CONSTANT-SPACE, CONSTANT-RANDOMNESS VERIFIERS WITH ARBITRARILY SMALL STRONG ERROR

by

Mehmet Utkan Gezer

B.S., Computer Engineering, Boğaziçi University, 2018

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2020

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Professor A. C. Cem Say, for the thesis problem he offered, and his guidance throughout. Thanks to his concise lexicon, every subproblem we have encountered has been much more understandable and tractable. His clear thinking, evident from his wording, is a gift that I wish to inherit.

I am grateful to Professor Martin Kutrib, Professor Emeritus Neal E. Young, Professor Ryan O'Donnell, and Professor Ryan Williams for their helpful answers to our questions. I again thank Professor A. C. Cem Say for making the contact with most of these names.

I thank Professor Haluk O. Bingöl and Associate Professor Tolga Ovatman, my thesis committee members, for their careful review of this thesis.

I would like to thank Instructor Suzan Üsküdarlı for being a great motivator for work, both passively by working hard in her office until late hours, and actively with her encouraging comments. I thank my colleague Fatih Mehmet Atak for his valuable ideas in our research meetings. I thank my friend and colleague Gökçehan Kara for that he patiently listened to my exploratory thoughts on several occasions, and has always been there for support.

I am grateful to the friendly people of the Boğaziçi University. I believe that timely breaks with friends are key to success.

Finally, I would like to express my gratitude towards every individual who has made choices for a habitable and sustainable future of our generation, and of future generations to come.

ABSTRACT

CONSTANT-SPACE, CONSTANT-RANDOMNESS VERIFIERS WITH ARBITRARILY SMALL STRONG ERROR

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 a strong error bound strictly less than $1/2$ is precisely NL, but their construction yields verifiers with strong 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 strong 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 strong error ε for every language that is recognizable by a linear-time multi-head nondeterministic finite automaton ($\mathsf{2nfa}(k)$). 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 $\mathsf{2nfa}(k)$'s to simultaneous time-space complexity classes defined in terms of Turing machines.

ÖZET

SONLU HAFIZA VE SONLU RASTGELELİK KULLANAN, ALABİLDİĞİNE KÜÇÜK KATI TİP HATALI DOĞRULAYICILAR

İşbu çalışmada, girdinin boyutundan bağımsız, sabit bir miktarda yazı-tura atma hakkı tanınan sonlu hal hesaplayıcılarının, aidiyet ispatlarını doğrulama konusunda yapabileceklerinin sınırlarını inceliyoruz. Cem Say ve Abuzer Yakaryılmaz'ın önceki bir çalışmada ispatladığı üzere, bu nitelikteki hesaplayıcıların katı tipte $1/2$'den az hata yaparaktan doğrulayabildikleri dillerin kümesi, tam da NL sınıfına denk gelmektedir. Fakat bu ispatta sunulan doğrulayıcıların katı tip hatası, NL'nin büyük çoğunluğu için $1/2$'ye oldukça yakın çıkmakta. Bu tezde bu zayıf türden hesaplayıcıların alabildiğine küçük katı tip hata ile doğrulayabildiği dilleri, NL sınıfının bir alt kümesi olarak belirliyoruz. Pozitif herhangi bir ε değeri için, doğrusal zamanda çalışan çok kafalı bir belirsiz sonlu hal hesaplayıcısı ($\mathsf{2nfa}(k)$) ile tanınabilen herhangi bir dili tanıyan, sonlu hafıza ve sonlu rastgelelik kullanan ve katı tip hatası en fazla ε olan bir doğrulayıcı inşa eden bir yöntem sunuyoruz. Bu yöntemi tüm NL'ye genellemenin neden zor olduğunu tartışıyor, doğrusal zamanda çalışan $\mathsf{2nfa}(k)$'lerin dil tanıma gücünün, aynı anda zaman ve de hafıza sınırlarına tabi Turing makineleri üzerinden tanımlanmış karmaşıklık sınıfları ile olan alakasını hayli sıkı bir biçimde kuruyoruz.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

$\mathcal{L}(\mathsf{A})$	Class of languages recognized by the automata of type A
$\mathbb{N}$	Set of natural numbers including 0
$\mathcal{P}(S)$	Set of all subsets of the set S
$\mathbb{Z}_+$	Set of positive integers
ε	Rate of error
λ	Identity element of string concatenation, the empty string
$A \sqcup B$	Union of sets A and B, also asserting that A and B are disjoint
$\langle O_1, \ldots, O_k \rangle$	Encoding of objects $O_1, \ldots, O_k$ in the alphabet of context
$\lvert\sigma\rvert$	Number of elements in the set or sequence σ
σ_i	i^{th} element of a sequence σ
$\sigma \circ \tau$ or $\sigma\tau$	Concatenation of the (possibly unit) sequences σ and τ
$\triangleright$	Symbol marking the left end of an input string
$\triangleleft$	Symbol marking the right end of an input string

LIST OF ACRONYMS/ABBREVIATIONS

1dfa	One-way Deterministic Finite Automaton
1nfa	One-way Nondeterministic Finite Automaton
2dfa	Two-way Deterministic Finite Automaton
2dfa(k)	Two-way k-head Deterministic Finite Automaton
2nfa	Two-way Nondeterministic Finite Automaton
2nfa(k)	Two-way k-head Nondeterministic Finite Automaton
2pfa	Two-way Probabilistic Finite Automaton
IPS	Interactive Proof System
NTM	Nondeterministic Turing Machine
PTM	Probabilistic Turing Machine
TM	(Deterministic) Turing Machine

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 trade-offs 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 size [1]. 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 [2] that the class of languages that could be verified by these machines within a strong error bound strictly less than $1/2$ is precisely NL, i.e. languages with deterministic logarithmic-space verifiers.

The construction given in [2] could exhibit a constant-randomness verifier operating within strong error ε 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 thesis, we characterize a subset of NL for which verification with arbitrarily low strong 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 strong error ε for every language that is recognizable by a linear-time multi-head nondeterministic finite automaton ($\mathsf{2nfa}(k)$). 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 $\mathsf{2nfa}(k)$'s to simultaneous time-space complexity classes defined in terms of Turing machines. We conclude with a list of open questions.

This thesis has yielded the publication [3], and the submitted preprint [4].

2. PRELIMINARIES

An *alphabet* is a finite and nonempty set of symbols. A *string* is a sequence of symbols, finite unless otherwise stated. A *language* over an alphabet is a set of strings that consist of the symbols from that alphabet.

A *decision problem* is a function that produces a yes-or-no question, given an input string. Each of those questions are an *instance* of the decision problem. A decision problem is *decidable*, if and only if the answer to any instance can be computed given enough, but finite, time and space. A decision problem is *recognizable*, if and only if whenever the answer is "yes" to an instance, the answer can be computed in finite amount of time and space. A decision problem is commonly formulated as a set of strings satisfying a predicate; those strings for which the answer is "yes".

The reader is assumed to be familiar with the standard concepts of automata theory, and Turing machines [5]. We will use the terms *algorithm* and the Turing machine (TM) interchangeably. We will call the functions that map the languages to algorithms recognizing them, a *method*.

2.1. Nondeterminism

A nondeterministic TM (NTM) is one that has alternative paths of execution to follow. An NTM accepts a string, if and only if there exists an execution path that accepts the string. An NTM decides its language, if and only if it halts on every execution path running on any input.

The means of execution of an NTM may be imagined as, among others, either one of the following:

(i) A TM forking into parallel universes in every transition by the alternatives.

(ii) A TM provided with a *certificate* to consult for the alternatives to take.

(iii) A TM that has an internal guide allowing it to *guess* from the alternatives.

The NTM accepts an string, if and only if a parallel universe, a certificate (i.e. a *proof of membership*), or a sequence of guesses exists, respectively, in/by which the imagined TM accepts the string. The NTM decides its language, if and only if for any input string, and in all the parallel universes, by all the certificates, or sequences of guesses, respectively, the imagined TM halts eventually.

2.2. Time and Space Complexity Classes

TIME($f(n)$) and NTIME($f(n)$) denote the classes of all languages that can respectively be recognized by a TM and an NTM in $O(f(n))$ steps on every execution path for inputs of length n. SPACE($f(n)$) and NSPACE($f(n)$) similarly denote the classes of all languages recognized respectively by a TM and an NTM that uses $O(f(n))$ space (i.e. tape cells) at most. The following denotations are standard:

$$\mathrm{P} = \bigcup_k \mathrm{TIME}\left(n^k\right)$$
$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}\left(n^k\right)$$
$$\mathrm{L} = \mathrm{SPACE}(\log n)$$
$$\mathrm{NL} = \mathrm{NSPACE}(\log n)$$

Finally, TISP($f(n), g(n)$) and NTISP($f(n), g(n)$) denote the class of languages that can be recognized respectively by a TM and an NTM that uses $O(f(n))$ time and $O(g(n))$ space, simultaneously.

2.3. Multi-head Finite Automata

A k-head nondeterministic finite automaton, denoted $\mathsf{2nfa}(k)$[1] , is a 6-tuple consisting of

(i) a finite set of states Q,

(ii) an input alphabet Σ,

(iii) a transition function $\delta\colon Q \times \Gamma^k \to \mathcal{P}\Big(Q \times \Delta^k\Big)$, where;

- $\Gamma = \Sigma \sqcup \{ \rhd, \lhd \}$ is the tape alphabet, where $\rhd$ and $\lhd$ are respectively the left and right end markers
- $\Delta = \{ -1, 0, 1 \}$ is the set of head movements, where -1 and 1 respectively indicate moving left and right, and 0 indicates staying put,

(iv) an initial state q_0,

(v) an accept state q_{acc}, and

(vi) a reject state q_{rej}.

A $\mathsf{2nfa}(k)$ $M = (Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ initially starts from q_0, and with $\rhd x \lhd$ written on its single read-only tape, where $x \in \Sigma^*$ is the input string. All k tape heads are initially on the $\rhd$ symbol. The function δ 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 $\mathsf{2nfa}(k)$ 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 δ, a $\mathsf{2nfa}(k)$ M may have several *computational paths* on the same string. A computational path of M *halts* if it reaches q_{acc} or q_{rej}, or if δ does not offer any steps for M

[1] The 2 in $\mathsf{2nfa}(k)$ is to indicate that they can move their heads in both directions.

to follow. M *accepts* an input string x, if there is a computational path of M running on x that halts on q_{acc}. M *rejects* an input string x, if M running on x halts on a state other than q_{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 given any input string halts on every computational path, M is said to be an *always halting* $\mathsf{2nfa}(k)$.

A k-head deterministic finite automaton, denoted $\mathsf{2dfa}(k)$, differs from a $\mathsf{2nfa}(k)$ in its transition function, which is defined as $\delta\colon Q \times \Gamma^k \to Q \times \Delta$. $\mathsf{2nfa}(1)$'s are simply called finite automata, and are denoted as $\mathsf{2dfa}$'s and $\mathsf{2nfa}$'s for the deterministic and nondeterministic counterparts, respectively. $\mathsf{1dfa}$'s and $\mathsf{1nfa}$'s are the *one-way finite automata*, which are respectively $\mathsf{2dfa}$'s and $\mathsf{2nfa}$'s that cannot move their head to the left.

For $k > 0$, let $\mathcal{L}(\mathsf{2dfa}(k))$ and $\mathcal{L}(\mathsf{2nfa}(k))$ denote the classes of languages recognized by a $\mathsf{2dfa}(k)$ and a $\mathsf{2nfa}(k)$, respectively.

Lemma 2.1. $\text{SPACE}(1) = \mathcal{L}(\mathit{2dfa}(1)) = \mathcal{L}(\mathit{2nfa}(1)) = \text{NSPACE}(1)$. *They are the class of regular languages.*

Proof. A $\mathsf{2nfa}$ with the states set Q can simulate an NTM with the tape alphabet Γ, and c tape cells, by expanding its states set into $Q \times \Gamma^c$. An NTM with a finite tape simulates a $\mathsf{2nfa}$ trivially. Hence, $\mathcal{L}(\mathsf{2nfa}(1)) = \text{NSPACE}(1)$. One can establish $\mathcal{L}(\mathsf{2dfa}(1)) = \text{SPACE}(1)$ the same way.

To show that $\mathcal{L}(\mathsf{2nfa}(1)) = \mathcal{L}(\mathsf{2dfa}(1))$, an equivalent $\mathsf{1dfa}$ and a $\mathsf{1nfa}$ can be obtained for any given $\mathsf{2dfa}$ and $\mathsf{2nfa}$, respectively, by [6]. Finally, [5] shows that both the $\mathsf{1dfa}$ and $\mathsf{1nfa}$ recognize the regular languages exactly. □

Let $\mathcal{L}(\mathsf{2dfa}(\mathbb{Z}_+))$ and $\mathcal{L}(\mathsf{2nfa}(\mathbb{Z}_+))$ denote the classes of languages that respectively has a $\mathsf{2dfa}(k)$ and a $\mathsf{2nfa}(k)$ recognizing them for some k. In other words;

$$\mathcal{L}(\mathsf{2dfa}(\mathbb{Z}_+)) = \bigcup_{k>0} \mathcal{L}(\mathsf{2dfa}(k)),$$
$$\mathcal{L}(\mathsf{2nfa}(\mathbb{Z}_+)) = \bigcup_{k>0} \mathcal{L}(\mathsf{2nfa}(k)).$$

Lemma 2.2. *Deterministic and nondeterministic multi-head finite automata are respectively equivalent to deterministic and nondeterministic logarithmic space TM's in terms of language recognition power [7]. Put formally;*

$$\mathcal{L}(\mathit{2dfa}(\mathbb{Z}_+)) = \mathrm{L}$$
$$\mathcal{L}(\mathit{2nfa}(\mathbb{Z}_+)) = \mathrm{NL}$$

Lemma 2.3. *Let $A \in \mathrm{L}$ and $B \in \mathrm{NL}$, recognized by a $\mathit{2dfa}(k)$ and a $\mathit{2nfa}(h)$, respectively. Then;*

$$A \text{ is not regular} \implies k \geq 2$$
$$B \text{ is not regular} \implies h \geq 2$$

Proof. The proof is by contradiction. If k was 1, then A would be regular. If h was 1, then B would be regular. □

Lemma 2.4. *The languages in* L *and* NL *are organized in a hierarchy based on the number of heads of the deterministic and the nondeterministic automata recognizing them, respectively. Formally, the following are true for any $k > 0$ [8]:*

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

For $k > 0$, let $\mathcal{L}(\mathsf{2dfa}(k), f(n))$ and $\mathcal{L}(\mathsf{2nfa}(k), f(n))$ denote the classes of languages that are recognized respectively by a $\mathsf{2dfa}(k)$ and a $\mathsf{2nfa}(k)$ 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 $\mathcal{L}(\mathsf{2dfa}(\mathbb{Z}_+), f(n))$ and $\mathcal{L}(\mathsf{2nfa}(\mathbb{Z}_+), f(n))$ denote the class of languages that are recognized respectively by the deterministic and nondeterministic multi-head finite automata with any number of heads, and running in $O(f(n))$ time. We use $\mathsf{linear\text{-}time}$ designation instead of $f(n) = n$.

Lemma 2.5. *The following are true for any $k > 0$:*

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

Proof. Let M be any $\mathsf{2nfa}(k)$ 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 $\mathsf{2nfa}(2k)$ 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 $|Q|^{\text{th}}$ 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 the right. If c_k attempts to move past the right end, M' rejects.

The $\mathsf{2nfa}(2k)$ M' recognizes the same language as M, but within the time limit of $O\Big(n^k\Big)$. The argument is similar for the deterministic version of the statement. □

Lemmas 2.4 and 2.5 can be combined into the following useful fact.

Corollary 2.6. *For every $A \in$ NL, there is a minimum number k_A, such that there exists an always halting $\mathsf{2nfa}(k_A)$ recognizing A, but not an always halting $\mathsf{2nfa}(h)$ with $h < k_A$.*

Proof. By Lemma 2.4, $A \in \mathcal{L}(\mathsf{2nfa}(k))$, but $A \notin \mathcal{L}(\mathsf{2nfa}(k-1))$ for some k. The existence of an always halting $\mathsf{2nfa}(k_A)$ recognizing A for a minimum k_A between k and $2k$ is guaranteed by Lemmas 2.4 and 2.5, respectively. □

Lemma 2.7. *$HALTING_{\mathsf{2nfa}} = \{ \langle M \rangle \mid M$ is an always halting $\mathsf{2nfa} \}$ is decidable.*

Proof[2]*.* 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* 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.

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

$D =$ "On input $\langle M \rangle$, where M is an 2nfa, and Σ 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 Σ^*. If it does, *accept*. Otherwise, *reject*."

By its construction, M'_{2afa} recognizes Σ^* 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 [9] and [5]. 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]We thank Neal E. Young, who introduced us the algorithm for this proof.

2.4. 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:

(i) a finite set of states $Q = Q_d \sqcup Q_r$, where;
   - Q_d is the set of deterministic states,[3] and
   - Q_r is the set of coin-tossing states,[3]

(ii) an input alphabet Σ,

(iii) a transition function overloaded as deterministic δ_d and coin-tossing δ_r;
   - $\delta_d \colon Q_d \times \Gamma \to Q \times \Delta$, where Γ and Δ are as defined for the 2nfa(k)'s, and
   - $\delta_r \colon Q_r \times \Gamma \times R \to Q \times \Delta$, where $R = \{ 0, 1 \}$ is a random bit provided by a "coin toss",

(iv) an initial state q_0,

(v) an accept state q_{acc}, and

(vi) a reject state q_{rej}.

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

[3] The letters d and r stand for *deterministic* and *random*, respectively.

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

(i) *Failing to accept* – rejecting or looping indefinitely given a member input

(ii) *Failing to reject* – accepting or looping indefinitely given a nonmember input

(iii) *False accept* – accepting given a nonmember input

2.5. Interactive Proof Systems

Our definitions of interactive proof systems (IPSes) are based on [10].

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. There are 2 specifications for an IPS; being one-way or two-way, and being private-coin or public-coin.

We will first describe the most general case; the *public-coin two-way IPS*. Let Σ and Λ denote respectively the input and the communication alphabets. The PTM verifier V executes on a given input string $x \in \Sigma*$ as usual, except for when it is interacting with the prover P via the shared communication cell. For V to initiate an interaction, a subset Q_c of the its states are designated as the communication states. Whenever V enters a state $q \in Q_c$, V communicates with the prover P by the following protocol:

(i) V writes $\kappa = \gamma_V(q)$ to the communication cell, where $\gamma_V \colon Q_c \to \Lambda$ is V's communication function.

(ii) P writes $\varkappa = \gamma_P(x, y, z)$ to the communication cell, where $\gamma_P \colon \Sigma^* \times \Lambda^* \times \{\, 0, 1 \,\}^* \to \Lambda$ is P's communication function, y is the entire communication transcript until and including κ, and z is the string of random bits V has generated so far.

(iii) V transitions into the state $\delta_c(q, \varkappa)$, where $\delta_c \colon Q_c \times \Lambda \to Q$ is V's communication transition function, that is a further overload to V's transition function.

The language of a verifier V is the set of all strings that a prover P can make V accept with over $1/2$ probability.

Compared to a public-coin two-way IPS;

- a *public-coin one-way IPS* does not have stage (i) in its protocol,
- a *private-coin two-way IPS* does not provide γ_P with z at stage (ii), the string of random bits V has generated, and
- a *private-coin one-way IPS* neither has stage (i), nor provides γ_P with z.

In the case of a private-coin one-way IPS, γ_P takes only the input string x, and its previous outputs as input. As such, the infinite string of symbols that γ_P is to communicate is predetermined by x. By this observation, the prover of a private-coin one-way IPS can be viewed as writing the infinite certificate string $c(x)$ to the verifier V's additional certificate tape with a head that cannot move left, for a given input string x, and the certificate function $c \colon \Sigma^* \to \Lambda^\infty$, defined as follows:

$$c(x)_1 = \gamma_P(x, \lambda)$$
$$c(x)_i = \gamma_P(x, c(x)_1 \cdots c(x)_{i-1})$$

This thesis will make use of this simplification.

A PTM verifier V naturally errs. Let A be the language of V. The *strong error of* V, denoted $\varepsilon_{\mathrm{s}}(V)$, is defined as the minimum value satisfying both of the following, given any such input string x:

- V, paired with some prover P, fails to accept $x \in A$ with a probability less than $\varepsilon_{\mathrm{s}}(V)$.

- V, paired with any prover P, fails to reject $x \notin A$ with a probability less than $\varepsilon_{\mathrm{s}}(V)$.

The *weak error of* V, denoted $\varepsilon_{\mathrm{w}}(V)$, is similarly defined for the member inputs, with the only difference that it considers the probability of false accepts, rather than the failure to rejects. In that regard, $\varepsilon_{\mathrm{w}}(V)$, unlike $\varepsilon_{\mathrm{s}}(V)$, does not count looping on nonmembers as an error.

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

Let $1\mathrm{IP}_{\varepsilon}(t(n), s(n), r(n))$ be the class of languages that have PTM verifiers with a strong error at most ε ($\varepsilon < 1/2$) 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 cons, log, poly, or exp to describe a constant, logarithmic, polynomial, or exponential limit, respectively, in terms of the input length. We write 0 and ∞ to describe that a resource is unavailable and unlimited, respectively. Furthermore, let;

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

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

We note the following known results [1, 2, 11]:

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

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

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

3. LINEAR-TIME 2nfa(k)'S AND VERIFICATION WITH SMALL STRONG ERROR

In [2], 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 μ_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 2.2 and 2.5, 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 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 $A \in$ NL which can be recognized by an always halting 2nfa(k) M_A, the verifier of μ_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. Therefore, the weak error of $\mu_1(A)$ 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 $\mu_1(A)$ cannot 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 nonzero 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 $1/k$.[1] Therefore, $\varepsilon_{\mathrm{w}}(\mu_1(A))$ is at most $((k-1)/k)^m$. 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 $\mathsf{2nfa}(k)$ M_A recognizing $A \in \mathrm{NL}$ 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 of $\mu_1(A)$ is therefore at most $(k-1)/k$. This upper bound to $\varepsilon_{\mathrm{s}}(\mu_1(A))$ 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 2.6. Moreover, by Lemma 2.3, $(k_A-1)/k_A$ is at least $1/2$ for any nonregular language.

[1] 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.

Say and Yakaryılmaz also propose the method μ_2, a slightly modified version of μ_1 that produces verifiers with errors less than $1/2$, albeit barely so. Let $A \in \mathrm{NL}$, and M_A be an always halting $\mathsf{2nfa}(k)$ recognizing A, for some k. Regardless of the input string, the verifier $\mu_2(A)$ rejects at the very beginning with a probability $(k-1)/2k$, using $\lceil \log k \rceil + 1$ coins. Then, it continues just like $\mu_1(A)$. The bounds for the strong error of $\mu_2(A)$ are as follows:

$$\frac{k-1}{2k} \leq \varepsilon_{\mathrm{s}}(\mu_2(A)) \leq \frac{k^2-1}{2k^2}$$

Compared $\mu_1(A)$, $\mu_2(A)$ trades one-sidedness (i.e. 0 probability of failing to accept) of the error for a strong error below $1/2$.[2]

3.1. Safe and Risky Heads

How much of NL may yet fit into $1\mathrm{IP}_*(\infty, \mathsf{cons}, \mathsf{cons})$? Method μ_1 was our starting point in working towards a lower bound for $1\mathrm{IP}_*(\infty, \mathsf{cons}, \mathsf{cons})$.

Let M_A be the $\mathsf{2nfa}(k)$ that $\mu_1(A)$ uses to verify $A \in \mathrm{NL}$. 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 $\mathsf{2nfa}(k)$ with the transition function $\delta\colon Q \times \Gamma^k \to \mathcal{P}\big(Q \times \Delta^k\big)$. For i between 1 and k, let M_i be a $\mathsf{2nfa}$ with the transition function $\delta_i\colon Q \times \Gamma \to \mathcal{P}(Q \times \Delta)$ defined as follows:

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

[2] If one desires to reduce the probability of looping without sacrificing from one-sidedness, a verifier V could be designed to accept its inputs with probability p, and pass the execution to $\mu_1(A)$ otherwise. The probability of looping with V will be reduced by a factor of $1 - p$ compared to that of $\mu_1(A)$. Although V's strong error will be even worse than that of $\mu_1(A)$, it will still be one-sided.

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 μ_1. Just like the verifier of μ_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 μ_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 μ_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 $\Sigma = \{ \mathtt{0}, \mathtt{1} \}$, and $EQ1 = \{ \mathtt{0}^i\mathtt{1}^i \mid i \in \mathbb{N} \}$. Figure 3.1 depicts a directed graph from the states of 2nfa(2) $M_{EQ1} = (Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$. Whenever $(r, d) \in \delta(q, x)$, an arc is drawn from q to r with the label $x \to d$. One additional arc arrives at q_0 from nowhere. Such pictorial representation of an automaton is called its *state diagram*.

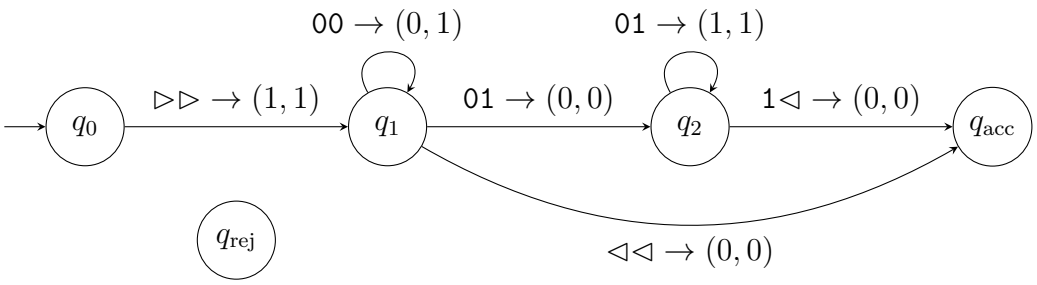

Figure 3.1. State diagram of 2nfa(2) M_{EQ1}.

After passing the left end marker, in state q_1, the first head of M_{EQ1} stays put until the second one goes past beyond all the 0's and reads a 1. Then, in state q_2, they simultaneously and respectively pass 0's and 1's, until they respectively reach to

a 1 and the right end marker. This journey is possible for M, if and only if the input string is a member of *EQ1*.

Consider the 2nfa M_{EQ1_1}, associated with the first head of M_{EQ1}. See Figure 3.2. Is there any input string on which M_{EQ1_1} may loop? Yes; M_{EQ1_1} indeed has an execution that runs forever on any input string that begins with a 0, via the self-loop at q_1.

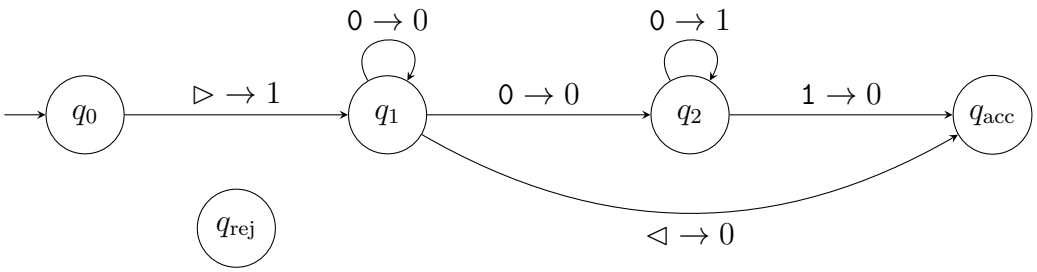

Figure 3.2. State diagram of 2nfa M_{EQ1_1}.

Definition 1 asserts that this evidence is equivalent to say that the first head of M_{EQ1} is risky. Indeed, $\mu_1(EQ1)$ that simulates M_{EQ1}, and has chosen the first head for verification, loops forever on q_1 when paired with a particular certificate that continuously and indefinitely reports 0's as the readings of the both heads.

Lemma 3.1. *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 $M_i \in HALTING_{\text{2nfa}}$ by the algorithm in Lemma 2.7. □

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 $\varepsilon_{\text{s}}(\mu_1(A)) \leq (^{(k-1)}/_k)^m$.

3.2. 2nfa(k)'s with a Safe Head and Verification with Small Strong Error

The distinction of safe and risky heads has been the key to our improvement to the method μ_1. Method μ_3, to be introduced in the proof of the following lemma, is able to produce verifiers with a strong error bound equaling any desired nonzero constant, for a subset of languages in NL.

Lemma 3.2. *Let $A \in$ NL. If there exists an always halting 2nfa(k) with at least one safe head recognizing A, then $A \in 1\mathrm{IP}_*(\infty, \mathsf{cons}, \mathsf{cons})$.*

Proof idea. The method μ_3 in the proof will construct verifiers similar to those of μ_1, except for a key difference. Given a language $A \in$ 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 nonzero 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 nonzero, 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 nonzero value by increasing m.

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

Let $s = \lceil \log k \rceil$. Regarding the risky heads, let H_R be the set of their indices, k_R be their count, and if $k_R > 0$, let $\nu_R \colon \{\, 0, 1 \,\}^s \to H_R$ be any total function, such that each i in H_R is mapped to by ν_R exactly $\lfloor 2^s / k_R \rfloor$ or $\lceil 2^s / k_R \rceil$ many times. Regarding the

safe heads, let H_S, k_S, and ν_S be defined analogously. The following parameters will be controlling the strong 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 its original position.
3. Choose i from $\{ 1, \ldots, k \}$ 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. Let $u \in \{ 0, 1 \}^s$ be the outcomes.
6. 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 $q = q_{\mathrm{acc}}$:
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 $q' = q_{\mathrm{rej}}$, *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 nonmember 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 δ and the reports of the certificate. At any given round, $\mu_3(A)$ can either be led to the state q_{acc} and pass, to the state q_{rej} and reject, to follow a loop of transitions availed by δ 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 $\mathsf{2nfa}(k)$ 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 δ, 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 8. 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_R, and the definition of ν_S and ν_R, every head of M_A has a nonzero 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

$$(1-p)^m. \tag{3.1}$$

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

$$(1-p)^{m'-1} \cdot P_R. \tag{3.2}$$

The probability that $\mu_3(A)$ falsely accepts (Equation (3.1)) can be reduced arbitrarily to any nonzero value by increasing m. The probability that it loops on a nonmember input (Equation (3.2)) 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 $A \in$ NL that can be recognized by a $\mathsf{2nfa}(k)$ with at least one safe head, and for any strong 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 ε, and is constant with respect to the input string.

3.3. Linear-time $\mathsf{2nfa}(k)$'s and Safe Heads

Lemma 3.3. *Given a language A, the following statements are equivalent:*

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

(ii) *A is recognized by a $\mathit{2nfa}(h)$ with at least one safe head.*

The proof of Lemma 3.3 will be in two parts.

Proof of (i) $\implies$ (ii). Given $A \in \mathcal{L}(\mathsf{2nfa}(k), \mathsf{linear\text{-}time})$, for some k, there exists a $\mathsf{2nfa}(k)$ 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 $\mathsf{2nfa}(k+1)$ 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.

The modified machine M' is obtained by installing c replicas of the original machine M, and then rerouting each transition to land on the same terminal state of the next copy. An additional head T is added, which is obliged to read a character other than $\lhd$ in every transition. This head T moves right during every transition emerging from the c^{th} replica of M, and staying put for the rest of the transitions. Where $\Gamma = \Sigma \cup \{ \rhd, \lhd \}$ and $M = (Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, the formal definition of $M' = (Q', \Sigma, \delta', q_0', q_{\text{acc}}', q_{\text{rej}}')$ is as follows:

$$Q' = (Q \times \{ 1, \ldots, c \}) \sqcup \{ q_{\text{acc}}', q_{\text{rej}}' \}$$
$$q_0' = (q_0, 1)$$

For $q \in Q$, $i \in \{ 1, \ldots, c \}$, $y \in \Gamma^k$ and $z \in \Gamma$:

$$\delta'((q,i), yz) = \begin{cases} \{ q_{\text{acc}}' \} & \text{if } q = q_{\text{acc}} \\ \{ q_{\text{rej}}' \} & \text{if } q = q_{\text{rej}} \\ \{ \} & \text{if } z = \lhd \\ \{ ((r, i+1), d \circ 0) \mid (r,d) \in \delta(q,y) \} & \text{if } i < c \\ \{ ((r,1), d \circ 1) \mid (r,d) \in \delta(q,y) \} & \text{if } i = c \end{cases}$$

When multiple pieces of the piecewise definition apply, only the one that comes first is considered. For example, for $c > 1$, $x \in \Gamma^k$, and $\delta'((q_{\text{acc}}, 1), x\lhd)$ satisfying the first, third, and fourth pieces pieces at the same time, the output is $\{ q_{\text{acc}}' \}$.

Consider the 2nfa M'_{k+1} associated with the timer head T of M', as in Definition 1. M'_{k+1} moves its head once to the right in every c^{th} step, unconditionally. If it ever runs that much, M'_{k+1} is guaranteed to read $\triangleleft$ after running for $c \cdot (|x| + 1)$ steps, and halt because of that. Thus, M'_{k+1} is always halting, and head T of M' is safe.

Recall that M never runs for more than $c \cdot |x|$ steps. Therefore M', inheriting its transitions from M, also never runs for more than that, and the safe head T of M' cannot ever reach $\triangleleft$. Thus, M' (a $\mathsf{2nfa}(h)$ with a safe head) recognizes A. □

Proof of (ii) $\implies$ (i). Let M be a $\mathsf{2nfa}(k)$ recognizing A, such that its i^{th} head is a safe head. Let $\delta\colon Q \times \Gamma^k \to \mathcal{P}\Big(Q \times \Delta^k\Big)$ be the transition function of M. Let M_i be the 2nfa 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^{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 δ_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 cannot 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 3.4. $\mathcal{L}(\mathit{2nfa}(\mathbb{Z}_+), \mathsf{linear\text{-}time}) \subseteq 1\mathrm{IP}_*(\infty, \mathsf{cons}, \mathsf{cons})$.

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

$$
\begin{aligned}
\mathit{EQ1} &= \left\{ \mathtt{0}^i\mathtt{1}^i \;\middle|\; i \in \mathbb{N} \right\} \\
\mathit{PAL} &= \{\, x \mid x \text{ is the reverse of itself} \,\} \\
\mathit{EQ2} &= \{\, x \mid x \in \{\, \mathtt{0}, \mathtt{1} \,\}^* \text{ and contains equally many } \mathtt{0}\text{'s and } \mathtt{1}\text{'s} \,\} \\
\mathit{CERT} &= \left\{ x_1 \cdots x_l \# x_1^+ \cdots x_l^+ \;\middle|\; l > 0 \text{ and } x_1, \ldots, x_l \in \{\, \mathtt{0}, \mathtt{1} \,\} \right\}
\end{aligned}
$$

To recognize *EQ1*, $\mathsf{2dfa}(2)$ can move its both heads to the right until they see the first `1`. After moving one of them once to the left, the left and right head move to the left and right end, until they read a symbol other than `0` and `1`, respectively. If they simultaneously read the respective end markers, the machine accepts.

To recognize *PAL*, a $\mathsf{2dfa}(2)$ can move one of its heads twice as fast as the other, and both to the right. When the fast head reaches the right end, it continues moving just as fast, but to the left. When the slow head reaches the right end, the fast head simultaneously reaches the left end, and they both stop. They then move towards the opposite end, checking whether they read the same symbols across. If they do, the machine accepts.

The $\mathsf{2dfa}(2)$'s above do not have any risky heads. We have not been able to find $\mathsf{2nfa}(k)$'s without risky heads that recognize the languages *EQ2* and *CERT*. The following $\mathsf{2dfa}(2)$'s recognizing them have one safe head each.

To recognize *EQ2*, a 2dfa(2) continuously moves its 1st head to the right until the end, and its 2nd head to the left and right respectively for each `0` and `1` read by the 1st head. Whenever the 2nd is reading the left end marker while it is about to move to the left, the movement directions are flipped, and the head moves to the right instead. The machine accepts, if the 2nd head ends up at the left end when the input is read through by the 1st head. Here, the 1st head is safe.

To recognize *CERT*, a 2dfa(2) sends one head to the beginning of the part after the `#` symbol. Then, the two heads move to the right simultaneously, checking whether they read the same symbol. When they read a different symbol, the head on the left stops. The right head, if it is reading the same symbol as the last one, continues to move to the right until it sees a different symbol, and the machine rejects otherwise. At this point, if the left head is on the `#` symbol, the machine accepts if the right head is on the right end marker, and rejects otherwise. Otherwise, the process repeats. Here, the head on the right is safe.

4. DISCUSSION

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

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

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

We will now show that $\mathcal{L}(\mathsf{2nfa}(\mathbb{Z}_+), \mathsf{linear\text{-}time})$ is contained in a subset of NL corresponding to a tighter time restriction of $O(n^2/\log(n))$ on the underlying nondeterministic Turing machine.[3]

Theorem 4.1. $\mathcal{L}(\mathit{2nfa}(\mathbb{Z}_+), \mathsf{linear\text{-}time}) \subseteq \mathrm{NTISP}(n^2/\log(n), \log n)$.

Proof idea[4]. Given a $\mathsf{2nfa}(k)$ 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

[3] Recall that logarithmic-space TM's require $\Omega(n^2/\log(n))$ time for recognizing the palindromes language [12–14], which is easily recognized by a linear-time $\mathsf{2dfa}(2)$.

[4] We thank Martin Kutrib for providing us with an outline of this proof.

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 ${}^{\log(n)}\!/_{2}{}^{\text{th}}$ cell to indicate the *middle* of the caches.

Proof. Let $M = (Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a $\mathsf{2nfa}(k)$ 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 ${}^{\log(n)}\!/_{2}{}^{\text{th}}$ cell to indicate the middle.

The work tape alphabet $\Gamma = \Gamma_\kappa^k \times \Gamma_\eta^k \times \{ \bowtie, \textvisiblespace \}$ allows N to encode those information, where;

- $\Gamma_\kappa = \{ \mathtt{0}, \mathtt{1}, \textvisiblespace \}$ to represent each κ_i
- $\Gamma_\eta = \Sigma_\diamond \cup \dot{\Sigma}_\diamond$ to represent each cache, where;
  - $\Sigma_\diamond = \Sigma \sqcup \{ \triangleright, \triangleleft, \#, \textvisiblespace \}$, and
  - $\dot{\Sigma}_\diamond$ is a clone of $\Sigma_\diamond$, containing "marked" versions of all $\Sigma_\diamond$'s symbols.

Initially, all cells of the work tape contain the symbol $\sqcup^{2k+1}$. The algorithm of N is as follows:

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

1. Write $\mathtt{0}$ to each κ_i.

2. Write $\mathtt{\#}\overset{\blacktriangledown}{\triangleright} x_1 \cdots x_{\log(n)}\mathtt{\#}$ on to each η_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[5] the marked symbol in η_i as y_i.

6. Guess a $(r, d) \in \delta(q, y_1 \cdots y_k)$. *Reject* if the set is empty, or $r = q_{\text{rej}}$.

7. For all i, adjust κ_i, and move the mark on η_i by d_i.

8. *Re-cache* each η_i that has a $\overset{\blacktriangledown}{\mathtt{\#}}$ symbol, as follows:

9. Clear the mark on $\overset{\blacktriangledown}{\mathtt{\#}}$ of η_i.

10. Go to ${\kappa_i}^{\text{th}}$ cell on the input.

11. Go to middle of η_i on the work tape.

12. Move both tape heads left, until the left end of η_i is reached.

13. Copy $\log(n)$ symbols from the input to between the $\mathtt{\#}$ symbols of η_i.

14. Move both tape heads left, until the middle of η_i is reached.

15. Mark the middle symbol on η_i.

16. Set κ_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.

N uses $O(\log n)$ cells on its tape, and that is to keep the caches and counters.

[5] This is done using the states of N, and does not use work tape.

Counting up to n in binary is a common task across this algorithm, and takes linear time, by a standard result of amortized analysis, which will be presented by the end of this proof. 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 marking the $\log(n)^{\text{th}}$ 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 $O\left(\log^2 n\right)$ 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 10 and 16 are the most time consuming substages of a re-cache, involving decrementing of κ_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 substage 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 cost to N is $O(n^2/\log(n))$.

We will finish our proof by demonstrating that counting from 0 up to n in binary takes $O(n)$ time.

Consider all the binary numbers with a streak of t many 1's in its least significant bits, and call them *t-streak binary numbers*. To increment such a number, a TM needs to propagate the carry for t digits, and spend $t+1$ steps in total. On top of that, to increment a number repeatedly, the head must position itself back onto the least significant bit between increments. Combining these two, the following function gives us the effective cost of incrementing a t-streak binary number, during a repeated incrementation:

$$g(t) = \begin{cases} 1 & \text{if } t = 0 \\ 2t & \text{otherwise} \end{cases} \quad \in \quad O(t)$$

A binary number is t-streak, if and only if it is $(2^t - 1) + y \cdot 2^{t+1}$ for some $y \in \mathbb{N}$. As a result, number of t-streak binary numbers between 0 and n is:

$$f(n,t) = \left\lfloor \frac{n - (2^t - 1)}{2^{t+1}} \right\rfloor + 1 \quad \in \quad O\left(\frac{n}{2^t}\right)$$

The product $g(t) \cdot f(n,t)$ gives the total number of steps needed for all the t-streak binary numbers while counting from 0 to n. Let $T(n)$ denote the total time complexity of counting from 0 to n. We note that the longest streak of 1's in any binary number up to n is $\lfloor \log(n+1) \rfloor$. Then, for some constant c, and large values of n, an asymptotic upper bound to $T(n)$ can be established as follows:

$$T(n) = \sum_{t=0}^{\lfloor \log(n+1) \rfloor} f(n,t) \cdot g(t) \leq c \cdot n \cdot \sum_{t=0}^{\lfloor \log(n+1) \rfloor} \frac{t}{2^t} \tag{4.1}$$

Let $L(n)$ be equal to the sum on the right hand side. Then;

$$\begin{aligned}
L(n) &= \frac{0}{2^0} + \cdots + \frac{\lfloor \log(n+1) \rfloor - 1}{2^{\lfloor \log(n+1) \rfloor - 1}} + \frac{\lfloor \log(n+1) \rfloor}{2^{\lfloor \log(n+1) \rfloor}} \\
2L(n) &= \frac{0}{2^{-1}} + \frac{1}{2^0} + \cdots + \frac{\lfloor \log(n+1) \rfloor}{2^{\lfloor \log(n+1) \rfloor}} \\
2L(n) - L(n) &= 0 + \sum_{t=0}^{\lfloor \log(n+1) \rfloor} \frac{1}{2^t} - \frac{\lfloor \log(n+1) \rfloor}{2^{\lfloor \log(n+1) \rfloor}} \\
&< \sum_{t=0}^{\lfloor \log(n+1) \rfloor} \frac{1}{2^t} \\
&< 2
\end{aligned}$$

Plugging this result back to Inequality (4.1), we get the following:

$$\begin{aligned}
T(n) &< c \cdot n \cdot 2 \\
&\in O(n)
\end{aligned}$$

Thus, 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 $\mathrm{NTISP}(n^2/\log(n), \log n)$.

If $\mathrm{1IP}_*(\infty, \mathsf{cons}, \mathsf{cons})$ is indeed a proper subset of $\mathrm{1IP}(\infty, \mathsf{cons}, \mathsf{cons})$, 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 4.2. *Let V be a 2pfa verifier flipping r coins at most, and recognizing the language A. If V, running on some string $x \notin A$ of length n, and paired with some certificate $c(x)$, always makes $\omega\left(n^{2^{r-1}}\right)$ steps, then $\varepsilon_s(V) \geq 1/2$.*

Proof. Let V be a 2pfa as described above, recognizing A. By an idea introduced in [2], 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 $O\left(n^{2^{r-1}}\right)$ 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 $\omega\left(n^{2^{r-1}}\right)$ 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 strong errors are at least $1/2$. □

5. CONCLUSION

We have shown that the languages recognized by a linear-time $\mathsf{2nfa}(k)$ can also be recognized by constant-randomness $\mathsf{2pfa}$ verifiers with arbitrarily small error. To arrive at that conclusion, we first introduced the notions of safe and risky heads of a $\mathsf{2nfa}(k)$, a characteristic we also proved decidable. We then present an algorithm for a constant-coin $\mathsf{2pfa}$ verifier that recognizes the language of any given $\mathsf{2nfa}(k)$ with a safe head with any desired nonzero error bound. Finally, we show how one can convert a $\mathsf{2nfa}(k)$ running in linear time into a $\mathsf{2nfa}(h)$ having at least one safe head, and vice versa.

It is an open question, whether there is any language requiring a multi-head finite automaton to spend superlinear time to verify, i.e. that if $\mathcal{L}(\mathsf{2nfa}(\mathbb{Z}_+), \mathsf{linear\text{-}time}) \subsetneq \mathcal{L}(\mathsf{2nfa}(\mathbb{Z}_+))$. With that said, we hope to have assisted to any further research on that question by showing $\mathcal{L}(\mathsf{2nfa}(\mathbb{Z}_+), \mathsf{linear\text{-}time}) \subseteq \mathrm{NTISP}(n^2/\log(n), \log n)$ — that any $\mathsf{2nfa}(k)$ running in linear time can be simulated by a nondeterministic TM using logarithmic space and in $O(n^2/\log(n))$ steps, for inputs of length n.

REFERENCES

1. Condon, A. and R. Ladner, "Interactive proof systems with polynomially bounded strategies", *Journal of Computer and System Sciences*, Vol. 50, No. 3, pp. 506–518, 1995.

2. Say, A. C. C. and A. Yakaryılmaz, "Finite state verifiers with constant randomness", *Logical Methods in Computer Science*, Vol. 10, No. 3, pp. 1–17, Aug. 2014.

3. Gezer, M. U., "Windable Heads and Recognizing NL with Constant Randomness", A. Leporati, C. Martín-Vide, D. Shapira and C. Zandron (Editors), *Language and Automata Theory and Applications*, Vol. 12038 of *Lecture Notes in Computer Science*, pp. 184–195, Springer International Publishing, Cham, 2020.

4. Gezer, M. U. and A. Say, "Constant-Space, Constant-Randomness Verifiers with Arbitrarily Small Error", *arXiv preprint arXiv:2006.12330*, 2020.

5. Sipser, M., *Introduction to the Theory of Computation*, Cengage Learning, 2012.

6. Rabin, M. O. and D. Scott, "Finite Automata and Their Decision Problems", *IBM Journal of Research and Development*, Vol. 3, No. 2, pp. 114–125, 1959.

7. Hartmanis, J., "On non-determinancy in simple computing devices", *Acta Informatica*, Vol. 1, No. 4, pp. 336–344, 1972.

8. Monien, B., "Two-way multihead automata over a one-letter alphabet", *RAIRO. Inform. théor.*, Vol. 14, No. 1, pp. 67–82, 1980.

9. Geffert, V. and A. Okhotin, "Transforming two-way alternating finite automata to one-way nondeterministic automata", *International Symposium on Mathematical Foundations of Computer Science*, pp. 291–302, Springer, 2014.

10. Dwork, C. and L. Stockmeyer, “Finite State Verifiers I: The Power of Interaction”, *J. ACM*, Vol. 39, No. 4, p. 800–828, Oct. 1992.

11. Condon, A., “The complexity of the max word problem and the power of one-way interactive proof systems”, *Computational Complexity*, Vol. 3, No. 3, pp. 292–305, 1993.

12. Cobham, A., “Time and memory capacity bounds for machines which recognize squares or palindromes”, *IBM Research Report RC-1621*, 1966.

13. van Melkebeek, D., “Time-Space Lower Bounds for NP-complete Problems”, G. Plun, G. Rozenberg and A. Salomaa (Editors), *Current Trends in Theoretical Computer Science*, pp. 265–291, World Scientific, 2004.

14. Dúriś, P. and Z. Galil, “A time-space tradeoff for language recognition”, *Mathematical systems theory*, Vol. 17, No. 1, pp. 3–12, 1984.