for \(\omega(n^{2r-1})\) steps. The collection \(\mathcal{V}\), in that many steps, necessarily repeats a collective configuration.

Consider the prefix \(p(x)\) of \(c(x)\) consumed by \(V'\) until the first time a collective configuration of \(\mathcal{V}\) is repeated. Also consider the suffix \(s(x)\) of \(p(x)\) consumed by \(V'\) since the first occurrence of the repeated collective configuration. Then, \(V'\) paired with the certificate \(c'(x) = p(x)s(x)\infty\) repeats its configurations indefinitely whenever it chooses any of the \(V_z \in \mathcal{V}\) to pass the execution to.

Both \(V'\) and \(V\) paired with \(c'(x)\) loop on \(x\) with a probability at least \(1/2\). Consequently, their errors are at least \(1/2\). \(\square\)

**Acknowledgements**

We are grateful to Martin Kutrib, Neal E. Young, Ryan O’Donnell, and Ryan Williams for their helpful answers to our questions. We also thank the anonymous referees of [1] for their constructive comments.
References

[1] M. U. Gezer, Windable heads and recognizing NL with constant randomness, in: A. Leporati, C. Martín-Vide, D. Shapira, C. Zandron (Eds.), Language and Automata Theory and Applications, Springer International Publishing, Cham, 2020, pp. 184–195.

[2] A. Condon, R. Ladner, Interactive proof systems with polynomially bounded strategies, Journal of Computer and System Sciences 50 (3) (1995) 506–518.

[3] A. C. C. Say, A. Yakaryılmaz, Finite state verifiers with constant randomness, Logical Methods in Computer Science 10 (3) (Aug. 2014).

[4] M. Sipser, Introduction to the Theory of Computation, Cengage Learning, 2012.

[5] J. Hartmanis, On non-determinacy in simple computing devices, Acta Informatica 1 (4) (1972) 336–344.

[6] B. Monien, Two-way multihead automata over a one-letter alphabet, RAIRO. Inform. théor. 14 (1) (1980) 67–82.

[7] V. Geffert, A. Okhotin, Transforming two-way alternating finite automata to one-way nondeterministic automata, in: International Symposium on Mathematical Foundations of Computer Science, Springer, 2014, pp. 291–302.

[8] C. Dwork, L. Stockmeyer, Finite state verifiers I: The power of interaction, J. ACM 39 (4) (1992) 800âĂŞ828.

[9] A. Condon, The complexity of the max word problem and the power of one-way interactive proof systems, computational complexity 3 (3) (1993) 292–305.

[10] A. Cobham, Time and memory capacity bounds for machines which recognize squares or palindromes, IBM Res. Rep. RC-1621 (1966).

[11] D. van Melkebeek, Time-space lower bounds for NP-complete problems, in: G. Plun, G. Rozenberg, A. Salomaa (Eds.), Current Trends in Theoretical Computer Science, World Scientific, 2004, pp. 265–291.

[12] P. Dúriš, Z. Galil, A time-space tradeoff for language recognition, Mathematical systems theory 17 (1) (1984) 3–12.