Small Depth Proof Systems

ANDREAS KREBS, University of Tübingen, Germany
NUTAN LIMAYE, Indian Institute of Technology, Bombay, India
MEENA MAHAJAN, The Institute of Mathematical Sciences, Chennai, India
KARTEEK SREENIVASAIAH, Max Planck Institute for Informatics, Saarbrücken, Germany

A proof system for a language L is a function f such that Range(f) is exactly L. In this article, we look at proof systems from a circuit complexity point of view and study proof systems that are computationally very restricted. The restriction we study is proof systems that can be computed by bounded fanin circuits of constant depth (NC^0) or of $O(\log \log n)$ depth but with $O(1)$ alternations (poly $\log AC^0$). Each output bit depends on very few input bits; thus such proof systems correspond to a kind of local error correction on a theorem-proof pair.

We identify exactly how much power we need for proof systems to capture all regular languages. We show that all regular languages have poly $\log AC^0$ proof systems, and from a previous result (Beyersdorff et al. [2011a], where NC^0 proof systems were first introduced), this is tight. Our technique also shows that MAJ has poly $\log AC^0$ proof system.

We explore the question of whether TAUT has NC^0 proof systems. Addressing this question about 2TAUT, and since 2TAUT is closely related to reachability in graphs, we ask the same question about Reachability. We show that if Directed reachability has NC^0 proof systems, then so does 2TAUT. We then show that both Undirected Reachability and Directed UnReachability have NC^0 proof systems, but Directed Reachability is still open.

In the context of how much power is needed for proof systems for languages in NP, we observe that proof systems for a good fraction of languages in NP do not need the full power of AC^0; they have SAC^0 or coSAC^0 proof systems.

Categories and Subject Descriptors: F.1.3 [**Computation by Abstract Devices**]: Complexity Measures and Classes—*Relations among complexity classes*; F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems—*Complexity of proof procedures*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational Logic*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation

General Terms: Theory

Additional Key Words and Phrases: Proof complexity, proof circuits, small depth proofs, circuit complexity

ACM Reference Format:
Andreas Krebs, Nutan Limaye, Meena Mahajan, and Karteek Sreenivasaiah. 2016. Small depth proof systems. ACM Trans. Comput. Theory 9, 1, Article 2 (October 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2956229

This work was done while the fourth author was working in The Institute of Mathematical Sciences, Chennai, India.

Authors' addresses: A. Krebs, University of Tübingen, Germany; email: mail@krebs-net.de; N. Limaye, Indian Institute of Technology, Bombay, India; email: nutan@cse.iitb.ac.in; M. Mahajan, The Institute of Mathematical Sciences, Chennai, India; email: meena@imsc.res.in; K. Sreenivasaiah, Max Planck Institute for Informatics, Saarbrücken, Germany; email: karteek@mpi-inf.mpg.de.

DOI: http://dx.doi.org/10.1145/2956229

1. INTRODUCTION

Let f be any computable function mapping strings to strings. Then f can be thought of as a proof system for the language $L = \text{range}(f)$ in the following sense: To prove that a word x belongs to L, provide a word y that f maps to x. That is, view y as a proof of the statement "$x \in L$," and computing $f(y)$ is then tantamount to verifying the proof. From the perspective of computational complexity, interesting proof systems are those functions that are efficiently computable and have succinct proofs for all words in their range. If we use polynomial-time computable as the notion of efficiency, and polynomial-size as the notion of succinctness, then NP is exactly the class of languages that have efficient proof systems with succinct proofs. For instance, the coNP-complete language TAUT has such proof systems if and only if NP equals coNP [Cook and Reckhow 1979].

Since we do not yet know whether NP equals co-NP, a reasonable question to ask is how much more computational power and/or non-succinctness is needed before we can show that TAUT has a proof system. For instance, allowing the verifier the power of randomized polynomial-time computation on polynomial-sized proofs characterizes the class MA; allowing quantum power characterizes the class QCMA. One could also allow the verifier access to some advice, yielding non-uniform classes; see, for instance, Cook and Krajíček [2007], Pudlák [2009], Hirsch [2010], Hirsch and Itsykson [2010], and Beyersdorff et al. [2011b].

An even more interesting, and equally reasonable, approach is to ask the following: How much do we need to reduce the computational power of the verifier before we can formally establish that TAUT does not have a proof system within those bounds? This approach has seen a rich body of results, starting from the pathbreaking work of Cook and Reckhow [1979]. The common theme in limiting the verifier's power is to limit the nature of proof verification or, equivalently, the syntax of the proof, for example, proof systems based on resolution, Frege systems, and so on. See Beame and Pitassi [2001] and Segerlind [2007] for excellent surveys on the topic.

Instead of restricting the proof syntax, if we only restrict the computational power of the verifier, it is not immediately obvious that we get anywhere. This is because it is already known that NP is characterised by succinct proof systems with extremely weak verifiers, namely AC^0 verifiers. Recall that in AC^0 we cannot even check if a binary string has an odd number of 1s [Furst et al. 1984; Håstad 1986]. But an AC^0 computation can verify that a given assignment satisfies a Boolean formula. Nonetheless, one can look for verifiers even weaker than AC^0; this kind of study was initiated in Beyersdorff et al. [2013] where NC^0 proof systems were investigated. In an NC^0 proof system, each output bit depends on just $O(1)$ bits of the input, so to enumerate L as the range of an NC^0 function f, f must be able to do highly local corrections to the alleged proof while maintaining the global property that the output word belongs to L. Unlike with locally decodable error-correcting codes, the correction here must be deterministic and always correct. This becomes so restrictive that even some very simple languages, that are regular and in AC^0, do not have such proof systems, even allowing non-uniformity. And yet there is an NP-complete language that has a uniform NC^0 proof system (see Cryan and Miltersen [2001]). (This should not really be that surprising, because it is known that in NC^0 we can compute various cryptographic primitives.) So the class of languages with NC^0 proof systems slices vertically across complexity classes. It is still not known whether TAUT has a (possibly non-uniform) NC^0 proof system. Figure 1 shows the relationships between classes of languages with proof systems of the specified kind. (Solid arrows denote proper inclusion, and dotted lines denote incomparability.)

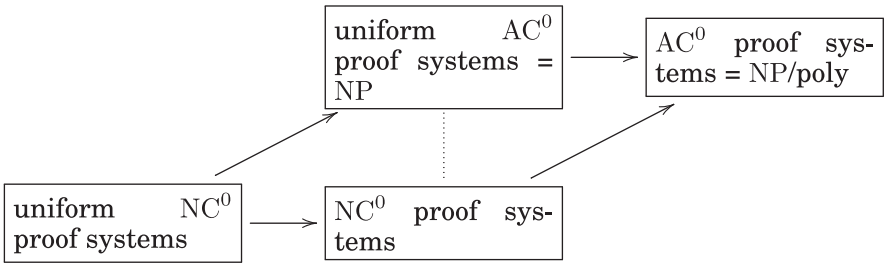

Fig. 1. Some constant-depth proof systems.

The work in Beyersdorff et al. [2013] shows that languages of varying complexity (complete for NC^1, P, NP) have uniform NC^0 proof systems, while the languages EXACT-OR and MAJ, among others, do not have even non-uniform NC^0 proof systems. It then focuses on regular languages and shows that a large subclass of regular languages has uniform NC^0 proof systems. This work takes off from that point.

Our Results

We address the question of exactly how much computational power is required to capture all regular languages via proof systems and answer this question exactly. One of our main results (Theorem 3.6) is that every regular language has a proof system computable by a circuit with bounded fanin gates, depth $O(\log\log n)$, and $O(1)$ alternations. Equivalently, the proof system is computable by an AC^0 circuit where each gate has fanin $(\log n)^{O(1)}$; we refer to the class of such circuits as $\text{poly}\log AC^0$ circuits. By the result of Beyersdorff et al. [2013], EXACT-OR requires depth $\Omega(\log\log n)$, so (up to constant multiplicative factors) this is tight. Our proof technique also generalises to show that MAJ has $\text{poly}\log AC^0$ proof systems (Theorem 3.9).

The most intriguing question here, posed in Beyersdorff et al. [2013], is to characterize the regular languages that have NC^0 proof systems. We state a conjecture for this characterization; the conjecture throws up more questions regarding decidability of some properties of regular languages.

We believe that TAUT does not have AC^0 proof systems because otherwise NP = coNP (see Cook [1971]). As a weaker step, can we at least prove that it does not have NC^0 proof systems? Although it seems that this should be easy to show, we have not yet succeeded. So we ask the same question about 2TAUT, which is in NL and hence may well have an NC^0 proof system. The standard NL algorithm for 2TAUT is via a reduction to STCONN. So it is interesting to ask the following: Does STCONN have an NC^0 proof system? We do not know yet. However, we show in Theorem 4.2 that if STCONN has an NC^0 proof system, then so does 2TAUT.

Intuition suggests that reachability between two vertices in a graph is, in some sense, a global property and hence STCONN should not have NC^0 proof systems. However, in our other main result, we show that the undirected analogue of STCONN, a language complete for L, has an NC^0 proof system (Theorem 4.11). Our construction relies on a careful decomposition of even-degrees-only graphs (established in the proof of Theorem 4.12) that may be of independent interest. We also show that directed unreachability, which is also complete for NL, has an NC^0 proof system (Proposition 4.17).

Finally, we observe that Graph Isomorphism does not have NC^0 proof systems. We also note that for every language L in NP, the language $(\{1\}\cdot L\cdot\{0\})\cup 0^*\cup 1^*$ has both SAC^0 and $coSAC^0$ proof systems (Theorem 5.2).

2. PRELIMINARIES

Unless otherwise stated, we consider only bounded fanin circuits over $\vee, \wedge, \neg$.

Definition 2.1 (*Beyersdorff et al. [2013]*). A circuit family $\{C_n\}_{n>0}$ is a proof system for a language L if there is a function $m : \mathbb{N} \longrightarrow \mathbb{N}$ such that for each n where $L^{=n} \neq \emptyset$,

(1) C_n has $m(n)$ inputs and n outputs,
(2) for each $y \in L^{=n}$, there is an $x \in \{0,1\}^{m(n)}$ such that $C_n(x) = y$ *(completeness)*,
(3) for each $x \in \{0,1\}^{m(n)}$, $C_n(x) \in L^{=n}$ *(soundness)*.

Note that the parameter n for C_n is the number of output bits, not input bits. NC^0 proof systems are proof systems as above where the circuit has $O(1)$ depth. The definition implies that the circuits are of linear size. AC^0 proof systems are proof systems as above where the circuit C_n has $O(\log n)$ depth but $O(1)$ alternations between gate types. Equivalently, they are proof systems as above of $n^{O(1)}$ size with unbounded fanin gates and depth $O(1)$.

Proposition 2.2 (Beyersdorff et al. [2013]). *A regular language L satisfying any of the following has an* NC^0 *proof system:*

(1) *L has a strict star-free expression (built from ϵ, a, and Σ^*, using concatenation and union).*
(2) *L is accepted by an automaton with a universally reachable absorbing final state.*
(3) *L is accepted by a strongly connected automaton.*

Proposition 2.3 (Beyersdorff et al. [2013]).

(1) *Proof systems for* Maj *need $\omega(1)$ depth.*
(2) *Proof systems for* Exact-Count^n_k *and* $\neg$Th$^n_{k+1}$ *need* $\Omega(\log(\log n - \log k))$ *depth. In particular, proof systems for* Exact-Or *and for* Exact-Or $\cup\, 0^*$ *need* $\Omega(\log \log n)$ *depth.*

3. PROOF SYSTEMS FOR REGULAR LANGUAGES

We first explore the extent to which the structure of regular languages can be used to construct NC^0 proof systems. At the base level, we know that all finite languages have NC^0 proof systems. Building regular expressions involves unions, concatenation, and Kleene closure. And the resulting class of regular languages is also closed under many more operations; see, for instance, Hopcroft and Ullman [1979]. We examine these operations one by one.

Theorem 3.1. *Let $\mathcal{C}$ denote the class of languages with* NC^0 *proof systems. Then $\mathcal{C}$ is closed under*

(1) *finite union [Beyersdorff et al. 2013],*
(2) *concatenation with finite sets [Beyersdorff et al. 2013],*
(3) *reversal,*
(4) *fixed-length morphisms,*
(5) *inverses of fixed-length morphisms,*
(6) *fixed-length regular transductions computed by strongly connected (nondeterministic) finite-state automata.*

Proof. Closure under reversal is trivial.

Let h be a fixed-length morphism $h : \{0,1\} \longrightarrow \{0,1\}^k$ for some fixed k. Given a proof system (C_n) for L, a proof system (D_n) for $h(L)$ consists of n parallel applications of h to the each bit of the output of the circuit C_n. Given a proof system D'_n for L, a proof system C'_n for $h^{-1}(L)$ consists of n parallel applications of h^{-1} applied to disjoint k-length

blocks of the output of the circuit D'_{kn}. C'_n needs additional input for each block to choose between possibly multiple pre-images.

If L has an NC^0 proof system (C_n) and h is a regular transduction computed by a strongly connected automaton M, then the construction from Beyersdorff et al. [2013] (Proposition 2.2 (3)) with the output w of C_n as input will produce a word $x \in L(M)$. A small modification allows us output the transduction $h(x)$ instead of x. This works provided there are constants k, ℓ such that each edge in M is labeled by a pair (a, b) with $a \in \{0,1\}^k$ and $b \in \{0,1\}^\ell$. □

THEOREM 3.2. *Let $\mathcal{C}$ denote the class of languages with* NC^0 *proof systems. $\mathcal{C}$ is not closed under*

(1) *complementation [Beyersdorff et al. 2013],*
(2) *concatenation,*
(3) *symmetric difference,*
(4) *cyclic shifts,*
(5) *permutations and shuffles,*
(6) *intersection,*
(7) *quotients.*

PROOF. As noted in Beyersdorff et al. [2013], TH_2^n has an NC^0 proof system but its complement EXACT-OR $\cup\ 0^*$ does not. The languages denoted by the regular expressions 1, 0^*, 10^*, and the languages TH_1, TH_2 all have NC^0 proof systems. The language EXACT-OR does not, but it can be written as $0^* \cdot 10^*$ (concatenation), as $\mathrm{TH}_2 \Delta \mathrm{TH}_1$ (symmetric difference), as the result of cyclic shifts or permutations on 10^*, and as the shuffle of 1 and 0^*.

To see the last two non-closures, it is easier to use non-binary alphabets; the coding back to $\{0,1\}$ is straightforward. Over the alphabet $\{0,1,a,b\}$, the languages $(0^*10^* \cup (0+1+a)^*a(0+1+a)^*)$ and $(0^*10^* \cup (0+1+b)^*b(0+1+b)^*)$ both have NC^0 proof systems (this follows from Proposition 2.2 (2)), but their intersection is EXACT-OR. Also, consider the languages $A = a0^*$, $B_1 = \{xay \mid |x| = |y|, x \in \text{EXACT-OR}, y \in 0^*\}$, $B_2 = \{xay \mid |x| = |y|, x \in (0+1)^*, y \in \mathrm{TH}_1\}$. Then A and $B = B_1 \cup B_2$ have NC^0 proof systems but EXACT-OR $= B \mid A$.

(A proof system for B is as follows: The input proof at length $2n+1$ consists of a word $w \in (0+1)^n$ and the sequence of n states $q_1, \ldots, q_n$ allegedly seen by an automaton M for EXACT-OR on reading w. The circuit copies w into x. If q_{i-1}, w_i, q_i is consistent with M, then it sets y_i to 0, otherwise it sets y_i to 1. It can be verified that the range of this circuit is exactly $B^{=2n+1}$.) □

A natural idea is to somehow use the structure of the syntactic monoid (equivalently, the unique minimal deterministic automaton) to decide whether a regular language has an NC^0 proof system and, if so, to build one. Unfortunately, this idea collapses at once: the languages EXACT-OR and TH_2 have the same syntactic monoid; by Proposition 2.3, EXACT-OR has no NC^0 proof system; and by Proposition 2.2 TH_2 has such a proof system.

The next idea is to use the structure of a well-chosen (non-deterministic) automaton for the language to build a proof system; Proposition 2.2 does exactly this. It describes two possible structures that can be used. However, one is subsumed in the other; see Observation 3.3 below.

OBSERVATION 3.3. *Let L be accepted by an automaton with a universally reachable absorbing final state. Then L is accepted by a strongly connected automaton.*

PROOF. Let M be the non-deterministic automaton with universally reachable and absorbing final state q. That is, q is an accepting state such that (1) q is reachable from every other state of M, and (2) there is a transition from q to q on every letter in Σ. Add moves from q to every state of M on any arbitrary letter (or even ϵ) to get automaton M'. Then M' is strongly connected, and $L(M') = L(M)$. □

A small generalisation beyond strongly connected automata is automata with exactly two strongly connected components. However, the automaton for EXACT-OR is like this, so even with this small extension, we can no longer construct NC^0 proof systems. (In fact, we need as much as $\Omega(\log \log n)$ depth.)

Finite languages do not have strongly connected automata. But they are strict star-free and hence have NC^0 proof systems. Strict star-free expressions lack non-trivial Kleene closure. What can we say about their Kleene closure? It turns out that for any regular language, not just a strict-star-free one, the Kleene closure has an NC^0 proof system.

THEOREM 3.4. *If L is regular, then L^* has an* NC^0 *proof system.*

PROOF. Let M be an automaton accepting L, with no useless states. Adding ϵ moves from every final state to the start state q_0, and adding q_0 to the set of final states gives an automaton M' for L^*. Now M' is strongly connected, so Proposition 2.2 gives the NC^0 proof system. □

Based on the above discussion and known (counter-) examples, we conjecture the following characterization. The structure implies the proof system, but the converse seems hard to prove.

CONJECTURE 3.5. *Let L be a regular language. The following are equivalent:*

(*1*) *L has an* NC^0 *proof system.*
(*2*) *For some finite k, $L = \bigcup_{i=1}^{k} u_i \cdot L_i \cdot v_i$, where each u_i, v_i is a finite word, and each L_i is a regular language accepted by some strongly connected automaton.*

An interesting question arising from this is whether the following languages are decidable:

$$\text{REG-SCC} = \left\{ M \;\middle|\; \begin{array}{l} M \text{ is a finite-state automaton; } L(M) \text{ is accepted by} \\ \text{some strongly connected finite automaton} \end{array} \right\}$$

$$\text{REG-NC}^0\text{-PS} = \left\{ M \;\middle|\; \begin{array}{l} M \text{ is a finite-state automaton; } L(M) \text{ has an NC}^0 \\ \text{proof system} \end{array} \right\}$$

(Instead of a finite-state automaton, the input language could be described in any form that guarantees that it is a regular language.)

It can be shown that REG-SCC is indeed decidable by using the result of Grunsky et al. [2006]. The main result from Grunsky et al. [2006] shows that for every regular language L, there exists a constant c_L such that every NFA with more than c_L states that recognizes L contains at least two mergeable states. Moreover, c_L can be computed from a representation of L. The key observation is that merging two states in a strongly connected NFA results in a strongly connected NFA. So if there is a strongly connected NFA for L, then there is one with at most c_L states. Hence the problem of checking if a regular language L has a strongly connected NFA can be decided by first computing c_L and then checking if any NFA with at most c_L states accepts L.

We do not yet know whether the second question is decidable.

We now establish one of our main results. NC^0 is the restriction of AC^0 where the fanin of each gate is bounded by a constant. By putting a fanin bound that is $\omega(1)$ but $o(n^c)$ for every constant c ("sub-polynomial"), we obtain intermediate classes. In particular, restricting the fanin of each gate to be at most $\text{poly}\log n$ gives the class that we call $\text{poly}\log AC^0$ lying between NC^0 and AC^0. We show that it is large enough to have proof systems for all regular languages. As mentioned earlier, Proposition 2.3 implies that this upper bound is tight.

THEOREM 3.6. *Every regular language has a* $\text{poly}\log AC^0$ *proof system.*

PROOF. Let $A = (Q, \Sigma, \delta, q_0, F)$ be an automaton for L. We assume that $\Sigma = \{0, 1\}$, and larger finite alphabets can be suitably coded. We unroll the computation of A on inputs of length n to get a layered branching program B with $n + 1$ layers numbered 0 to n. The initial layer of B has just the start node s, which behaves like q_0 in the automaton, while every other layer of the branching program has as many vertices as $|Q|$. Since A may have multiple accepting states, we add an extra layer at the end with a single sink node t and connect all copies of accepting states at layer n to t by edges labeled 1. Note that B has the following properties:

—Length $l = n + 2$.
—Every layer except the first and last layer has width (number of vertices in that layer) $w = |Q|$.
—Edges are only between consecutive layers. These edges and their labelling are according to δ.
—All edges from layer $i - 1$ to layer i are labelled either x_i or $\overline{x}_i$.
—A word $a = a_1 \dots a_n$ is accepted by A if and only if B has a path from s to t (with $n+1$ edges) with all edge labels consistent with a.

Any vertex $u \in B$ can be indexed by a two tuple (ℓ, p) where ℓ stands for the layer where u appears and p is the position where u appears within layer ℓ.

Represent the interval $(0, n + 1]$ as a binary tree T where

(1) the root corresponds to the interval $(0, n + 1] = \{1, 2, \dots, n + 1\}$,
(2) a node corresponding to interval $(i, j]$ has children corresponding to intervals $(i, \lceil \frac{i+j}{2} \rceil]$ (left child) and $(\lceil \frac{i+j}{2} \rceil, j]$ (right child), and
(3) a node corresponding to interval $(k - 1, k]$ for $k \in [n + 1]$ is a leaf.

We call this the interval tree. For each interval $(i, j]$ in T, we provide a pair of states $\langle u, v \rangle$; these are intended to be the states q_i and q_j in the alleged accepting run ρ. (Note that the state sequence on ρ itself is now supposed to be specified at the leaves of T.)

Consider the interval tree T for $(0, n + 1]$ described above. The input to the proof system consists of a string $a \in \{0, 1\}^n$ and a pair of labels $\langle u, v \rangle$ for each node in the interval tree. The labels u, v point to nodes of B. For interval $(i, j]$, the labels are of the form $u = (i, p)$, $v = (j, q)$. Since i, j are determined by the node in T, the input only specifies the pair $\langle p, q \rangle$ rather than $\langle u, v \rangle$. That is, it specifies a pair of states from A. At the root node, the labeling is hardwired to be $\langle s, t \rangle$.

Given a word $a = a_1 \dots a_n$ and a labeling as above of the interval tree, we define feasibility and consistency as follows:

(1) A leaf node $(k - 1, k]$ with $k \in [n]$, labeled $\langle p, q \rangle$, is
   (a) **feasible** if there exists an edge from $(k - 1, p)$ to (k, q) in B. (That is, there exists $b \in \Sigma$ such that $q \in \delta(p, b)$.)
   (b) **consistent** if there exists an edge from $(k - 1, p)$ to (k, q) in B labeled x_k if $a_k = 1$, labeled $\overline{x_k}$ if $a_k = 0$. (That is, $q \in \delta(p, a_k)$.)

(The case $k = n + 1$ is simpler: feasible and consistent if p is a final state of A.)

(2) An internal node $(i, j]$ labeled $\langle p, q\rangle$ is
   (a) **feasible** if there exists a path from (i, p) to (j, q) in B. (That is, there exists a word $b \in \Sigma^{j-i}$ such that $q \in \tilde{\delta}(p, b)$.)
   (b) **consistent** if it is feasible, both its children are feasible, and the labels $\langle p', q'\rangle$ and $\langle p'', q''\rangle$ of its left and right children respectively satisfy: $p = p'$, $q = q''$, $q' = p''$.

(3) A node is **fully consistent** if all its ancestors (including itself) are consistent.

Since the label at the root of T is hardwired, the root node is always feasible. But it may not be consistent.

For each node $(i, j]$ in the interval tree, and each potential labeling $\langle p, q\rangle$ for this node, let $u = (i, p)$ and $v = (j, q)$. Define the predicate $R(u, v)$ to be 1 if and only if there is a path from u to v in B (i.e., this potential labeling is feasible.) Whenever $R(u, v) = 1$, fix a partial assignment $w_{u,v}$ that assigns 1 to all literals that occur as labels along an arbitrarily chosen path from u to v. Note that $w_{u,v}$ assigns exactly $j - i$ bits, to the variables $x_{i+1}, \ldots, x_j$. We call $w_{u,v}$ the **feasibility witness** for the pair (u, v).

Let y be the output string of the proof system we construct. A bit y_k of the output y is computed as follows: Find the lowest ancestor of the node $(k - 1, k]$ that is fully consistent.

—If the leaf node $(k - 1, k]$ is fully consistent, then output a_k.
—If there is no such node, then the root node is inconsistent. Since it is feasible, the word $w_{s,t}$ is defined. Output the kth bit of $w_{s,t}$.
—If such a node is found, and it is not the leaf node itself but some $(i, j]$ labeled $\langle p, q\rangle$, then let $u = (i, p)$ and $v = (j, q)$. The word $w_{u,v}$ is defined and assigns a value to x_k. Output this value.

It follows from this construction that every word $a \in L$ can be produced as output: Give in the proof the word a and label the interval tree fully consistent with an $s - t$ path of B consistent with a (equivalently, an accepting run of A on a).

It also follows that every word y output by this construction belongs to L. On any proof, moving down from the root of the interval tree, locate the frontier of lowest fully consistent nodes. These nodes are feasible and correspond to a partition of the input positions, and the procedure described above outputs a word constructed by patching together the feasibility-witnesses for each part.

To see that the above construction can be implemented in depth $O(\log \log n)$ with $O(1)$ alternations, observe that each of the conditions, feasibility, consistency, and equality, of two labels depend on $O(\log w)$ bits. Hence depth of $O(\log \log n)$ and $O(1)$ alternations suffices for their implementation.

More formally, define the following set of predicates:

—$\textsc{Equal} : [w]^2 \longrightarrow \{0, 1\}$ the Equality predicate on $\log w$ bits.
—For each $0 \le i < j \le n + 1$, $\textsc{Feasible}_{i,j} : [w]^2 \longrightarrow \{0, 1\}$ is the Feasibility predicate with arguments the labels (p, q) at interval $(i, j]$.
—For each $0 \le i < j + 1 \le n + 1$, $\textsc{Consistent}_{i,j} : [w]^6 \longrightarrow \{0, 1\}$ is the Consistency predicate at an internal node, with arguments the labels at interval $(i, j]$ and at its children.
—For each $0 < k \le n + 1$, $\textsc{ConsistentLeaf}_k : [w]^2 \times \Sigma \longrightarrow \{0, 1\}$ is the Consistency predicate at leaf $(k - 1, k]$ with arguments the label $\langle p, q\rangle$ and the bit a_k at the leaf.

All the predicates depend on $O(\log w)$ bits. So a naive truth-table implementation suffices to compute them in depth $O(\log w)$ with $O(1)$ alternations.

For any $0 < k \leq n+1$, let the nodes on the path from $(k-1,k]$ to the root of the interval tree be the intervals $(k-1,k] = (i_0,j_0),(i_1,j_1],\ldots,(i_r,j_r] = (0,n+1]$. Note: $r \in O(\log n)$.

Given a labeling of the tree, the output at position k is given by the expression below (it looks ugly, but it is just implementing the scheme described above; we write it in this detail to make the poly log AC^0 computation explicit),

$$\begin{aligned} y_k \;=\; & \left[a_k \wedge \textsc{ConsistentLeaf}_k \wedge \bigwedge_{h=1}^{r} \textsc{Consistent}_{i_h,j_h} \right] \\ \vee\; & \left[(w_{s,t})_k \wedge \overline{\textsc{Consistent}_{0,n+1}} \right] \\ \vee\; & \left[\bigvee_{h=1}^{r} \left(w_{(i_h,p_h),(j_h,q_h)}\right)_k \wedge \overline{\textsc{Consistent}_{i_{h-1},j_{h-1}}} \wedge \bigwedge_{g=h}^{r} \textsc{Consistent}_{i_g,j_g} \right], \end{aligned}$$

where the arguments to the predicates are taken from the tree labeling. This computation adds $O(1)$ alternations and $O(\log\log n)$ depth to the computation of the predicates, so it is in poly log AC^0. □

While proving Theorem 3.6, we unrolled the computation of a w-state automaton on inputs of length n into a layered branching program BP of width w with $\ell = n+2$ layers. The BP so obtained is nondeterministic whenever the automaton is. The BP has a very restricted structure that we exploited to construct the poly log AC^0 proof system.

We observe that some restrictions on the BP structure can be relaxed and still we can construct a poly log AC^0 proof system.

Definition 3.7. A branching program for length-n inputs is **structured** if it satisfies the following:

(1) It is layered: vertices are partitioned into $n+1$ layers $V_0,\ldots,V_n$ and all edges are between adjacent layers $E \subseteq \cup_i (V_{i-1} \times V_i)$.
(2) Each layer has the same size $w = |V_i|$, the width of the BP. (This is not critical; we can let $w = \max |V_i|$.)
(3) There is a permutation $\sigma \in S_n$ such that for $i \in [n]$, all edges in $V_{i-1} \times V_i$ read $x_{\sigma(i)}$ or $\overline{x_{\sigma(i)}}$.

Non-uniform automata [Barrington 1989; Barrington and Thérien 1988] give rise to branching programs that are structured with w the number of states in the automaton. For instance, the language $\{xx \mid x \in \{0,1\}^*\}$ is not regular. But if the input bits are provided in the order $1, m+1, 2, m+2, \ldots, m, 2m$ then it can be decided by a finite-state automaton. This gives rise to a structured BP where σ is the inverse of the above order (e.g., $r_2 = m+1$, $r_3 = 2$, $\sigma(m+1) = 2$, $\sigma(2) = 3$).

The idea behind the construction in Theorem 3.6 works for such structured BPs. It yields a proof system with depth $O(\log\log n + \log w)$. This means that for $w \in O(\text{poly}\log n)$, we still get poly log AC^0 proof systems. Potentially, this is much bigger than the class of languages accepted by non-uniform finite-state automata. Formally,

Theorem 3.8. *Languages accepted by structured branching programs of width* $w \in (\log n)^{O(1)}$ *have* poly log AC^0 *proof systems.*

For the language MAJ of strings with more 1s than 0s, and in general for threshold languages TH_k^n of strings with at least k 1s, we know that there are constant-width branching programs, but these are not structured in the sense above. It can be shown that a structured BP for MAJ must have width $\Omega(n)$ (a family of growing automata M_n for MAJ, where M_n is guaranteed to be correct only on $\{0,1\}^n$, must have $1+n/2$ states in M_n). This is much much more than the poly log width bound used in the construction in Theorem 3.6. Nevertheless, we show below how we can modify that construction to get a poly log AC^0 proof system, even for threshold languages.

THEOREM 3.9. *For every n and $t \leq n$, the language TH_t^n has a* poly log AC^0 *proof system.*

PROOF. We follow the approach in Theorem 3.6: The input to the proof system is a word $a = a_1, \ldots, a_n$ and auxiliary information in the interval tree allowing us to correct the word if necessary. The labeling of the tree differs for this language and is as follows. Each interval $(i, j]$ in the tree gets a label that is an integer in the range $\{0, 1, \ldots, j-i\}$. The intention is that for an input $a = a_1, \ldots, a_n$, this label is the number of 1s in the subword $a_{i+1} \ldots a_j$. For thresholds, we relax the constraint: We expect the label of interval $(i, j]$ to be *no more than* the number of 1s in the subword. At a leaf node $(k-1, k]$, we do not give explicit labels; a_k serves as the label. At the root also, we do not give an explicit label; the label t is hard wired. (We restrict the label of any interval $(i, j]$ to the range $[0, j-i]$, and interpret larger numbers as $j-i$.)

For any node u of T, let $l(u)$ denote the label of u. A node u with children v, w is *consistent* if $l(u) \leq l(v) + l(w)$.

Let the output of our proof system be $y_1, \ldots, y_n$. The construction is as follows:

—If all nodes on the path from $(k-1, k]$ to the root in T are consistent, then $y_k = a_k$.
—Otherwise, $y_k = 1$.

In analogy with Theorem 3.6, we use here for each interval $(i, j]$ the feasibility witness 1^{j-i}, independent of the actual labels. Thus the construction forces this property: At a node u corresponding to interval $(i, j]$ labelled $\ell(u)$, the subword $y_{i+1}, \ldots, y_j$ has at least $\min\{\ell(u), j-i\}$ 1s. Thus, the output word is always in TH_t^n. Every word in TH_t^n is produced by the system at least once, on the proof that gives, for each interval other than $(0, n]$, the number of 1s in the corresponding subword.

As before, the $\text{CONSISTENT}_{i,j}$ predicate at a node depends on three labels, each of which is $O(\log n)$ bits long. A truth-table implementation is not good enough; it will give an AC^0 circuit. But the actual consistency check only involves adding and comparing $m = \log n$ bit numbers. Since addition and comparison are in AC^0, this can be done in depth $O(\log m)$ with $O(1)$ alternations. Thus the overall depth is $O(\log \log n)$. □

COROLLARY 3.10. *For every n and $t \leq n$,* EXACT-COUNT_t^n *has a* poly log AC^0 *proof system.*

PROOF. We follow the same approach as Theorem 3.9. We redefine *consistent* as follows: For any node u of T, let $l(u)$ denote the label of u. A node u with children v, w is consistent if $l(u) = l(v) + l(w)$. Let the output of our proof system be $y_1, \ldots, y_n$. The construction is as follows:

—If all nodes on the path from $(k-1, k]$ to the root in T are consistent, then $y_k = a_k$.
—Otherwise, let $u = (p, q]$ be the topmost node along the path from $(k-1, k]$ to the root that is not consistent. We output $y_k = 1$ if $k - p \leq l(u)$, 0 otherwise.

That is, for $u = (i, j]$ labeled $\ell(u)$, if $L = \min\{\ell(u), j-i\}$, use feasibility witness $1^L 0^{j-i-L}$. □

4. 2TAUT AND NC^0 PROOF SYSTEMS

In this section, we try to understand NC^0 proof systems better in the context of 2TAUT—the language of all 2DNFs that are tautologies. The main goal is to understand the relationship between TAUT and NC^0 proof systems. However, the language 2TAUT has more structure because of the connection with implication graphs. Hence, we want to examine 2TAUT and the associated language of graph reachability.

First, we show a general construction for monotone properties. Recall that a function f is monotone if whenever $f(x) = 1$ and y dominates x (that is, $\forall i \in [n], x_i = 1 \Rightarrow y_i = 1$), then it also holds that $f(y) = 1$. For such a function, a string x is a *minterm* if $f(x) = 1$ but x does not dominate any z with $f(z) = 1$. Minterms(f) denotes the set of all minterms of f. Clearly, Minterms(f) $\subseteq f^{-1}(1)$. The following lemma states that for any monotone function f, constructing a proof system for a language that sits in between Minterms(f) and $f^{-1}(1)$ suffices to get a proof system for $f^{-1}(1)$.

LEMMA 4.1. *Let $f : \{0,1\}^* \longrightarrow \{0,1\}$ be a monotone Boolean function and let $L = f^{-1}(1)$. Let $L_n = L \cap \{0,1\}^n$. Let L' be a language such that for each length n, $(Minterms(L) \cap \{0,1\}^n) \subseteq (L' \cap \{0,1\}^n) \subseteq L_n$. If L' has a proof system of depth d, size s, and a alternations, then L has a proof system of depth $d+1$, size $s+n$, and at most $a+1$ alternations.*

PROOF. Let C be a proof circuit for L' that takes input string x. We construct a proof system for L using C and asking another input string $y \in \{0,1\}^n$. The ith output bit of our proof system is $C(x)_i \vee y_i$. □

In the following section, we will show that 2TAUT and directed graph reachability share a similar relationship with each other in terms of proof systems as their decision versions do in computational complexity.

4.1. Directed Reachability and 2TAUT

Let UNSAT denote the language of all unsatisfiable formulas. Recall that deciding if a 2CNF formula F belongs to UNSAT can be done in polynomial time. This is because one can build the following "implication graph" G from F: G is a directed graph with twice as many vertices as the number of variables in F—one vertex for each possible literal. Each clause $(\ell_i \vee \ell_j)$ of F can be written as two implications: $\overline{\ell_i} \Rightarrow \ell_j$ and $\overline{\ell_j} \Rightarrow \ell_i$. An edge (ℓ_i, ℓ_j) is present in G if the implication $\ell_i \Rightarrow \ell_j$ is in the formula F, that is, for every term $(\ell_i \vee \ell_j)$ in the formula F, G will have two edges, namely $(\overline{\ell_i}, \ell_j)$ and $(\overline{\ell_j}, \ell_i)$. Let vertices u_i be associated with literals x_i and vertices v_i be associated with the literal $\overline{x_i}$. Now F is unsatisfiable if and only if there exists an i such that there is a path from u_i to v_i and a path from v_i to u_i. Checking the existence of such paths can be done in non-deterministic logspace and hence in polynomial time; thus deciding if F is unsatisfiable is in NL and hence in P. Let 2UNSAT denote all 2CNF formulas that are unsatisfiable. Note that since 2TAUT $= \{F | \overline{F} \in$ 2UNSAT$\}$, deciding if a 2DNF formula F is in 2TAUT is also in non-deterministic logspace and hence in P.

The encoding used by Cryan and Miltersen [2001] in their construction of an NC^0 proof system for the NP-Complete problem E3SAT has exactly $2^3\binom{n}{3}$ bits—one bit for each possible clause. Using a similar idea, we can encode 2CNF formulas on n variables by a bit string of $2n(2n-1)$ bits—one bit for each possible clause. (Each clause has two literals. For the first literal, we have $2n$ possible choices and for the second we have $2n-1$ possible choices.) Note that here we are allowing for trivial clauses like $(x_i \vee \overline{x}_i)$. We will also assume that if the bit corresponding to a clause $l_i \vee l_j$ is 1, then the bit corresponding to the clause $l_j \vee l_i$ is also 1 and vice versa. This encoding would correspond to the adjacency matrix of the implication graph on $2n$ vertices

described previously but without the diagonal entries. However, for convenience, we will generate adjacency matrices of the implication graphs along with the diagonal entries included. In other words, we allow for self-loops in the graph. This is equivalent to allowing clauses such as $(x_i \vee x_i)$ in the formula. Hence we work with this encoding of $4n^2$ bits for the remainder of this section. It is easy to go from the $4n^2$ bit encoding to the $2n(2n-1)$ bit encoding: merely hide the diagonal entries from being output.

Let STCONN be the language of all n vertex graphs with a path from vertex 1 to vertex n. In the following, we will show a reduction between proof systems generating 2TAUT and STCONN. More precisely, we show that if STCONN has an NC^0 proof system, then so does 2TAUT.

Define the following languages:

$$\mathrm{STCONN}_n = \left\{ A \in \{0,1\}^{n \times n} \;\middle|\; \begin{array}{l} A \text{ is the adjacency matrix of a directed graph } G \text{ with} \\ \text{a path from vertex } s = 0 \text{ to vertex } t = n-1. \end{array} \right\}$$

$$\mathrm{STCONN} = \bigcup_{n>0} \mathrm{STCONN}_n.$$

When handling graphs, throughout this section, we write $u \rightsquigarrow v$ to denote "$\exists$ a path from u to v" where u and v are vertices of the graph being used.

We will show that a proof system for the set STCONN can be used to construct a proof system for 2UNSAT with only a constant blowup to the depth of the proof system.[1]

THEOREM 4.2. *If* STCONN *has an* NC^0 *proof system, then* 2UNSAT *has an* NC^0 *proof system.*

PROOF. Let $\mathcal{Q}$ be a proof system computable in NC^0 for STCONN.

We first show that, using output of circuits from $\mathcal{Q}$, we can generate the language GOODCONN defined as follows:

$$\mathrm{GOODCONN} = \bigcup_i \mathrm{GOOD}_i,$$

where GOOD_i is defined as:

$$\mathrm{GOOD}_i = \{G \subseteq \mathrm{STCONN}_{2n+2} | \exists i \in [n], \exists \text{ simple path } 0 \to i \rightsquigarrow \bar{i} \to 2n+1\},$$

where $\bar{i}$ is short for $n+i$.

LEMMA 4.3. *If* STCONN *has an* NC^0 *proof system, then so does* GOODCONN.

PROOF. We construct proof system $\mathcal{GC}$ computable in NC^0 for GOODCONN by using proof system $\mathcal{Q}$ of STCONN. Let $Q \in \mathcal{Q}$ be the proof circuit that outputs adjacency matrices of graphs in STCONN_{2n+2}. We number the vertices of the graph output by Q as $0, 1, 2, \ldots, (2n+1)$. Let $s = 0$ and $t = 2n+1$ as shown in Figure 2. Let the adjacency matrix output by Q be H.

Construction: We will construct proof circuit $P \in \mathcal{GC}$ that outputs every graph in GOODCONN on $2n+2$ vertices. P takes the following as input: H and another adjacency matrix B. P outputs every edge in H, every edge in B, and also some additional edges determined by the following rules:

[1]The authors thank Vladimir Podolskii for suggesting this approach and for useful discussions on the proof of this theorem.

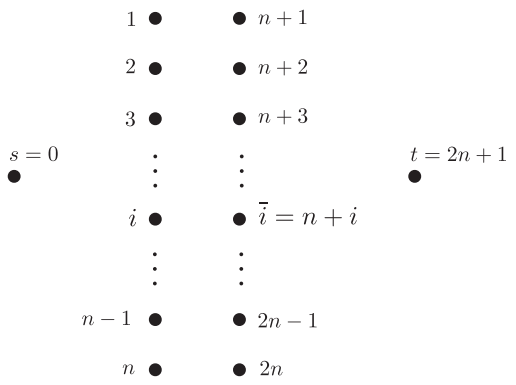

Fig. 2. Vertex numbering.

(1) If $H[s,t] = 1$, then include path $s \to 1 \to n+1 \to t$.
(2) For each $i \in [n]$, if $H[s,\bar{i}] = 1$, then include path $s \to i \to \bar{i} \to t$.
(3) For each $i \in [n]$, if $H[i,t] = 1$, then include path $s \to i \to \bar{i} \to t$.
(4) For each $i \in [n]$, if $H[s,i] = 1$, then add edge $(\bar{i} \to t)$.
(5) For each $i, j \in [n]$ such that $i \neq j$, if $H[s,i] = H[\bar{j},t] = 1$, then add edge $(\bar{j},\bar{i})$.

Intuitively, the vertices from 1 to n represent the literals $x_1, \ldots, x_n$ and the vertices from $n+1$ to $2n$ represent the literals $\overline{x}_1, \ldots, \overline{x}_n$. The idea behind the construction is to force a simple path $s \to i \rightsquigarrow \bar{i} \to t$ for some $i \in [n]$. However, if the graph H output by P was such that it had a simple path $s \to i \rightsquigarrow \bar{i} \to t$ and no other edges, then none of the rules apply. The extra input B is simply for upward closure (like in Lemma 4.1) to guarantee completeness.

Soundness: Let G be a graph output by P. We need to show that G is in GoodConn. It suffices to show the following:

Claim 4.4. *$\exists i \in [n]$ such that $G \in \text{Good}_i$*

Proof. Let $G = P(H,B)$. Since G has all edges from H and $H \in \text{STConn}_{2n+2}$, there is a simple path ρ from s to t in H (and in G). Consider the last edge (u,t) in ρ. We have the following cases:

—Case $u = s$: Rule 1 implies $G \in \text{Good}_1$.
—Case $u = i$ for some $i \in [n]$: Rule 3 implies $G \in \text{Good}_i$.
—Case $u = \bar{i}$ for some $i \in [n]$: Consider the first edge (s,v) in ρ. We have three cases:
  —Case $v = \bar{j}$ for some $j \in [n]$: Rule 2 implies $G \in GOOD_j$.
  —Case $v = i$: Straightforward to see $G \in GOOD_i$.
  —Case $v = j$ for some $j \in [n]$ and $j \neq i$: Rule 4 and Rule 5 together imply $G \in GOOD_j$. An example of this case is shown in Figure 3. □

Completeness: We need to show that any graph $G \in$ GoodConn can be produced by P. We observe the following:

Observation 4.5. *Let $J \in \text{STConn}_{2n+2}$ be a graph that has a simple path $s \to i \rightsquigarrow \bar{i} \to t$ for some $i \in [n]$ and no other edges. If B is all 0s, then $P(J,B)$ is exactly J.*

From definition of GoodConn, $G \in$ GoodConn implies $G \in \text{Good}_i$ for some $i \in [n]$. Hence there exists a simple path ρ that proceeds as $s \to i \rightsquigarrow \bar{i} \to t$ in G. Consider

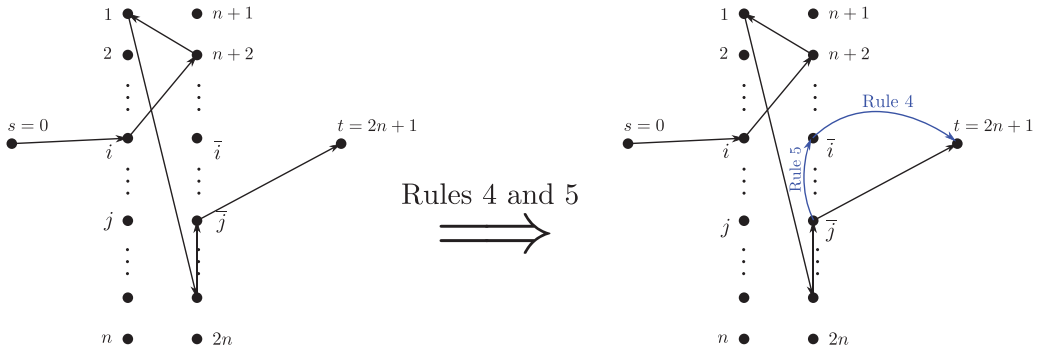

Fig. 3. Example of Rules 4 and 5.

a graph J with path ρ and no other edges. Then J is an output of Q. Let B be the $(2n+2) \times (2n+2)$ adjacency matrix of G. Then $P(J, B) = G$.

$\mathcal{GC}$ is Computable in NC^0*:*

LEMMA 4.6. *If $\mathcal{Q}$ is computable in* NC^0*, then $\mathcal{GC}$ is computable in* NC^0.

PROOF. Assume $\mathcal{Q}$ is computable in NC^0. It suffices to show that each output in a proof circuit $P \in \mathcal{GC}$ is a function of only $O(1)$ of its input bits. P uses the output H of a proof circuit $Q \in \mathcal{Q}$. Let A be the output adjacency matrix of P. To see that the rules determining the edges can be implemented in NC^0, we explicitly write down the circuit for each type of edge:

—(Rules 2 and 3)
  —For all $i \in [n]\backslash\{1\}$, $A[s, i] = H[s, \bar{i}] \vee H[i, t] \vee H[s, i] \vee B[s, i]$.
  —For all $i \in [n]\backslash\{1\}$, $A[i, \bar{i}] = H[s, \bar{i}] \vee H[i, t] \vee H[i, \bar{i}] \vee B[i, \bar{i}]$
—(Rules 1,2,3)
  —$A[s, 1] = H[s, t] \vee H[s, n+1] \vee H[1, t] \vee H[s, 1] \vee B[s, 1]$.
  —$A[1, n+1] = H[s, t] \vee H[s, n+1] \vee H[1, t] \vee H[1, n+1] \vee B[1, n+1]$.
  —$A[n+1, t] = H[s, t] \vee H[s, n+1] \vee H[1, t] \vee H[n+1, t] \vee B[n+1, t]$.
—(Rules 2,3 and 4) For all $i \in [n]$, $A[\bar{i}, t] = H[s, i] \vee H[s, \bar{i}] \vee H[i, t] \vee H[\bar{i}, t] \vee B[\bar{i}, t]$.
—(Rule 5) For all $i, j \in [n]$, $i \neq j$, $A[\bar{j}, \bar{i}] = (H[s, i] \wedge H[\bar{j}, t]) \vee H[\bar{j}, \bar{i}] \vee B[\bar{j}, \bar{i}]$.

For all edges $e = (u, v)$ that do not fall under any of the above types, $A[e] = H[e] \vee B[e]$.

Thus each output bit is a function of at most 4 bits from H and 1 bit from B. H is the output of the proof circuit Q. Hence if proof system $\mathcal{Q}$ is an NC^0 circuit family, then so is $\mathcal{P}$. □

This completes the proof of Lemma 4.3. □

We define the following languages that have paths in the reverse direction as GOODCONN:

$$\textsc{Good}_i^R = \{G \subseteq \textsc{STConn}_{2n+2} | \exists i \in [n], \exists \text{ simple path } 2n+1 \to \bar{i} \rightsquigarrow i \to 0\}$$

$$\textsc{GoodConn}^R = \bigcup_i \textsc{Good}_i^R,$$

where the R in the superscript means "reversed."

Note that $\textsc{GoodConn}^R$ can be generated using proof circuits from $\mathcal{GC}$ and either numbering the vertices of the output graph in reverse or by reversing the direction of

each edge in the output graph. Let $\mathcal{GC}^R$ be such a proof system. The following is an easy observation:

OBSERVATION 4.7. *$\mathcal{GC}^R$ is computable in* NC^0 *if and only if $\mathcal{GC}$ is computable in* NC^0.

We now show how to use the proof system for GOODCONN to obtain a proof system for 2TAUT.

LEMMA 4.8. *If* GOODCONN *has an* NC^0 *proof system, then so does* 2TAUT.

PROOF. We will construct proof system $\mathcal{P}$ that generates all 2CNF formulas that are unsatisfiable and hence for 2TAUT. The idea is to take two graphs, G_1 with a path from some $i \in [n]$ to $\bar{i}$ and G_2 with a path from $\bar{j}$ to j for some $j \in [n]$, and combine them together to get a graph with path $i \rightsquigarrow \bar{i} \rightsquigarrow i$. When the vertices from 1 to n are interpreted as positive literals and vertices from $n+1$ to $2n+1$ are interpreted as negated literals, we get an implication graph of a formula that is not satisfiable. We will describe the construction of a proof circuit $P \in \mathcal{P}$ that generates every such formula on n variables. As mentioned before, the encoding we use has $4n^2$ bits and represents the adjacency matrix of the implication graph. Equivalently, each bit represents a clause in the formula.

Construction. P takes the following as input:

—Adjacency matrix A_1 of a graph $G_1 \in$ GOODCONN on $2n+2$ vertices.
—Adjacency matrix A_2 of a graph $G_2 \in$ GOODCONN^R on $2n+2$ vertices.
—$(2n+2) \times (2n+2)$ Adjacency matrix B.

The guarantee that the first two input adjacency matrices come from GOODCONN and GOODCONN^R, respectively, is achieved by using the outputs of the appropriate proof circuits $P_1 \in \mathcal{GC}$ and $P_2 \in \mathcal{GC}^R$ as constructed before.

For convenience, we will make P compute a graph on $2n+2$ vertices. For the final output graph, we do not output the vertices s and t or any edges that involve s or t.

P outputs a graph that has all edges in G_1, all edges in G_2, all edges indicated by B, and some additional edges determined by the following "Stitch" rule:

—Stitch rule: If $(\bar{i}, t) \in G_1$ and $(t, \bar{j}) \in G_2$, then add edges $(\bar{i}, \bar{j})$ and (j, i).

Soundness. We need to show that any graph G output by P is an implication graph for a formula in 2UNSAT. It suffices to show that there exists an i such that there is a path $i \rightsquigarrow \bar{i}$ and $\bar{i} \rightsquigarrow i$. Putting together these two paths results in a walk starting at i and ending at i via $\bar{i}$. For convenience, we will refer to this as a path (although strictly a walk) and write $i \rightsquigarrow \bar{i} \rightsquigarrow i$.

CLAIM 4.9. *For any graph G output by a circuit $P \in \mathcal{P}$, $\exists i \in [n]$ such that there is a path $i \rightsquigarrow \bar{i} \rightsquigarrow i$.*

PROOF. Let $G = P(G_1, G_2, B)$. Since $G_1 \in$ GOODCONN, there exists an $i \in [n]$ such that a path $\rho_1 \colon s \to i \rightsquigarrow \bar{i} \to t$ is in G_1. Similarly, for some $j \in [n]$, a path $\rho_2 \colon t \to \bar{j} \rightsquigarrow j \to s$ exists in G_2. We now have two cases:

(1) If $i = j$, then since G contains all edges in G_1 and G_2, G has a path $\rho \colon i \rightsquigarrow \bar{i} \rightsquigarrow i$.
(2) If $i \neq j$, then the Stitch rule forces the edges $(\bar{i}, \bar{j})$ and (j, i). Together with ρ_1 and ρ_2, we get a path $\rho \colon i \overset{\rho_1}{\rightsquigarrow} \bar{i} \to \bar{j} \overset{\rho_2}{\rightsquigarrow} j \to i$. An example of this case is shown in Figure 4. □

Completeness. Take any formula $F \in$ 2UNSAT on n variables. Let the implication graph of F be G. We will show that G is produced by P. We know that in G, $\exists i \in [n]$

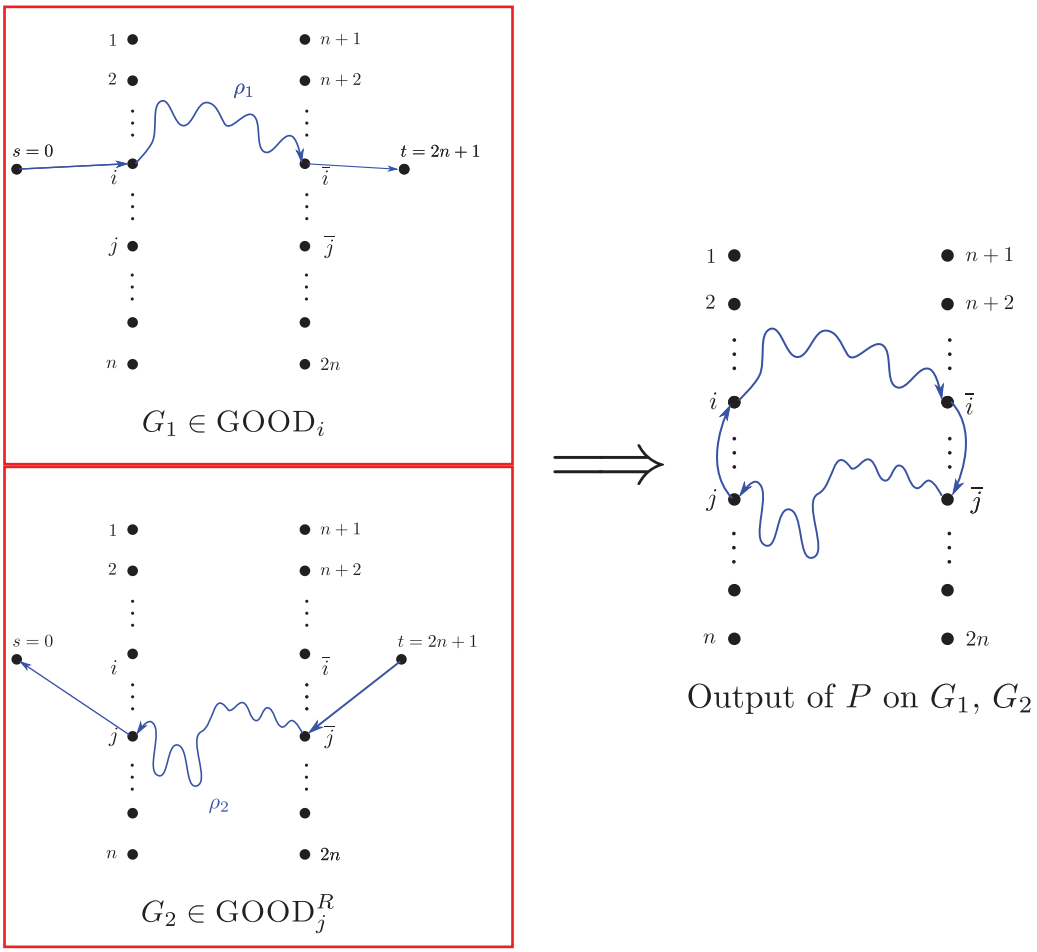

Fig. 4. Effect of Stitch rule.

such that there are simple paths $\rho_1 : i \rightsquigarrow \bar{i}$ and $\rho_2 : \bar{i} \rightsquigarrow i$. Define graph G_1 on $2n$ vertices to contain the path ρ_1 and no other edges. Similarly, let G_2 be a graph on $2n$ vertices with the path ρ_2 and no other edges. Construct graph G'_1 on $2n+2$ vertices as follows: G'_1 is exactly G_1 with two additional vertices $s = 0$ and $t = 2n + 2$ and edges (s, i) and $(\bar{i}, t)$. Clearly, $G'_1 \in$ GoodConn. Similarly, construct G'_2 from G_2 such that $G'_2 \in$ GoodConn^R. Adjacency matrix A_1 of G'_1 is produced as output by proof system P_1 on some input and adjacency matrix A_2 of G'_2 is produced as output by proof system P_2 on some input. Let B be the adjacency matrix of G. It is easy to see that $P(A_1, A_2, B) = G$.

$\mathcal{P}$ is Computable in NC^0.

Claim 4.10. *If $\mathcal{GC}$ is computable in* NC^0*, then $\mathcal{P}$ is computable in* NC^0.

Proof. We need to show that every output bit of a proof circuit $P \in \mathcal{P}$ is a function of at most $O(1)$ many input bits. P uses outputs A_1 and A_2 of proof circuits $P_1 \in \mathcal{GC}$ and $P_2 \in \mathcal{GC}^R$. Let output adjacency matrix of P be A. P incorporates only one rule and this can be expressed formally as follows:

—For all $i, j \in [n]$, $A[\bar{i}, \bar{j}] = (A_1[\bar{i}, t] \wedge A_2[t, \bar{j}]) \vee A_1[\bar{i}, \bar{j}] \vee A_2[\bar{i}, \bar{j}] \vee B[\bar{i}, \bar{j}]$
—For all $i, j \in [n]$, $A[j, i] = (A_1[\bar{i}, t] \wedge A_2[t, \bar{j}]) \vee A_1[j, i] \vee A_2[j, i] \vee B[j, i]$

For all edges $e = (i, j)$ that do not look like $(\bar{i}, \bar{j})$ or (j, i), $A[i, j] = A_1[i, j] \vee A_2[i, j] \vee B[i, j]$. Hence each output bit is a function of at most two bits from A_1, two bits from A_2, and one bit from B.

Combining Observation 4.7 and the fact that A_1 and A_2 are outputs of proof circuits $P_1 \in \mathcal{GC}$ and $P_2 \in \mathcal{GC}^R$, we have the claim. □

This completes proof of Lemma 4.8. □

Combining Lemma 4.3 and Lemma 4.8 completes the proof of Theorem 4.2.

Intuitively, reachability, and connectedness are global properties. Hence intuition suggests that STCONN should not have NC^0 proof systems. However, we have not been able to show this. In the following subsection, we study USTCONN—the undirected analogue of STCONN. We show that, contrary to the intuition that reachability is a global property, USTCONN indeed has a proof system computable in NC^0.

4.2. Undirected Reachability

In this section, we show that the set USTCONN of all undirected graphs with a path between two fixed vertices s and t has a proof system computable in NC^0.

Correcting an input that has no s-t path can be done locally by just adding the edge (s, t) to the output. However, detecting that the input does not have an s-t path with only local checks seems difficult. Always adding the (s, t) edge without checking for the absence of an $s - t$ path does not give us completeness. For this reason, we interpret the input differently.

The idea is as follows: We will first construct an NC^0 proof system $\mathcal{C}$ for the language CYCLES of all undirected graphs that are a union of edge disjoint cycles. Consider a circuit $C \in \mathcal{C}$ that outputs graphs on n vertices. The proof system we construct for USTCONN takes the adjacency matrix of the graph G output by C and does an EXOR with the edge (s, t) to obtain a graph G' (i.e., if the edge (s, t) was present, we remove it and if the edge (s, t) was not present, we add it). Note that if G contained a cycle with the (s, t) edge, then removing this edge leaves an s-t path in the resultant graph. If G did not contain an (s, t) edge, then the EXOR with (s, t) results in a graph with an s-t path of length 1. To get completeness, we take the upward closure of the graph G' by computing a bitwise OR with another input adjacency matrix (just like Lemma 4.1).

Now we formally define the language USTCONN, which is known to be in (and complete for) L ([Reingold 2008]). Our proof system will output adjacency matrices of all graphs that have a path between s and t and of no other graphs.

Define the following languages:

$$\text{USTCONN} = \left\{ A \in \{0,1\}^{n \times n} \mid \begin{array}{l} A \text{ is the adjacency matrix of an undirected graph } G \\ \text{where vertices } s = 1, t = n \text{ are in the same connected} \\ \text{component.} \end{array} \right\}$$

$$\text{CYCLES} = \left\{ A \in \{0,1\}^{n \times n} \mid \begin{array}{l} A \text{ is the adjacency matrix of an undirected graph} \\ G = (V, E) \text{ where E is the union of edge-disjoint} \\ \text{simple cycles.} \end{array} \right\}$$

(For simplicity, we will say $G \in$ USTCONN or $G \in$ CYCLES instead of referring to the adjacency matrices.)

THEOREM 4.11. *The language* USTCONN *has an* NC^0 *proof system.*

PROOF. We will need an addition operation on graphs: $G_1 \oplus G_2$ denotes the graph obtained by adding the corresponding adjacency matrices modulo 2. If A is a collection of graphs and $B = \text{UpClose}(A)$, then B is the collection of super-graphs obtained by adding edges. Note that (undirected) reachability is monotone and hence $\text{UpClose}(\text{USTCONN}) = \text{USTCONN}$.

Let $L_1 = \{G = G_1 \oplus (s,t) | G_1 \in \text{CYCLES}\}$ and $L_2 = \text{UpClose}(L_1)$. We show:

(1) $L_2 = \text{USTCONN}$.
(2) If L_1 has an NC^0 proof system, then L_2 has an NC^0 proof system.
(3) If CYCLES has an NC^0 proof system, then L_1 has an NC^0 proof system.
(4) CYCLES has an NC^0 proof system.

Proof of 1: We show that $L_1 \subseteq \text{USTCONN} \subseteq L_2$. Then applying upward closure, $L_2 = \text{UpClose}(L_1) \subseteq \text{UpClose}(\text{USTCONN}) = \text{USTCONN} \subseteq \text{UpClose}(L_2) = L_2$.

$L_1 \subseteq \text{USTCONN}$: Any graph $G \in L_1$ looks like $G = H \oplus (s,t)$, where $H \in \text{CYCLES}$. If $(s,t) \notin H$, then $(s,t) \in G$ and we are done. If $(s,t) \in H$, then s and t lie on a cycle C, and hence removing the (s,t) edge will still leave s and t connected by a path $C \setminus \{(s,t)\}$.

$\text{USTCONN} \subseteq L_2$: Let $G \in \text{USTCONN}$. Let ρ be an s-t path in G. Let $H = (V,E)$ be a graph such that $E =$ edges in ρ. Then, $G \in \text{UpClose}(\{H\})$. We can write H as $H' \oplus (s,t)$ where $H' = H \oplus (s,t)$. If $\rho = (s,t)$, then $E(H')$ is empty and hence $H' \in \text{CYCLES}$. Else $\rho \neq (s,t)$, and then $H' = H \oplus (s,t) = \rho \cup (s,t)$ since ρ is a simple path, and hence $H' \in \text{CYCLES}$. Either way, $H' \in CYCLES$ and so $H \in L_1$. Hence $G \in L_2$.

Proof of 2: Note that $\text{Minterms}(\text{USTCONN})$ is exactly the set of graphs where the edge set is a simple s-t path. We have seen that $L_1 \subseteq \text{USTCONN}$. As above, we can see that $H \in \text{Minterms}(\text{USTCONN}) \Rightarrow H \oplus (s,t) \in \text{CYCLES} \Rightarrow H \in L_1$. Statement 2 now follows from Lemma 4.1.

Proof of 3: Let A be the adjacency matrix output by the NC^0 proof system for CYCLES. The proof system for L_1 outputs A' such that $A'[s,t] = \overline{A[s,t]}$, and the rest of A' is same as A.

Proof of 4: This is of independent interest and is proved in Theorem 4.12 below.

This completes the proof of Theorem 4.11. □

We now construct NC^0 proof systems for the language CYCLES.

THEOREM 4.12. *The language* CYCLES *has an* NC^0 *proof system.*

PROOF. Let T be a family of graphs. We say that an edge e is *generated* by a sub-family $\mathcal{S} \subseteq T$ if the number of graphs in $\mathcal{S}$ which contain e is odd. We say that the family T generates a graph G if there is some sub-family $\mathcal{S} \subseteq T$ such that every edge in G is generated by $\mathcal{S}$ and no other edge is generated. We first observe that to generate every graph in the set CYCLES, we can set T to be the set of *all* triangles. Given any cycle, it is easy to come up with a set of triangles that generates the cycle; namely, take any triangulation of the cycle. Therefore, if we let T be the set of all triangles on n vertices, it will generate every graph in CYCLES. Also, no other graph will be generated because any set $\mathcal{S} \subseteq \text{CYCLES}$ generates a set contained in CYCLES (see Lemma 4.14 below). This immediately gives a proof system for CYCLES: Given a vector $x \in \binom{n}{3}$, we will interpret it as a subset $\mathcal{S}$ of triangles. We will output an edge e if it is a part of odd number of triangles in $\mathcal{S}$. Finally, because of the properties observed above, any graph generated in this way will be a graph from the set CYCLES.

Unfortunately, this is not an NC^0 proof system because to decide if an edge is generated, we need to look at $\Omega(n)$ triangles. For designing an NC^0 proof system, we need to come up with a set of triangles such that, for any graph $G \in \text{CYCLES}$, every edge in G is a part of $O(1)$ triangles.

So, on the one hand, we want the set of triangles to generate every graph in CYCLES, and, on the other hand, we need that for any graph $G \in$ CYCLES, every edge in G is a part of $O(1)$ triangles. We show that such a set of triangles indeed exists.

Thus our task now is to find a set of triangles $T \subseteq$ CYCLES such that:

(1) Every graph in CYCLES can be generated using triangles from T, that is,

$$\text{CYCLES} \subseteq \text{Span}(T) \triangleq \left\{ \sum_{i=1}^{|T|} a_i t_i \mid \forall i, a_i \in \{0,1\}, t_i \in T \right\}.$$

(2) Every graph generated from triangles in T is in CYCLES; $\text{Span}(T) \subseteq$ CYCLES.
(3) $\forall u, v \in [n]$, the edge (u, v) is contained in at most six triangles in T.

Once we find such a set T, then our proof system asks as input the coefficients a_i that indicate the linear combination needed to generate a graph in CYCLES. An edge e is present in the output if, among the triangles that contain e, an odd number of them have coefficient set to 1 in the input. By property 3, each output edge needs to see only $O(1)$ input bits and hence the circuit we build is NC^0. We will now find and describe T in detail.

Let the vertices of the graph be numbered from 1 to n. Define the length of an edge (i, j) as $|i - j|$. A triple $\langle i, j, k \rangle$ denotes the set of all graphs on n vertices that have exactly one triangle on vertices (u, v, w) where $|u - v| = i$, $|v - w| = j$, and $|u - w| = k$ and no other edges. So each graph in $\langle i, j, k \rangle$ is on n vertices but has exactly three edges that form a triangle with lengths i, j, and k. We now define the set

$$T = \bigcup_{i=1}^{n/2} \langle i, i, 2i \rangle \cup \langle i, i+1, 2i+1 \rangle.$$

Although the graphs in T have all n vertices, we sometimes refer to them as triangles when it is convenient.

OBSERVATION 4.13. *It can be seen that* $|T| \leq \frac{3}{2}n^2$. *This is linear in the length of the output, which has* $\binom{n}{2}$ *independent bits.*

We now show that T satisfies all properties listed earlier in reverse order.

***T* satisfies property 3**: Take any edge $e = (u, v)$. Let its length be $l = |u - v|$. e can either be the longest edge in a triangle or one of the two shorter ones. If l is even, then e can be the longest edgeone1 triangle in T and can be a shorter edge in at most four triangles in T. If l is odd, then e can be the longest edge for at most two triangles in T and can be a shorter edge in at most four triangles. Hence, any edge is contained in at most six triangles.

***T* satisfies property 2**: To see this, note first that $T \subseteq$ CYCLES. Next, observe the following closure property of cycles:

LEMMA 4.14. *For any* $G_1, G_2 \in$ CYCLES, *the graph* $G_1 \oplus G_2 \in$ CYCLES.

PROOF. A well-known fact about connected graphs is that they are Eulerian if and only if every vertex has even degree. The analogue for general (not necessarily connected) graphs is Veblen's theorem [Veblen 1912], which states that $G \in$ CYCLES if and only if every vertex in G has even degree.

Using this, we see that if for $i \in [2]$, $G_i \in$ CYCLES and if we add the adjacency matrices modulo 2, then degrees of vertices remain even and so the resulting graph is also in CYCLES. □

It follows that $\text{Span}(T) \subseteq$ CYCLES.

***T* satisfies property 1**: We will show that any graph $G \in$ CYCLES can be written as a linear combination of graphs in T. Define, for a graph G, the parameter $d(G) = (l, m)$ where l is the length of the longest edge in G and m is the number of edges in G that have length l. For graphs $G_1, G_2 \in$ CYCLES, with $d(G_1) = (l_1, m_1)$ and $d(G_2) = (l_2, m_2)$, we say $d(G_1) < d(G_2)$ if and only if either $l_1 < l_2$ holds or $l_1 = l_2$ and $m_1 < m_2$. Note that for any graph $G \in$ CYCLES with $d(G) = (l, m)$, $l \geq 2$ (unless G is the graph with no edges).

CLAIM 4.15. *Let $G \in$ CYCLES. If $d(G) = (2, 1)$, then $G \in T$.*

PROOF. It is easy to see that G has to be a graph that has a triangle with edge lengths 1, 1, and 2 and no other edges. All such graphs are contained in T by definition. □

LEMMA 4.16. *For every $G \in$ CYCLES with $d(G) = (l, m)$, either $G \in T$ or there is a $t \in T$, and $H \in$ CYCLES such that $G = H \oplus t$ and $d(H) < d(G)$.*

PROOF. If $G \in T$, then we are done. So now consider the case when $G \notin T$:

Let e be a longest edge in G. Let C be a cycle that contains e. Pick $t \in T$ such that e is the longest edge in t. G can be written as $H \oplus t$, where $H = G \oplus t$. From Lemma 4.14 and since $T \subseteq$ CYCLES, we know that $H \in$ CYCLES. Let t have the edges e, e_1, e_2. Any edge present in both G and t will not be present in H. Since $e \in G \cap t$, $e \notin H$. Lengths of e_1 and e_2 are both less than l since e was the longest edge in t. Hence the number of times an edge of length l appears in H is reduced by 1 and the new edges added (if any) to H (namely e_1 and e_2) have length less then l. Hence if $m > 1$, then $d(H) = (l, m-1) < d(G)$. If $m = 1$, then $d(H) = (l', m')$ for some m' and $l' < l$, and hence $d(H) < d(G)$. □

By repeatedly applying Lemma 4.16, we can obtain the exact combination of triangles from T that can be used to give any $G \in$ CYCLES.

Figure 5 illustrates such a repeated application of Lemma 4.16 on an example graph G shown at the top. The numbers in black next to vertices indicate vertex numbers and the numbers on the edges indicate the edge lengths. The final decomposition S has all graphs belonging to T. In Figure 6, the graphs in S are superimposed on each other. Any edge that appears twice in the graph formed by the superimposition does not exist in G since G is a sum modulo 2 of the graphs from S.

A more formal proof will proceed by induction on the parameter $d(G)$ and each application of Lemma 4.16 gives a graph H with a $d(H) < d(G)$ and hence allows for the induction hypothesis to be applied. The base case of the induction is given by Lemma 4.15. Hence T satisfies property 1.

Since T satisfies all three properties, we obtain an NC^0 proof system for CYCLES, proving the theorem. □

We have not been able to show an NC^0 proof system for the directed analogue of USTCONN. However, in the following proposition, we will define the language UNREACH that is known to be NL-complete ([Immerman 1988; Szelepcsényi 1988]) and show that it has an NC^0 proof system.

PROPOSITION 4.17. *The language* UNREACH *defined below has an* NC^0 *proof system under the standard adjacency matrix encoding.*

$$\text{UNREACH} = \left\{ A \in \{0,1\}^{n \times n} \mid \begin{array}{l} A \text{ is the adjacency matrix of a directed graph } G \text{ with} \\ \text{no path from } s = 1 \text{ to } t = n. \end{array} \right\}.$$

PROOF. As proof, we take as input an adjacency matrix A and an n-bit vector X with $X(s) = 1$ and $X(t) = 0$ hardwired. Intuitively, X is like a characteristic vector that represents all vertices that can be reached by s.

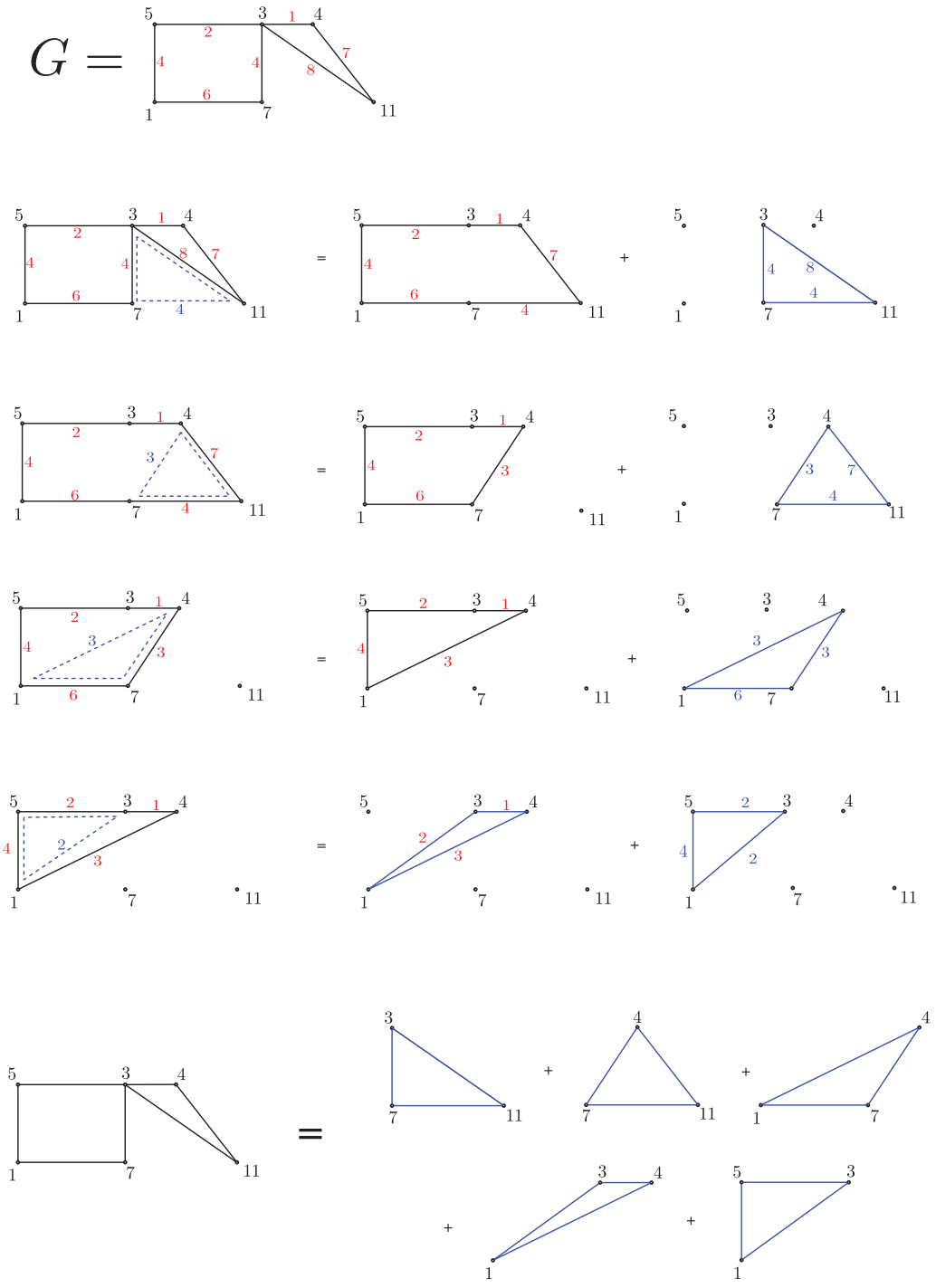

Fig. 5. Decomposition of G into graphs from T.

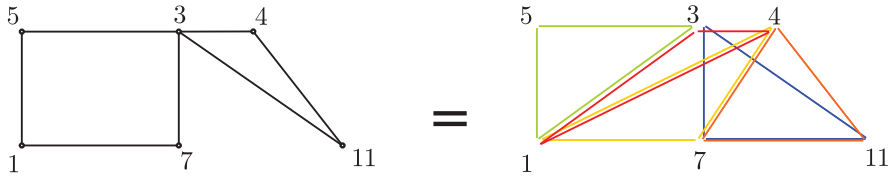

Fig. 6. Decomposition of G into graphs from T (superimposed).

The adjacency matrix B output by our proof system is as follows:

$$B[i,j] = \begin{cases} 1 & \text{if } A[i,j] = 1 \text{ and it is not the case that } X(i) = 1 \text{ and } X(j) = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Soundness. No matter what A is, X describes an s,t cut since $X(s) = 1$ and $X(t) = 0$ and $\forall i,j,\ X(i) = 1 \wedge X(j) = 0 \Rightarrow B[i,j] = 0$. So any graph output by the proof system will not have a path from s to t.

Completeness. For any $G \in$ UNREACH, use the adjacency matrix of G as A and give input X such that $X(v) = 1$ for a vertex v if and only if v is reachable from s. □

In the following section, we study the question of whether the power of AC^0 is necessary to compute proof systems for languages in NP.

5. PUSHING THE BOUNDS

We know that any language in NP has AC^0 proof systems. In the bounded fanin model, this corresponds to $O(\log n)$ depth. A natural question to ask is if depth $\Omega(\log n)$ is necessary. The class obtained by restricting AC^0 by not allowing $\wedge$ gates ($\vee$ gates) to have unbounded fanin and forcing all negations to be applied to leaves is called SAC^0 ($coSAC^0$). The class SAC^i was defined in Borodin et al. [1989]. It has been known that SAC^0 is not closed under complement (see Venkateswaran [1991]) and hence $SAC^0 \subsetneq AC^0$.

The following theorem implies that there is a language in NP for which any proof system generating it requires power at least that of SAC^0 or $coSAC^0$.

THEOREM 5.1 (SRIKANTH SRINIVASAN (PRIVATE COMMUNICATION)). *There is a language A in* NP *such that any bounded-fanin proof system for A needs $\Omega(\log n)$ depth.*

PROOF. Let $A \subseteq \{0,1\}^*$ be an error correcting code of constant rate (for each n, A has $2^{\Omega(n)}$ strings of length n) and linear distance (the Hamming distance between two words of length n is $\Omega(n)$) that can be efficiently computed. Such codes are known to exist. See, for example, Justesen [1972]. Suppose there is a proof system $C_n : \{0,1\}^m \longrightarrow \{0,1\}^n$ of depth d that outputs exactly the strings in A. Assume that C is non-degenerate. that is, for every input position i, $\exists x \in \{0,1\}^m$ such that $C(x) \neq C(x \oplus e_i)$. Note that $m \in \Omega(n)$ since A has constant rate ($|A \cap \{0,1\}^n| = 2^{\Omega(n)}$). Note that since each output bit is a function of at most 2^d input bits, it must be the case that there exists an input position i such that x_i is connected to at most $O(2^d)$ output positions. For this i, let x be an input such that $C(x) \neq C(x \oplus e_i)$. But since $C(x)$ and $C(x \oplus e_i)$ are both codewords in A, they must differ in at least $\Omega(n)$ positions since A is has linear distance. This implies that x_i is connected to at least $\Omega(n)$ output positions and this is true for all i. Hence, $d = \Omega(\log n)$. □

However, we note that proof systems for a big fragment of NP do not require the full power of AC^0. In particular, for every language in NP, an extremely simple padding yields another language with simpler proof systems.

THEOREM 5.2. *Let L be any language in* NP.

(*1*) *If L contains 0^*, then L has a proof system where negations appear only at leaf level, $\wedge$ gates have unbounded fanin, $\vee$ gates have $O(1)$ fanin, and the depth is $O(1)$. That is, L has a* coSAC^0 *proof system.*
(*2*) *If L contains 1^*, then L has a proof system where negations appear only at leaf level, $\vee$ gates have unbounded fanin, $\wedge$ gates have $O(1)$ fanin, and the depth is $O(1)$. That is, L has an* SAC^0 *proof system.*
(*3*) *The language $(\{1\} \cdot L \cdot \{0\}) \cup 0^* \cup 1^*$ has both* SAC^0 *and* coSAC^0 *proof systems.*

PROOF. Let L be a language in NP. Then there is a family of uniform polynomial-sized circuits (C_n), where each C_n has $q(n)$ gates, n standard inputs x, and $p(n)$ auxiliary inputs y, such that for each $x \in \{0,1\}^n$, $x \in L \iff \exists y : C_n(x,y) = 1$. We use this circuit to construct the proof system. The input to the proof system consists of words $x = x_1 \ldots x_n$, $y = y_1 \ldots y_{p(n)}$, $z = z_1 \ldots z_{q(n)}$. The intention is that y represents the witness such that $C_n(x,y) = 1$, and z represents the vector of values computed at each gate of C_n on input x, y. There are two ways of doing self-correction with this information:

—Check for consistency: Check that every gate $g_i = g_j \circ g_k$ satisfies $z_i = z_j \circ z_k$. Output the string w where $\langle w \rangle = \langle x \rangle \wedge (\bigwedge_{i=1}^{q(n)} [z_i = z_j \circ z_k])$. If even one gate is inconsistent, w equals 0^*, otherwise w is the input x that has been certified by y, z; hence w is in $L \cup 0^*$. Every string in L can be produced by giving witness y and consistent z. The expression shows that this is a coSAC^0 circuit.
—Look for an inconsistency: Find a gate $g_i = g_j \circ g_k$ where $z_i \neq z_j \circ z_k$. Output the string w where $\langle w \rangle = \langle x \rangle \vee (\bigvee_{i=1}^{q(n)} [z_i \neq z_j \circ z_k])$. If even one gate is inconsistent, then w equals 1^*, otherwise w equals the input x that has been certified by y, z; hence, w is in $L \cup 1^*$. Every string in L can be produced by giving suitable y, z. The expression shows that this is an SAC^0 circuit.

All three parts of the theorem follow directly from the above. □

Ideally, we would like to have a notion of a reduction $\leq$ such that if $A \leq B$ and if A needs $\Omega(d)$ depth in proof systems, then so does B. Such a notion was implicitly used in proving Theorem 4.11; we showed that a lower bound for CYCLES translated to a lower bound for USTCONN. However, part 3 of Theorem 5.2 suggests that for NC^0 proof systems in general, such "reductions" are necessarily rather fragile, and we do not yet see what is a reasonable and robust definition to adopt. Using some reduction-like techniques, we can give depth lower bounds for proof systems for some more languages. We collect some such results in Lemma 5.3 below; all start from the hardness of MAJ.

Using Lemma 4.1 and the known lower bound for MAJ from Beyersdorff et al. [2013], we can show that the following languages have no NC^0 proof systems:

LEMMA 5.3. *The following languages do not have* NC^0 *proof systems.*

(*1*) EXMAJ*, consisting of strings x with exactly $\lceil |x|/2 \rceil$ 1s.*
(*2*) EQUALONES $= \{xy \mid x, y \in \{0,1\}^*, |x| = |y|, |x|_1 = |y|_1\}$.
(*3*) GI $= \{G_1, G_2 \mid$ *Graph G_1 is isomorphic to graph G_2*$\}$.
Here we assume that G_1 and G_2 are specified via their 0-1 adjacency matrices and that 1s on the diagonal are allowed (the graphs may have self-loops).

PROOF.

(1) To show that EXMAJ does not have NC^0 proof systems, note that:
—The language MAJ does not have NC^0 proof systems (see Beyersdorff et al. [2013]).
—Minterms(MAJ) = EXMAJ; MAJ = UpClose(EXMAJ).
—Lemma 4.1 now implies EXMAJ does not have an NC^0 proof system.
By the same argument, EXMAJ restricted to even-length strings, call it EXMAJEVEN, has no NC^0 proof systems.

(2) We will show that if EQUALONES has an NC^0 proof system, then so does the language EXMAJEVEN. Consider the slice

$$\text{EQUALONES}^{=2n} = \{xy \mid |x| = |y| = n;\ \ x \text{ and } y \text{ have an equal number of 1s}\}.$$

If x, y are length-n strings, then $xy \in \text{EQUALONES}^{=2n}$ if and only if $xy' \in \text{EXMAJEVEN}$, where y' is the bitwise complement of y. Thus a depth d proof system for EQUALONES implies a depth $d+1$ proof system for EXMAJEVEN.

(3) Let G_1, G_2 be n-node isomorphic graphs with adjacency matrices A_1, A_2. Then (A_1, A_2) is in $\text{GI}^{=2n^2}$. Let y_b be the string appearing on the diagonal of A_b. Then $y_1y_2 \in \text{EQUALONES}^{=2n}$.

Conversely, for each $xy \in \text{EQUALONES}^{=2n}$ where $|x| = |y| = n$, the pair $(\text{Diag}(x), \text{Diag}(y))$ is in $\text{GI}^{=2n^2}$. (For an n-bit vector w, $\text{Diag}(w)$ is the $n \times n$ matrix with w on the diagonal and zeroes elsewhere.)

Thus a depth d proof system for GI implies a depth d proof system for EQUALONES. □

6. DISCUSSION

For MAJ, we have given a proof system with $O(\log \log n)$ depth (and $O(1)$ alternations), and it is known from Beyersdorff et al. [2013] that $\omega(1)$ depth is needed. Can this gap between the upper and lower bounds be closed?

Can we generalize the idea we use in Theorem 4.11 and apply it to other languages? In particular, can we obtain good upper bounds using this technique for the language STCONN?

Our construction from Theorem 3.6 can be generalized to work for languages accepted by growing-monoids or growing-non-uniform-automata with poly-log growth rate (see, e.g., Bedard et al. [1993]). Can we obtain good upper bounds for linearly growing automata?

In Krebs and Limaye [2013], proof systems computable in DLOGTIME are investigated. The techniques used there seem to differ considerably from those that work for small-depth circuits, especially $\text{poly log } AC^0$. Though in both cases each output bit can depend on at most $\text{poly log } n$ input bits, the circuit can pick an arbitrary set of $\text{poly log } n$ bits whereas a DLOGTIME proof system needs to write the index of each bit on the index tape using up $\log n$ time.

From the results of Beyersdorff et al. [2013] and this article, we now know languages complete for NC^1, L, NL, P, and NP with NC^0 proof systems. This gives further evidence that the complexity of the membership problem for a language L and the amount of resources needed to generate exactly L do not go hand in hand.

ACKNOWLEDGMENTS

The authors thank Vladimir Podolskii for many stimulating discussions and ideas leading up to the proof of Theorem 4.2 and for graciously allowing inclusion of the theorem in this manuscript. The authors thank Srikanth Srinivasan for describing the $\Omega(\log n)$ depth lower bound from Theorem 5.1 and for allowing its

inclusion in this manuscript. The authors also thank Markus Blaeser, who told us that REG-SCC is decidable and pointed us to the reference Grunsky et al. [2006]. This work was done while the fourth author was working in The Institute of Mathematical Sciences, Chennai, India.

REFERENCES

D. A. Barrington. 1989. Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. *J. Comput. Syst. Sci.* 38 (1989), 150–164.

D. A. Barrington and D. Thérien. 1988. Finite monoids and the fine structure of NC^1. *J. Assoc. Comput. Mach.* 35 (1988), 941–952.

Paul Beame and Toniann Pitassi. 2001. Propositional proof complexity: Past, present, and future. In *Current Trends in Theoretical Computer Science: Entering the 21st Century*, G. Paun, G. Rozenberg, and A. Salomaa (Eds.). World Scientific Publishing, 42–70.

F. Bedard, F. Lemieux, and P. McKenzie. 1993. Extensions to Barrington's M-program model. *Theor. Comput. Sci.* 107 (1993), 31–61.

Olaf Beyersdorff, Samir Datta, Andreas Krebs, Meena Mahajan, Gido Scharfenberger-Fabian, Karteek Sreenivasaiah, Michael Thomas, and Heribert Vollmer. 2013. Verifying proofs in constant depth. *ACM Trans. Comput. Theor.* 5, 1, Article 2 (May 2013), 23 pages. DOI:http://dx.doi.org/10.1145/2462896.2462898

Olaf Beyersdorff, Samir Datta, Meena Mahajan, Gido Scharfenberger-Fabian, Karteek Sreenivasaiah, Michael Thomas, and Heribert Vollmer. 2011a. Verifying proofs in constant depth. In *Proceedings of 36th Mathematical Foundations of Computer Science Symposium (MFCS)*. Lecture Notes in Computer Science, Vol. 6907. Springer, 84–95.

Olaf Beyersdorff, Johannes Köbler, and Sebastian Müller. 2011b. Proof systems that take advice. *Inform. Comput.* 209, 3 (2011), 320–332.

A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. 1989. Two applications of inductive counting for complementation problems. *SIAM J. Comput.* 18, 3 (June 1989), 559–578. DOI:http://dx.doi.org/10.1137/0218038

Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Annual Symposium on Theory of Computing (STOC)*. ACM, 151–158.

Stephen A. Cook and Jan Krajíček. 2007. Consequences of the provability of NP $\subseteq$ P/poly. *J. Symbol. Logic* 72, 4 (2007), 1353–1371.

Stephen A. Cook and Robert A. Reckhow. 1979. The relative efficiency of propositional proof systems. *J. Symbol. Logic* 44, 1 (1979), 36–50.

Mary Cryan and Peter Bro Miltersen. 2001. On pseudorandom generators in NC^0. In *Proceedings of 26th Symposium on Mathematical Foundations of Computer Science (MFCS)*. Springer, Marianske Lazne, Czech Republic, 272–284.

Merrick L. Furst, James B. Saxe, and Michael Sipser. 1984. Parity, circuits, and the polynomial-time hierarchy. *Math. Syst. Theor.* 17, 1 (1984), 13–27.

Igor Grunsky, Oleksiy Kurganskyy, and Igor Potapov. 2006. On a maximal NFA without mergible states. In *Proceecings of the Computer Science Symposium in Russia (CSR)*. Lecture Notes in Computer Science, Vol. 3967. Springer, 202–210.

Johan Håstad. 1986. Almost optimal lower bounds for small depth circuits. In *Proceedings of the 18th Symposium on Theory of Computing (STOC)*. IEEE Computer Society, 6–20.

Edward A. Hirsch. 2010. Optimal acceptors and optimal proof systems. In *Theory and Applications of Models of Computation (TAMC)*, Jan Kratochvl, Angsheng Li, Ji Fiala, and Petr Kolman (Eds.). Lecture Notes in Computer Science, Vol. 6108. Springer, Berlin, 28–39. DOI:http://dx.doi.org/10.1007/978-3-642-13562-0_4

Edward A. Hirsch and Dmitry Itsykson. 2010. On optimal heuristic randomized semidecision procedures, with application to proof complexity. In *Proceedings of the of 27th Annual Symposium on Theoretical Aspects of Computer Science (STACS) (LIPIcs)*, Vol. 5. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 453–464.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

N. Immerman. 1988. Nondeterministic space is closed under complementation. *SIAM J. Comput.* 17, 5 (1988), 935–938. DOI:http://dx.doi.org/10.1137/0217058

J. Justesen. 1972. Class of constructive asymptotically good algebraic codes. *IEEE Trans. Inform. Theor.* 18, 5 (1972), 652–656. DOI:http://dx.doi.org/10.1109/TIT.1972.1054893

Andreas Krebs and Nutan Limaye. 2013. DLOGTIME proof systems. In *FSTTCS (LIPIcs)*, Anil Seth and Nisheeth K. Vishnoi (Eds.), Vol. 24. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 189–200.

Pavel Pudlák. 2009. Quantum deduction rules. *Ann. Pure Appl. Logic* 157, 1 (2009), 16–29.

Omer Reingold. 2008. Undirected connectivity in log-space. *J. ACM* 55, 4, Article 17 (2008), 24 pages. (Originally appeared in STOC'05).

Nathan Segerlind. 2007. The complexity of propositional proofs. *Bull. Symbol. Logic* 13, 4 (2007), 417–481.

R. Szelepcsényi. 1988. The method of forced enumeration for nondeterministic automata. *Acta Inform.* 26, 3 (Nov. 1988), 279–284. DOI:http://dx.doi.org/10.1007/BF00299636

Oswald Veblen. 1912. An application of modular equations in analysis situs. *Ann. Math.* 14, 1/4 (1912), 86–94.

H. Venkateswaran. 1991. Properties that characterize LOGCFL. *J. Comput. System Sci.* 43, 2 (1991), 380–404. DOI:http://dx.doi.org/10.1016/0022-0000(91)90020-6

Received September 2014; revised February 2016; accepted June 2016