

# Constant-Space, Constant-Randomness Verifiers with Arbitrarily Small Error<sup>\*</sup>

M. Utkan Gezer<sup>\*</sup>, A. C. Cem Say

*Department of Computer Engineering, Boğaziçi University, Bebek 34342, 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 Yakaryılmaz 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 when the definition of “error” is strengthened to include looping forever without giving a response. 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 nondeterministic finite automaton ( $2nfa(k)$ ). 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  $2nfa(k)$ ’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 major concern of computational complexity theory. In this context, important tradeoffs among different types of resources such as time, space, and randomness have been demonstrated: The power of deterministic polynomial-time, polynomial-space bounded verifiers characterized by the class NP has, for instance, been shown to be identical to

---

<sup>\*</sup>This paper is a substantially improved version of [1].

<sup>\*</sup>Corresponding author

*Email addresses:* [utkan.gezer@boun.edu.tr](mailto:utkan.gezer@boun.edu.tr) (M. Utkan Gezer), [say@boun.edu.tr](mailto:say@boun.edu.tr) (A. C. Cem Say)

that of probabilistic bounded-error polynomial-time logarithmic-space verifiers that toss only logarithmically many coins in terms of the input size [2].

The study of finite-state probabilistic verifiers started in the late 1980's. Condon and Lipton [3] showed that, even under this severe space restriction, one can verify membership in any Turing-recognizable language if one is not required to halt with high probability on rejected inputs. Dwork and Stockmeyer [4] showed that interactive proof systems with constant-space verifiers outperform “stand-alone” finite-state recognizers when required to halt with high probability as well. The area has grown to have a rich literature where scenarios with multiple provers and quantum verifiers have also been considered. The study of interactive proof systems with quantum finite automata, which was initiated by Nishimura and Yamakami [5, 6], continued with the consideration of more powerful quantum models by Yakaryılmaz [7] and Zheng et al. [8]. The power of finite-state verifiers that are faced with two opposing provers were studied by Feige and Shamir [9] and Demirci et al. [10] in the classical setup, and by Yakaryılmaz et al. [11] in the quantum setup.

Recently, Say and Yakaryılmaz initiated the study of the power of classical finite-state verifiers that are restricted to toss some fixed, nonzero number of coins regardless of the input size, and proved [12] that the class of languages which have certificates 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 [12] could exhibit a constant-randomness verifier operating within error  $\varepsilon$  for some  $\varepsilon < 1/2$  for any language in NL, however, it provided a method for reducing this error to more desirable smaller values only in the “weak” regime where looping forever without a response is not considered to be an error. Indeed, when the error definition is strengthened to include this behavior, 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 ( $2nfa(k)$ ). 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  $2nfa(k)$ ’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 (TMs), and basic complexity classes [13].

The following notation will be used throughout this paper:

- $\mathcal{P}(A)$  is the power set of  $A$ .
- $A \sqcup B$  is the union of sets  $A$  and  $B$ , that also asserts that the two are disjoint.
- $\sigma\tau$  is the sequences  $\sigma$  and  $\tau$  concatenated.
- $\sigma_i$  is the  $i^{\text{th}}$  element of the sequence  $\sigma$ .
- $\langle O_1, \dots, O_k \rangle$  is the encoding of objects  $O_i$  in the alphabet of context.

### 2.1. Multihead finite automata

A (two-way)  $k$ -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
  - $\Gamma = \Sigma \sqcup \{ \triangleright, \triangleleft \}$  is the tape alphabet, where  $\triangleright$  and  $\triangleleft$  are respectively the left and right end markers, and
  - $\Delta = \{ -1, 0, +1 \}$  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  $q_0 \in Q$ ;
5. an accept state  $q_{\text{acc}} \in Q$ ; and
6. a reject state  $q_{\text{rej}} \in Q$ .

The 2 in the denotation  $2\text{nfa}(k)$  indicates that these automata can move their heads in both directions, i.e. that their heads are two-way. For the rest of the paper, unless specified otherwise, our (multi-head) finite automata should be assumed as two-way.

A  $2\text{nfa}(k)$   $M = (Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$  starts from the state  $q_0$  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  $i^{\text{th}}$  head by  $d_i$ .

The *configuration* of a  $2\text{nfa}(k)$   $M$  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  $2\text{nfa}(k)$   $M$  may have several *computational paths* on the same string. A computational path of  $M$  *halts* if it reaches  $q_{\text{acc}}$  or  $q_{\text{rej}}$ , or if  $\delta$  does not offer any further 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  $q_{\text{acc}}$ .  $M$  *rejects* an input string  $x$  if  $M$  running on  $x$  halts on a state other than  $q_{\text{acc}}$  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  $M$  halts on every computational path for every input string,  $M$  is said to be an *always halting 2nfa(k)*.

A (two-way)  $k$ -head *deterministic* finite automaton, denoted  $2dfa(k)$ , differs from a  $2nfa(k)$  in its transition function, which is defined as  $\delta: Q \times \Gamma^k \rightarrow Q \times \Delta$ . 1-head finite automata are simply called finite automata and are denoted as  $2dfa$  and  $2nfa$  for the deterministic and nondeterministic counterparts, respectively.

For any  $k$ , let  $\mathcal{L}(2nfa(k))$  denote the class of languages recognized by a  $2nfa(k)$ .  $\mathcal{L}(2nfa(1))$  is the class of regular languages [14].

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

**Lemma 1.** *Nondeterministic multi-head finite automata are equivalent to logarithmic space NTMs in terms of language recognition power [15]. Put formally;*

$$\bigcup_{k>0} \mathcal{L}(2nfa(k)) = \text{NL}.$$

**Lemma 2.** *The languages in NL are organized in a strict hierarchy, based on the number of heads of the nondeterministic finite automata recognizing them [16]. Formally, the following is true for any  $k > 0$ :*

$$\mathcal{L}(2nfa(k)) \subsetneq \mathcal{L}(2nfa(k+1))$$

For any given  $k$ , let  $\mathcal{L}(2nfa(k), f(n))$  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  $\mathcal{L}(2nfa(*), f(n))$  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  $k > 0$ :*

$$\mathcal{L}(2nfa(k)) \subseteq \mathcal{L}(2nfa(2k), n^k)$$

*Proof.* Let  $M$  be any  $2nfa(k)$  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 more than  $T$  steps, then it must have repeated a configuration. Therefore, for every input string it accepts,  $M$  should have an accepting computational 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  $c_1, \dots, c_k$  denote the counter heads. Head  $c_1$  moves right every  $|Q|^{\text{th}}$  step of  $M$ 's simulation. For all  $i < k$ , 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 the right. If  $c_k$  attempts to move past the right end,  $M'$  rejects.

If the simulation halts before timeout,  $M'$  reports  $M$ 's decision. The strings that  $M$  would loop on are rejected by  $M'$  due to timeout. The  $2\text{nfa}(2k)$   $M'$  recognizes the same language as  $M$ , but within the time limit of  $O(n^k)$ .  $\square$

Lemmas 1 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  $2\text{nfa}(k_A)$  recognizing  $A$ , but not an always halting  $2\text{nfa}(h)$  where  $h < k_A$ .*

*Proof.* Let  $K_A$  be the set of numbers of heads of always halting multi-head non-deterministic finite automata recognizing  $A$ . By Lemmas 1 and 3, for some  $k$ , there is a  $2\text{nfa}(k)$  and thereby an always halting  $2\text{nfa}(2k)$  recognizing  $A$ , respectively. Thus,  $2k \in K_A$  and  $K_A$  is non-empty. By the well-ordering principle,  $K_A$  has a least element, which we call  $k_A$ .  $\square$

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

*Proof.* The *two-way alternating finite automaton*, denoted  $2\text{afa}$ , is a generalization of the  $2\text{nfa}$  model. The state set of a  $2\text{afa}$  is partitioned into *universal* and *existential* states. A  $2\text{afa}$  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  $2\text{nfa}$  is a  $2\text{afa}$  with only existential states. We refer the reader to [17] for a formal definition of the  $2\text{afa}$  model.

A *one-way nondeterministic finite automaton*, denoted  $1\text{nfa}$ , is a  $2\text{nfa}$  that cannot move its head to the left. A  $1\text{dfa}$  is a deterministic  $1\text{nfa}$ .

Consider the following algorithm to recognize  $\text{HALTING}_{2\text{nfa}}$ :

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

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

By its construction,  $M'_{2\text{afa}}$  (and therefore  $M'_{1\text{dfa}}$ ) recognizes  $\Sigma^*$  if and only if  $M$  halts in every computational path while running on every possible input string, i.e. it is always halting. Stages 2 and 3 can be implemented by the algorithms given in [18] and the proof for the Theorem 1.39 of [13], respectively. The final check in stage 4, also known as the universality problem for  $1\text{dfa}$ 's, is decidable in nondeterministic logarithmic space [19], thus in polynomial time by Corollary 8.26 in [13]. So the algorithm  $D$  decides whether a given  $2\text{nfa}$   $M$  is always halting.  $\square$

## 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 (two-way) 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.<sup>1</sup> 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  $Q = Q_d \sqcup Q_r$ , where
  - $Q_d$  is the set of deterministic states, and
  - $Q_r$  is the set of coin-tossing states.
2. An input alphabet  $\Sigma$ .
3. A transition function overloaded as deterministic  $\delta_d$  and coin-tossing  $\delta_r$ , where
  - $\delta_d: Q_d \times \Gamma \rightarrow Q \times \Delta$ , where  $\Gamma$  and  $\Delta$  are as defined for the 2nfa( $k$ )'s, and
  - $\delta_r: Q_r \times \Gamma \times R \rightarrow Q \times \Delta$ , where  $R = \{0, 1\}$  is a random bit provided by a “coin toss”.
4. An initial state  $q_0$ .
5. An accept state  $q_{acc}$ .
6. A reject state  $q_{rej}$ .

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

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

---

<sup>1</sup>One may also think of a 2pfa as a 2nfa where each state has probabilities associated with each of its outgoing transitions, and the machine selects which transition to follow with these corresponding probabilities. To make this alternative model equivalent to the constant-randomness machines studied in this paper, it is sufficient to restrict the transition probabilities to dyadic rationals.

### 2.3. Interactive proof systems

Our definitions of interactive proof systems (IPSeS) are based on [4]. We will focus on a single variant, namely the private-coin one-way IPS with a finite-state verifier.

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 a *private-coin one-way IPS*, the coin flips (both their outcomes and the information on when they are flipped) are hidden from the prover  $P$ , and  $P$  communicates the proof to the verifier  $V$  in a monologue. In such an IPS, as a simplification,  $P$  can instead 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.  $V$ , in turn, can be thought of as having an additional certificate tape to read with a head that cannot move to the left. Given an input string  $x \in \Sigma^*$ ,  $V$  executes on it as usual with  $c(x)$  written on its certificate tape.

Note that the “one-way” denotation for an IPS qualifies only the interaction (i.e. specifies that the verifier does not communicate back), and not the head movements of the verifier.

In this paper, the term “PTM verifier in a private-coin one-way IPS” will be abbreviated as “PTM verifier”. Accordingly, “2pfa verifier” shall mean “two-way probabilistic finite automaton verifier in a private-coin one-way IPS”.

The language  $A$  of a PTM verifier  $V$  is the set of strings that  $V$  can be “convinced” to accept with a probability greater than  $1/2$  by some certificate function  $c$ . The error bound<sup>2</sup> of  $V$ , denoted  $\varepsilon_V$ , is then defined as the minimum value satisfying both of the following:

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

Let  $1IP_\varepsilon(t(n), s(n), r(n))$  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  $n$ , we write simply **cons**, **log**, **poly**, and **exp** to describe constant, logarithmic, polynomial, and exponential limits in terms of the input length, respectively. We write 0 and  $\infty$  to describe that a resource is unavailable and unlimited, respectively. Furthermore, let

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

---

<sup>2</sup>Our definition of the error bound corresponds to the “strong” version of the IPS definition in [4].

and

$$1IP_*(t(n), s(n), r(n)) = \bigcap_{\varepsilon > 0} 1IP_\varepsilon(t(n), s(n), r(n)).$$

The following are trivial:

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

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

The class NP is further characterized [20, 2] as

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

and the class NL [12] as

$$NL = 1IP(\infty, \text{cons}, \text{cons}).$$

For polynomial-time verifiers with the ability to use at least logarithmic space, the class  $1IP_*(t(n), s(n), r(n))$  is identical to the corresponding class  $1IP(t(n), s(n), r(n))$ , 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  $1IP_*(\infty, \text{cons}, \text{cons})$  equals  $1IP(\infty, \text{cons}, \text{cons})$  is nontrivial, as we will examine in the following sections.

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

In [12], Say and Yakaryılmaz showed that membership in any language in NL may be checked by a 2pfa verifier using some constant number of random coin tosses. They also showed how the *weak* error of the verifier can be made arbitrarily small.<sup>3</sup> 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  $A \in NL$ , takes an always halting 2nfa( $k$ )  $M_A$  recognizing  $A$  (for some  $k$ ), which exists by Lemmas 1 and 3, as its starting point. The constructed verifier  $\mu_1(A)$  will attempt to repeatedly simulate  $M_A$  by looking at the certificate while relying on its private coins to compensate for having  $k - 1$  fewer input heads than  $M_A$ . Given any input string  $x$ ,  $\mu_1(A)$  expects a certificate  $c(x)$  to contain  $m$  successive segments, each of which describe an accepting computational path of  $M_A$  on  $x$ .  $c(x)$  is supposed 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

---

<sup>3</sup>In contrast to the (strong) error definition we use in this paper, the weak error definition (also by [4]) does not regard the verifier looping forever on a non-member input as an error.



taken. Verifier  $\mu_1(A)$  attempts to simulate  $M_A$  through the provided computational path until either the simulation halts, or  $\mu_1(A)$  catches a “lie” in the certificate and *rejects*.  $\mu_1(A)$  chooses a head of  $M_A$  at random by tossing  $\lceil \log k \rceil$  coins in private before each simulation. Throughout the simulation,  $\mu_1(A)$  mimics the movements of this chosen head and compares  $c(x)$ ’s claims against what is being scanned by that head, while leaving the remaining  $k - 1$  unverified. If the simulation rejects, then so does  $\mu_1(A)$ . If  $m$  such simulation rounds end with acceptance,  $\mu_1(A)$  *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 tosses 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 false values for the symbols read by at least one of the heads. Since  $\mu_1(A)$  cannot check many of the actual readings, it may fail to notice those lies. However, since  $\mu_1(A)$  chooses a head to verify at random, there is a non-zero chance that  $\mu_1(A)$  detects any such lie.

The likelihood that  $\mu_1(A)$  chooses a head that the certificate is lying about is at least  $1/k$ .<sup>4</sup> Therefore, the weak error of  $\mu_1(A)$  is at most  $((k-1)/k)^m$ . This upper bound for weak error can be brought 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  $\mu_1(A)$  may 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, lies 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 by lying about one head alone. If this happens during the first round, there would not be any more rounds for  $\mu_1(A)$  since it would be in a loop. The (strong) error  $\varepsilon_{\mu_1(A)}$  of  $\mu_1(A)$  is therefore at most  $(k-1)/k$ . This upper bound to  $\varepsilon_{\mu_1(A)}$  cannot be reduced to less than  $(k_A - 1)/k_A$ , where  $k_A$  is the minimum number of heads required in an always halting machine to recognize  $A$  by Corollary 4.

Say and Yakaryılmaz also propose the method  $\mu_2$ , which is a slightly modified version of  $\mu_1$  that produces verifiers with errors less than  $1/2$ , 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  $(k-1)/2k$  by tossing  $\lceil \log k \rceil + 1$  coins. Then it continues just

---

<sup>4</sup>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.

like  $\mu_1(A)$ . The bounds for the error  $\varepsilon_{\mu_2(A)}$  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  $1IP_*(\infty, \text{cons}, \text{cons})$ ? Method  $\mu_1$  was our starting point in working towards a lower bound for  $1IP_*(\infty, \text{cons}, \text{cons})$ .

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

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

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

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

The execution of each  $2\text{nfa}$   $M_i$  in Definition 1 is designed to correspond to the  $i^{\text{th}}$ -head-only simulation of the  $2\text{nfa}(k)$   $M$  by the verifier of  $\mu_1$ . Just like the verifier of  $\mu_1$  and by the way  $\delta_i$  is defined,  $M_i$  can make any of the transitions allowed by  $M$ 's transition function ( $\delta$ ) and chooses one by the certificate while making sure that the  $i^{\text{th}}$  symbol fed to  $M$ 's transition function ( $y_i$ ) is the same as the symbol it is reading itself ( $x$ ). Crucially, if a certificate can wind the verifier of  $\mu_1$  into a loop during the single-headed simulation of  $M$ , then the  $2\text{nfa}$   $M_i$  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.

**Example 1.** Let  $\Sigma = \{0, 1\}$  and  $\text{EQ} = \{0^i 1^i \mid i \in \mathbb{N}\}$ . Figure 1 depicts the state diagram of the  $2\text{nfa}(2)$   $M_{\text{EQ}} = (Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ : Whenever  $(r, d) \in \delta(q, x)$ , an arc is drawn from  $q$  to  $r$  with the label  $x \rightarrow d$ . Recall that  $x$  and  $d$  are tuples of symbols and head movements, respectively. The arc coming from nowhere to  $q_0$  indicates that it is the initial state. Such a pictorial representation of an automaton is called its *state diagram*.

$M_{\text{EQ}}$  starts by moving its second head to the leftmost 1 in the input. It then moves both heads to the right as long as they scan 0 and 1, respectively, and accepts only if this walk ends when these heads simultaneously read 1 and  $\triangleleft$ , respectively. The empty string is accepted immediately.

Figure 2 depicts  $M_{\text{EQ}_1}$ , which is the  $2\text{nfa}$  associated with the first head of  $M_{\text{EQ}}$  obtained as described in Definition 1. It has a computational path that

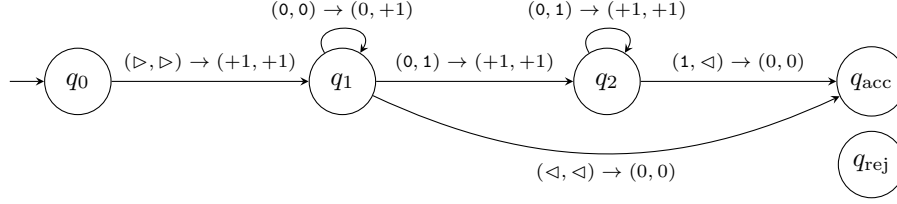


Figure 1: State diagram of  $2\text{nfa}(2) M_{\text{EQ}}$ .

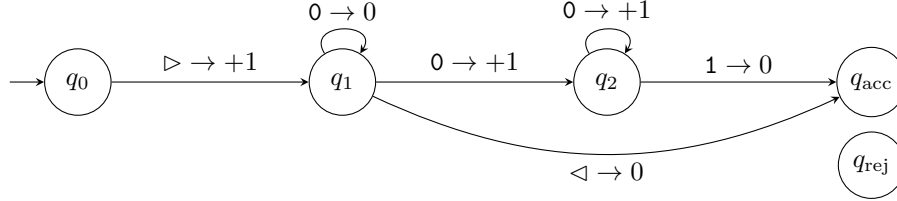


Figure 2: State diagram of  $2\text{nfa} M_{\text{EQ}_1}$ .

loops indefinitely on  $q_1$  given any input string that begins with a 0. Thus,  $M_{\text{EQ}_1}$  is not always halting, and by Definition 1, the first head of  $M_{\text{EQ}}$  is risky.

Indeed,  $\mu_1(\text{EQ})$  simulating  $M_{\text{EQ}}$  and running on an input string that begins with 0 would loop forever at state  $q_1$  if  $\mu_1(\text{EQ})$  were to track the first head, and the certificate were to report infinite sequences of 0's as both heads' readings.

Unlike the first one, the second head of  $M_{\text{EQ}}$  is safe. Figure 3 depicts the  $2\text{nfa} M_{\text{EQ}_2}$  associated with it. Since every transition other than those that lead into the accept state moves the head to the right, the head will eventually reach the end of the input, and  $M_{\text{EQ}_2}$  will accept, unless it gets “stuck” (and implicitly rejects) by encountering a 0 while at state  $q_2$ .

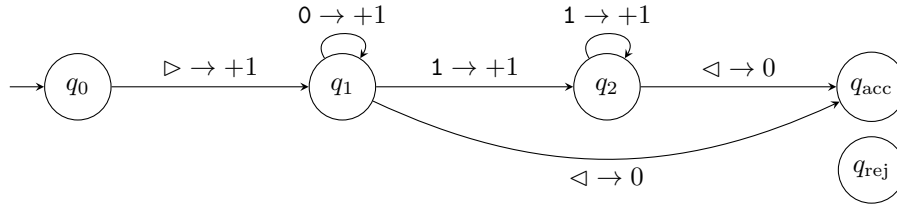


Figure 3: State diagram of  $2\text{nfa} M_{\text{EQ}_2}$ .

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

*Proof.* To decide whether the  $i^{\text{th}}$  head of a  $2\text{nfa}(k) M$  is safe, an algorithm can construct the  $2\text{nfa} M_i$  described in Definition 1 and test whether  $M_i \in \text{HALTING}_{2\text{nfa}}$  by the algorithm in Lemma 5.  $\square$

Consider a language  $A \in \text{NL}$  that is recognized by a  $2\text{nfa}(k)$   $M_A$  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  $\varepsilon_{\mu_1(A)} \leq ((k-1)/k)^m$ .

### 3.2. $2\text{nfa}(k)$ '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  $\mu_3$ , which is 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  $2\text{nfa}(k)$  with at least one safe head that recognizes  $A$ , then  $A \in 1\text{IP}_*(\infty, \text{cons}, \text{cons})$ .*

*Proof idea.* The method  $\mu_3$  in the proof will construct verifiers similar to those of  $\mu_1$ , except with a key difference. Given a language  $A \in \text{NL}$  recognized by an always halting  $2\text{nfa}(k)$   $M_A$  that has at least one safe head, every head of  $M_A$  has essentially the same probability of getting chosen by  $\mu_1(A)$ . If  $M_A$  does not have any risky heads, then  $\mu_3(A)$  will be identical to  $\mu_1(A)$ .

If  $M_A$  does have some risky heads, then  $\mu_3(A)$  will mostly avoid choosing them, although still giving each of them a slight chance to be chosen. Since  $\mu_3(A)$  may loop only if it is tracking a risky head, the looping probability of  $\mu_3(A)$  decreases as the probability of the selected head being risky gets lower.

The redeemable disadvantage of  $\mu_3(A)$ 's bias towards choosing among safe heads is that the certificate's lies about the risky heads are now less likely to be detected. However, since the bias is not absolute, the probability  $p$  of choosing the least likely of the heads is still non-zero. Thus, the probability of detecting an existing lie in any given round is at least  $p$ , and  $\mu_3(A)$  fails to catch a lie in  $m$  rounds with probability at most  $(1-p)^m$ , which can be lowered to any non-zero value by increasing  $m$ .

It is impossible for  $\mu_3(A)$  to read multiple rounds of infinite computational paths from the certificate, since  $\mu_3(A)$  would not be able to get past the first one. The probability of  $\mu_3(A)$  looping is the greatest when an infinite computational path is given in the first round. Thus, the increased number of rounds does not make  $\mu_3(A)$  any more likely to loop.

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

Let  $s = \lceil \log k \rceil$ . Let  $k_s$  and  $k_r$  be the number of safe and risky heads, respectively. Let  $\nu_s(i) \in \{1, \dots, k\}$  be the head index of the  $i^{\text{th}}$  safe head where  $i \in \{1, \dots, k_s\}$ . If  $k_r > 0$ , let  $\nu_r$  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:

$\mu_3(A)$  = “On input  $x$ :

1. Repeat  $m$  times:
  2. Move the tape head to the left end of the input.
  3. Choose  $i$  from  $\{1, \dots, k\}$  randomly with bias, as follows:
    4. Flip  $r$  coins for a uniformly random binary probability value  $t$  with  $r$  fractional digits.
    5. Flip  $s$  more coins for an  $s$ -digit binary number  $u < 2^s$ .
    6. Let  $i = \nu_R((u \bmod k_R) + 1)$  if  $t < P_R$ , and  $\nu_S((u \bmod k_S) + 1)$  otherwise.
  7. Let  $q = q_0$ . Repeat the following until  $q = q_{\text{acc}}$ :
    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  $q' = q_{\text{rej}}$ , *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 pass the round by arriving at  $q_{\text{acc}}$ , *reject* by arriving at  $q_{\text{rej}}$  or via verification error, or loop via a loop of transitions availed by  $\delta$ .

Verifier  $\mu_3(A)$  running on an input string  $x \in A$  accepts with probability 1 when paired with an honest certificate that logs an accepting execution path of  $M_A$  for  $m$  rounds.

Given an input  $x \notin A$ , every execution path of the always halting  $2\text{nfa}(k)$   $M_A$  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  $\delta$ , despite being 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 8. Hence,  $c(x)$  must lie about those overlooked symbols. Since, however,  $\mu_3(A)$  chooses a head to verify randomly and in private, any lie about any head has just as much chance of being caught 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  $P_R$  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 a lie in it, 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  $A$ ,  $c(x)$  lies for more than  $m$  rounds, and  $\mu_3(A)$  fails to detect the lies in each

round. The probability of this event is at most

$$(1 - p)^m. \quad (1)$$

Looping on a string  $x \notin A$  is possible for  $\mu_3(A)$  only if  $c(x)$  is a lying certificate with  $m' \leq m$  rounds,  $\mu_3(A)$  fails to detect the lies 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

$$(1 - p)^{m'-1} \cdot P_R. \quad (2)$$

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  $\mu_3(A)$  tosses  $m \cdot (r + s)$  coins; a constant amount that does not depend on the input string.  $\square$

In summary, given any language  $A \in \text{NL}$  that can be recognized by a  $2\text{nfa}(k)$  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 on  $\varepsilon$  only and is constant with respect to the input string.

### 3.3. Linear-time $2\text{nfa}(k)$ 's and safe heads

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

- (1)  $A \in \mathcal{L}(2\text{nfa}(k), \text{linear-time})$ .
- (2)  $A$  is recognized by a  $2\text{nfa}(k)$  with at least one safe head.

The proof of Lemma 8 will be in two parts.

*Proof of (1)  $\implies$  (2).* Given  $A \in \mathcal{L}(2\text{nfa}(k), \text{linear-time})$ , for some  $k$ , there exists a  $2\text{nfa}(k)$   $M$  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  $2\text{nfa}(k+1)$   $M'$  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  $c^{\text{th}}$  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  $M$  since  $M$ , as well as  $M'$ , runs for at most  $c \cdot |x|$  steps for any given input string  $x$ , and therefore  $T$  never reaches the end of  $x$  nor times out. Apparent from its monotonic movement, however, head  $T$  in  $M'$  is safe.  $\square$

*Proof of (2)  $\implies$  (1).* Let  $M$  be a  $2\text{nfa}(k)$  recognizing  $A$  such that its  $i^{\text{th}}$  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  $2\text{nfa}$  with the following transition function as in Definition 1:

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

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  $i^{\text{th}}$  head and hide the rest of the information in  $M$ ’s configurations, one will only see legitimate computational paths of  $M_i$ .

Since the  $i^{\text{th}}$  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  $i^{\text{th}}$  head) can never be in the same combination of values at two different steps. As a result,  $M$  cannot run for more than  $|Q| \cdot (n + 2)$  steps, where  $n$  is the length of the input string.  $\square$

We have proven the following theorem.

**Theorem 9.**  $\mathcal{L}(2\text{nfa}(*), \text{linear-time}) \subseteq 1\text{IP}_*(\infty, \text{cons}, \text{cons})$ .

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

$$\begin{aligned} \text{EQ} &= \{ 0^i 1^i \mid i \in \mathbb{N} \} \\ \text{PAL} &= \{ x \mid x \text{ is the reverse of itself} \} \\ \text{MIXEDEQ} &= \{ x \mid x \in \{0, 1\}^* \text{ and contains equally many 0's and 1's} \} \\ \text{CERT} &= \{ x_1 \cdots x_l \# x_1^+ \cdots x_l^+ \mid l > 0 \text{ and } x_1, \dots, x_l \in \{0, 1\} \} \end{aligned}$$

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

#### 4. Towards tighter bounds

Having determined that  $\mathcal{L}(2\text{nfa}(*), \text{linear-time}) \subseteq 1\text{IP}_*(\infty, \text{cons}, \text{cons}) \subseteq \text{NL}$ , 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  $1\text{IP}_*(\infty, \text{cons}, \text{cons}) = 1\text{IP}(\infty, \text{cons}, \text{cons})$ ) would be to show that NL equals  $\mathcal{L}(2\text{nfa}(*), \text{linear-time})$ , i.e. that any  $2\text{nfa}(k)$  has a linear-time counterpart recognizing the same language. This, however, is a difficult open question [12]. As a matter of fact, there are several examples of famous languages in NL, e.g.

$$\text{PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a path from node } s \text{ to node } t \},$$

for which we have not been able to construct  $2\text{nfa}(k)$ 's with a safe head, and we conjecture that  $\mathcal{L}(2\text{nfa}(*), \text{linear-time}) \neq \text{NL}$ .

We will now show that  $\mathcal{L}(2\text{nfa}(*), \text{linear-time})$  is contained in a subset of NL corresponding to a tighter time restriction of  $O(n^2/\log(n))$  on the underlying non-deterministic Turing machine. We will use the notation  $\text{NTISP}(f(n), g(n))$  for the class of languages that can be verified by a TM that uses  $O(f(n))$  time and  $O(g(n))$  space, simultaneously. For motivation, recall that logarithmic-space TM's require  $\Omega(n^2/\log(n))$  time for recognizing the palindromes language [21, 22, 23], which is easily recognized by a linear-time  $2\text{dfa}(2)$ .

**Theorem 10.**  $\mathcal{L}(2\text{nfa}(*), \text{linear-time}) \subseteq \text{NTISP}(n^2/\log(n), \log n)$ .

*Proof idea.* Given a  $2\text{nfa}(k)$   $M$  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 to 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  $\log(n)/2^{\text{th}}$  cell to indicate the *middle* of the caches.

*Proof.* Let  $M = (Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$  be a  $2\text{nfa}(k)$  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,  $\kappa_1, \dots, \kappa_k$ , with their least significant digit on their left end;
- $k$  caches of input excerpts of  $\log(n)$  length,  $\eta_1, \dots, \eta_k$ ; and
- a mark on the  $\log(n)/2^{\text{th}}$  cell to indicate the middle.

The work tape alphabet  $\Gamma = \Gamma_\kappa^k \times \Gamma_\eta^k \times \{\triangleright, \sqcup\}$  allows  $N$  to encode this information, where

- $\Gamma_\kappa = \{0, 1, \sqcup\}$  to represent each  $\kappa_i$  and
- $\Gamma_\eta = \Sigma_\diamond \sqcup \check{\Sigma}_\diamond$  to represent each cache, where
  - $\Sigma_\diamond = \Sigma \sqcup \{\triangleright, \triangleleft, \#, \sqcup\}$  and
  - $\check{\Sigma}_\diamond$  is a clone of  $\Sigma_\diamond$ , containing “marked” versions of all  $\Sigma_\diamond$ 's symbols.

Cells of the work tape are initialized with  $\sqcup^{2k+1}$  symbols. The algorithm of  $N$  is as follows:



$N$  = “On input  $x$  of length  $n$ :

1. Write 0 to each  $\kappa_i$ .
2. Write  $\# \triangleright x_1 \cdots x_{\log(n)} \#$  to each  $\eta_i$ .
3. Write  $\bowtie$  to the  $\log(n)/2^{\text{th}}$  cell of the last track.
4. Let  $q = q_0$ . Repeat the following until  $q = q_{\text{acc}}$ :
5. Scan the caches. Note the marked symbol in each  $\eta_i$  as  $y_i$  via state transitions.
6. Guess a  $(r, d) \in \delta(q, y_1 \cdots y_k)$ . *Reject* if the set is empty, or  $r = q_{\text{rej}}$ .
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:
9. Clear the mark on  $\#$  of  $\eta_i$ .
10. Go to  $\kappa_i^{\text{th}}$  cell on the input.
11. Go to middle of  $\eta_i$  on the work tape.
12. Move both tape heads left until the left end of  $\eta_i$  is reached.
13. Copy  $\log(n)$  symbols from the input to between the  $\#$  symbols of  $\eta_i$ .
14. Move both tape heads left until the middle of  $\eta_i$  is reached.
15. Mark the middle symbol on  $\eta_i$ .
16. Set  $\kappa_i$  to the input head’s position index.
17. Update  $q$  as  $r$ .
18. *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 it takes linear time by a standard result of amortized analysis. Only the stages that take a constant number 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 mark the  $\log(n)^{\text{th}}$  cell on the caches. After putting  $\#$  on both ends, copying  $x_1 \cdots x_{\log(n)}$  in between them takes  $\log(n)$  more steps. Stage 3 can be performed in  $O(\log^2 n)$  steps by putting  $\bowtie$  symbols to both ends (aligned with the  $\#$  symbols) 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 10 and 16 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 since 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 moves to the middle with every re-cache, a subsequent re-cache cannot 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 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  $N$ .  $\square$

It is not known whether NL contains any language that is not a member of  $\text{NTISP}(n^2/\log(n), \log n)$ .

If  $\text{IIP}_*(\infty, \text{cons}, \text{cons})$  is indeed a proper subset of  $\text{IIP}(\infty, \text{cons}, \text{cons})$ , studying the effects of imposing an additional time-related bound on the verifier may be worthwhile in the search for a characterization. We conclude this section 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 that flips at most  $r$  coins in a private-coin one-way IPS for the language  $A$ . If some string  $x \notin A$  of length  $n$  can be paired with some certificate  $c(x)$  that causes  $V$  to run for  $\omega(n^{2^{r-1}})$  steps with probability 1, then  $V$  has error at least  $1/2$ .*

*Proof.* Let  $V$  be a 2pfa as described above. By an idea introduced in [12], 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  $i^{\text{th}}$  “coin flip” has the outcome  $z_i$ . Construct a 2pfa verifier  $V'$  that flips  $r$  coins at the beginning of its execution and obtains the  $r$ -bit random string  $z$ .  $V'$  then passes control to  $V_z$ .

Verifiers  $V$  and  $V'$  have the same behavior whenever their random bits are the same. Therefore, they are equivalent.

Each  $V_z$  has  $\Theta(n)$  different configurations, where  $n$  denotes the length of the input string. Similarly, any collection of  $2^{r-1}$  distinct  $V_z$  has  $\Theta(n^{2^{r-1}})$  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  $\omega(n^{2^{r-1}})$  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 forever 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  $1/2$ .  $\square$

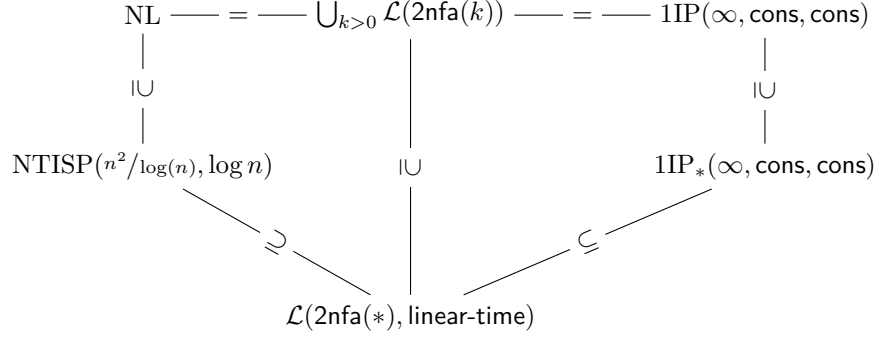


Figure 4: Inclusion diagram of the language classes covered.

## 5. Open questions

For an overview of our results, we present the inclusion diagram in Figure 4. The equalities on the left- and right-hand sides of the diagram were shown in [15] and [12], respectively. We conclude with a list of open questions.

- Is there a language in NL, or even in  $\text{NTISP}(n^2/\log(n), \log n)$ , requiring any  $2\text{nfa}(k)$  recognizing it to have a super-linear runtime?
- Is there a language in NL that cannot be recognized by any log-space NTM running in  $O(n^2/\log(n))$  time?
- Is there a language verified by some constant-space, constant-randomness machine, but not by one with smaller strong error? Is it possible to build such a verifier for any language in NL and for any desired positive error bound?
- Is it possible to construct a linear-time  $2\text{nfa}(k)$  for every language that has verifiers using constant space and randomness for any desired positive strong error?

## Acknowledgments

We thank Neal E. Young for the algorithm in the proof of Lemma 5. We are grateful to Martin Kutrib for providing us with an outline of the proof of Theorem 10. We also thank Ryan O'Donnell and Ryan Williams for their helpful answers to our questions, and the anonymous referees 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. Condon, R. J. Lipton, On the complexity of space bounded interactive proofs, in: *30th Annual Symposium on Foundations of Computer Science*, 1989, pp. 462–467.
  - [4] C. Dwork, L. Stockmeyer, Finite state verifiers I: The power of interaction, *J. ACM* 39 (4) (1992) 800–828.
  - [5] H. Nishimura, T. Yamakami, An application of quantum finite automata to interactive proof systems, *Journal of Computer and System Sciences* 75 (4) (2009) 255–269.
  - [6] H. Nishimura, T. Yamakami, Interactive proofs with quantum finite automata, *Theoretical Computer Science* 568 (2015) 1–18.
  - [7] A. Yakaryılmaz, Public qubits versus private coins, in: *Workshop on Quantum and Classical Complexity*, University of Latvia Press, Riga, 2013, pp. 45–60, ECCC:TR12-130.
  - [8] S. Zheng, D. Qiu, J. Gruska, Power of the interactive proof systems with verifiers modeled by semi-quantum two-way finite automata, *Information and Computation* 241 (2015) 197–214.
  - [9] U. Feige, A. Shamir, Multi-oracle interactive protocols with constant space verifiers, *Journal of Computer and System Sciences* 44 (2) (1992) 259–271.
  - [10] H. G. Demirci, A. C. C. Say, A. Yakaryılmaz, The complexity of debate checking, *Theory of Computing Systems* 57 (1) (2015) 36–80.
  - [11] A. Yakaryılmaz, A. C. C. Say, H. G. Demirci, Debates with small transparent quantum verifiers, *International Journal of Foundations of Computer Science* 27 (02) (2016) 283–300.
  - [12] A. C. C. Say, A. Yakaryılmaz, Finite state verifiers with constant randomness, *Logical Methods in Computer Science* 10 (3) (Aug. 2014).
  - [13] M. Sipser, *Introduction to the Theory of Computation*, Cengage Learning, 2012.
  - [14] M. Holzer, M. Kutrib, A. Malcher, Complexity of multi-head finite automata: Origins and directions, *Theoretical Computer Science* 412 (1-2) (2011) 83–96.

- [15] J. Hartmanis, On non-determinacy in simple computing devices, *Acta Informatica* 1 (4) (1972) 336–344.
- [16] B. Monien, Two-way multihead automata over a one-letter alphabet, *RAIRO. Inform. théor.* 14 (1) (1980) 67–82.
- [17] R. E. Ladner, R. J. Lipton, L. J. Stockmeyer, Alternating pushdown automata, in: *Proceedings of 19th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, 1978, pp. 92–106.
- [18] 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.
- [19] M. Holzer, M. Kutrib, Descriptive and computational complexity of finite automata—A survey, *Information and Computation* 209 (3) (2011) 456–470.
- [20] 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.
- [21] A. Cobham, Time and memory capacity bounds for machines which recognize squares or palindromes, *IBM Res. Rep.* RC-1621 (1966).
- [22] 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.
- [23] P. Dúriš, Z. Galil, A time-space tradeoff for language recognition, *Mathematical systems theory* 17 (1) (1984) 3–12.