Arithmetizing classes around NC^1 and L

Nutan Limaye, Meena Mahajan, and B. V. Raghavendra Rao

The Institute of Mathematical Sciences, Chennai 600 113, India.
{nutan,meena,bvrr}@imsc.res.in

Abstract. The parallel complexity class NC^1 has many equivalent models such as bounded width branching programs. Caussinus et.al[11] considered arithmetizations of two of these classes, $\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$. We further this study to include arithmetization of other classes. In particular, we show that counting paths in branching programs over visibly pushdown automata has the same power as $\#\mathsf{BWBP}$, while counting proof-trees in logarithmic width formulae has the same power as $\#\mathsf{NC}^1$. We also consider polynomial-degree restrictions of SC^i, denoted sSC^i, and show that the Boolean class sSC^1 lies between NC^1 and L, whereas sSC^0 equals NC^1. On the other hand, $\#\mathsf{sSC}^0$ contains $\#\mathsf{BWBP}$ and is contained in FL, and $\#\mathsf{sSC}^1$ contains $\#\mathsf{NC}^1$ and is in SC^2. We also investigate some closure properties of the newly defined arithmetic classes.

1 Introduction

The parallel complexity class NC^1, comprising of languages accepted by logarithmic depth, polynomial size, bounded fan in Boolean circuits, is of fundamental interest in circuit complexity. NC^1 is known to be contained within logarithmic space L. The classes NC^1 and L have many equivalent characterizations. Bounded width branching programs BWBP, as well as bounded width circuits SC^0, (both of polynomial size), were shown by Barrington [7] to be equivalent to NC^1, while it it is folklore that poly size $O(\log n)$ width circuits SC^1 equals L.

However, arithmetizations of these classes are not necessarily equivalent. In [11], Caussinus et al proposed three arithmetizations of NC^1: (1) counting proof-trees in an NC^1 circuit, (2) computation by a poly size log depth circuit over $+$ and $\times$, and (3) counting paths in a nondeterministic bounded width branching program. It is straightforward to see that the first two definitions of function classes, over $\mathbb{N}$, coincide (see for instance [26, 28]); and this class is denoted $\#\mathsf{NC}^1$. It is shown in [11] that the third class, $\#\mathsf{BWBP}$, is contained in $\#\mathsf{NC}^1$, though the converse inclusion is still open. (However, the arithmetizations over $\mathbb{Z}$ are shown to coincide.) Also, using the programs over monoids framework, [11] observe that $\#\mathsf{BWBP}$ equals $\#\mathsf{BP}$-NFA, the class of functions that count the number of accepting paths in a nondeterministic finite-state automaton NFA when run on the output of a deterministic branching program. It is known (see e.g. [3, 28]) that $\#\mathsf{NC}^1$ has Boolean poly size circuits of depth $O(\log n \log^* n)$ and is thus very close to NC^1. It follows from more recent results [12] that $\#\mathsf{NC}^1$ is contained in FL; see e.g. [3].

We continue this study here (and also extend it to L) by arithmetizing other Boolean classes also known to be equivalent to NC^1. The first extension we consider is from

NFA to VPA. Visibly pushdown automata (VPA) are ϵ-moves-free pushdown automata whose stack behaviour (push/pop/no change) is dictated solely by the input letter under consideration. They are also referred to as input-driven pda, and have been studied in [19, 9, 15, 6, 5]. In [15], languages accepted by such pda are shown to be in NC^1, while in [6] it is shown that such pda can be determinized. Thus they lie properly between regular languages and deterministic context-free languages, and membership is complete for NC^1. The arithmetic version we consider is #BP-VPA, counting the number of accepting paths in a VPA, when run on the output of a deterministic branching program. Clearly, this contains #BP-NFA; we show that in fact the two are equal. Thus adding a stack to an NFA but restricting its usage to a visible nature adds no power to the closure of the class under projections.

The next class we consider is arithmetic formulae. It is known that formulae F (circuits with fanout 1 for each gate) and even logarithmic width formulae LWF have the same power as NC^1 [17]. Applying either of definition (1) or (2) above to formulae give the function classes #F and #LWF. It is known [10] that $\#\mathsf{LWF} \subseteq \#\mathsf{F} = \#\mathsf{NC}^1$. We show that this is in fact an equality. Thus even in the arithmetic setting, LWF have the full power of NC^1.

Next we consider bounded width circuits. SC is the class of polynomial size poly logarithmic width (width $O(\log^i n)$ for SC^i) circuits, and corresponds in the uniform setting to a simultaneous time-space bound. (SC stands for Steve's Classes, named after Stephen Cook who proved the first non-trivial result about polynomial time log-squared space PLoSS, i.e. SC^2, in [13]. See for instance [18]). It is known that SC^0 equals NC^1 [7]. However, this equality provably does not carry over to the arithmetic setting, since it is easy to see that even SC^0 over $\mathbb{N}$ can compute values that are infeasible (needing more than polynomially long representation). So we consider the restriction to polynomial degree, denoted by sSC^0, before arithmetizing to get $\#\mathsf{sSC}^0$. We observe that in the Boolean setting, this is not a restriction at all; sSC^0 equals NC^1 as well. However, the arithmetization does not appear to collapse to either of the existing classes. We show that $\#\mathsf{sSC}^0$ lies between #BWBP and FL.

The polynomial-degree restriction of SC^0 immediately suggests a similar restriction on all the SC^i classes. We thus explore the power of sSC^i and sSC, the polynomial-degree restrictions of SC^i and SC respectively, and their corresponding arithmetic versions $\#\mathsf{sSC}^i$ and #sSC. This restriction automatically places the corresponding classes in LogCFL and #LogCFL, since LogCFL is known to equal languages accepted by polynomial size polynomial degree circuits [24, 22], and since the arithmetic analogue also holds [26, 20]. Thus we have a hierarchy of circuit classes between NC^1 and LogCFL. Other hierarchies sitting in this region are poly size branching programs of polylog width, limited by NL in LogCFL, and poly size log depth circuits with AND fan in 2 and OR fan in polylog, limited by SAC^1 which equals LogCFL [25]; see [27]. In both of these hierarchies, [27] establishes closure under complementation. For sSC^i, we have a weaker result: co-sSC^i is contained in sSC^{2i}.

It is not clear what power the Boolean class sSC^1 possesses: is it strong enough to equal SC^1, or is the polynomial degree restriction crippling enough to bring it down to SC^0=NC^1? We show that all of $\#\mathsf{NC}^1$ is captured by $\#\mathsf{sSC}^1$, which is contained in Boolean SC^2. Note that the maximal fragments of NC hitherto known to be in SC

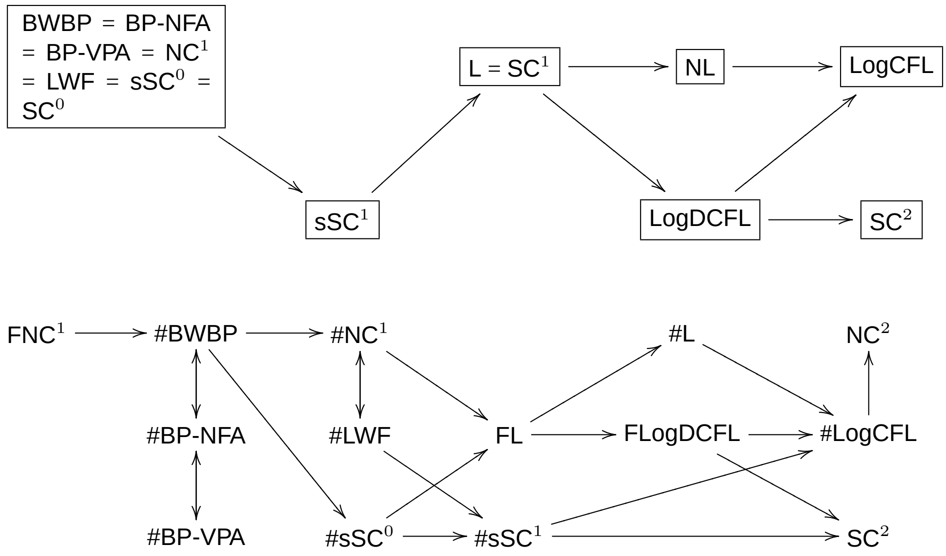

Fig. 1. Boolean classes and their arithmetizations

were LogDCFL [23, 14, 16] and randomized log space RL [21]; we do not know how this fragment compares with them. In fact, turning the question around, studying sSC is an attempt to understand fragments of NC that lie within SC.

Our main results can be summarized in Figure 1. It shows that corresponding to Boolean NC^1, there are three naturally defined arithmetizations, while the correct arithmetization of L is still not clear. We also show that the three arithmetizations of NC^1 coincide under modulo tests, for any fixed modulus.

A key to understanding function classes better is to investigate their closure properties. We present some such results concerning #sSC^i.

This paper is organized as follows. Definitions and notation are presented in Section 2. Sections 3 and 4 present the bounds on #BP-VPA and #LWF, respectively. Section 5 introduces and presents bounds involving sSC^i and #sSC^i. Some closure properties of these classes are presented in Section 6, where also the collapse of the modulus test classes NC^1= $\oplus$NC^1 = $\oplus$BWBP = $\oplus$sSC^0 follows.

2 Preliminaries

By NC^1 we denote the class of languages which can be accepted by a family $\{C_n\}_{n\geq 0}$ of polynomial size $O(\log n)$ depth bounded circuits, with each gate having a constant fan-in. A branching program is a layered acyclic graph G with edges labeled by constants or literals, and with two special vertices s and t. It accepts an input x if it has an $s \rightsquigarrow t$ path where each edge is labeled by a true literal or the constant 1. BWBP denotes the class of languages that can be accepted by polynomial size bounded width branching programs. BWC is the class of languages which can be accepted by a family $\{C_n\}_{n\geq 0}$ of constant width, polynomial size circuits, where *width* of a circuit is the maximum number of gates at any level of the circuit. A branching program can be equivalently viewed as a skew circuit i.e, a circuit in which each AND gate has at most one input wire that is not a circuit input; hence BWBP is in BWC. SC^i is the class of languages which can be accepted by a family $\{C_n\}_{n\geq 0}$ of polynomial size circuits of width $O((\log n)^i)$. Thus we have by definition, $\mathsf{BWC} = \mathsf{SC}^0$. For the class SC^i we assume, without loss of generality, that every gate has fan-in $O(1)$ (fan-in $f = O((\log n)^i)$ is replaced by a width $O(1)$, depth $O(f)$ circuit). LWF is the class of languages which can be accepted by a family $\{F_n\}_{n\geq 0}$ of polynomial size formulae with width bounded by $O(\log n)$. Without the width bound, denote the family of poly size formula by F.

For defining branching programs over automata, we follow notation from [11]. A nondeterministic automaton is a tuple of the form $(Q, \Delta, q_0, \delta, F)$, where Q is the finite set of states, Δ is the input alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states and $\delta :\ Q \times \Sigma \rightarrow \mathcal{P}(Q)$.

A projection $P = (\Sigma, \Delta, S, B, E)$ over Δ is a family $P = (P_n)_{n\in N}$ of n-projections over Δ, where an n-projection over Δ is a finite sequence of pairs (i, f) with $1 \leq i \leq n$ and $f : \Sigma \rightarrow \Delta$. The length of the sequence is denoted by S_n, its j-th instruction is denoted by $(B_n(j), E_n(j))$ where $S : \mathbb{N} \rightarrow \mathbb{N}$, $B : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $E : \mathbb{N} \times \mathbb{N} \rightarrow \Delta^{\Sigma}$. B pulls out a letter $x_{B_{|x|}(j)} \in \Sigma$ from the input x and E projects it to a letter in the alphabet Δ. Thus the string $x \in \Sigma^*$ is projected to a string $P(x) \in \Delta^*$. FDLOGTIME uniformity for the projections is assumed.

A branching program over a nondeterministic automaton $N = (Q, \Delta, q_0, \delta, F)$ is a projection $P = (\Sigma, \Delta, S, B, E)$. It accepts $x \in \Sigma^*$ if N accepts the projection of x. BP-NFA is the class of all languages recognized by uniform polynomial length branching programs over a nondeterministic automaton[1].

A visibly pushdown automaton (VPA) is a pda $M = (Q, Q_{in}, \Delta, \Gamma, \delta, Q_F)$ working over an input alphabet Δ that is partitioned as $(\Delta_c, \Delta_r, \Delta_{int})$. Q is a finite set of states, $Q_{in}, Q_F \subseteq Q$ are the sets of initial and final states respectively, Γ is the stack alphabet containing a special bottom-of-stack marker $\perp$, and acceptance is by final state. The transition function δ is constrained so that: If $a \in \Delta_c$, then $\delta(p, a) = (q, \gamma)$ (push move, independent of top-of-stack). If $a \in \Delta_r$, then $\delta(p, a, \gamma) = q$ (pop move), and $\delta(p, a, \perp) = q$ (pop on empty stack). If $a \in \Delta_{int}$, then $\delta(p, a) = q$ (internal move, independent of top-of-stack). The input letter completely dictates the stack movement. Also the pda is assumed to be ϵ-move-free, while δ is allowed to be non-deterministic.

BP-VPA is the class of all languages recognized by uniform polynomial length branching programs over a VPA.

In [7], Barrington showed that NC^1= BWBP= BWC. As observed in [11], BWBP coincides with BP-NFA; thus NC^1= BP-NFA. Istrail and Zivkovic showed in [17] that NC^1= LWF. In [15], Dymond showed that acceptance by VPAs can be checked in NC^1, and hence BP-VPA= NC^1. Thus

Lemma 1 ([7, 11, 17, 15]). NC^1= BWBP= SC^0= LWF= BP-NFA= BP-VPA

[1] In [11], this class is called BP. We introduce this new notation to better motivate the next definition, of BP-VPA.

The corresponding arithmetic classes are defined as follows:

$$\#\mathsf{BWBP} = \{f : \{0,1\}^n \to \mathbb{N} \mid f = \#s \rightsquigarrow t \text{ paths in a } \mathsf{BWBP}\}$$

$$\#\mathsf{NC}^1 = \left\{ f : \{0,1\}^n \to \mathbb{N} \;\middle|\; \begin{array}{l} f \text{ can be computed by a poly size } O(\log n) \text{ depth} \\ \text{bounded fan in circuit over } \{+, \times, 1, 0, x_i, \overline{x_i}\}. \end{array} \right\}$$

$$\#\mathsf{BP\text{-}NFA} = \left\{ f : \{0,1\}^n \to \mathbb{N} \;\middle|\; \begin{array}{l} f(x) = \#\text{accept}(P_n, x) \text{ for some uni-} \\ \text{form poly length } \mathsf{BP}\ P \text{ over an } \mathsf{NFA}\ N \end{array} \right\}$$

Here, $\#\text{accept}(P, x)$ denotes the number of distinct accepting paths of N on the projection of x, $P(x)$.

For each of these counting classes, the corresponding Diff classes are defined by taking the difference of two # functions, while the Gap classes are defined by taking closure of the class under subtraction (equivalently, by allowing the constant -1 in the circuit). For reasonable classes (in particular, for the classes we consider), Diff and Gap coincide, see [28].

Though the above classes are all equal in the Boolean setting, in the arithmetic setting the equivalences are not established, and strict containments are also not known. The best known relationships among these classes are as below.

Lemma 2 ([11]).
$\mathsf{FNC}^1 \subseteq \#\mathsf{BWBP} = \#\mathsf{BP\text{-}NFA} \subseteq \#\mathsf{NC}^1 \subseteq \mathsf{GapBWBP} = \mathsf{GapNC}^1 \subseteq \mathsf{L}$.

3 Counting accepting runs in visibly pushdown automata

We introduce a natural arithmetization of BP-VPA, by counting the number of accepting paths in a VPA rather than in an NFA. The definition mimics that of BP-NFA. Given a uniform polynomial length branching program P over a VPA M, the number of distinct accepting paths of M on the projection of x is denoted by #accept(P, x).

Definition 1. $$\#\mathsf{BP\text{-}VPA} = \left\{ f : \{0,1\}^n \to \mathbb{N} \;\middle|\; \begin{array}{l} f(x) = \#\textit{accept}(P_n, x) \textit{ for some} \\ \textit{uniform poly length } \mathsf{BP}\ P \textit{ over a} \\ \mathsf{VPA}\ M \end{array} \right\}$$

The main result of this section is that adding a visible pushdown to an NFA adds no power to the corresponding counting class. That is,

Theorem 1. #BP-NFA= #BP-VPA

Proof. #BP-NFA$\subseteq$ #BP-VPA: Obvious from the definition, since a VPA can simulate a NFA for *any* partition of the input.

#BP-VPA$\subseteq$ #BP-NFA: For this direction, we use the fact that #BP-NFA equals #BWBP (Lemma 1) and place #BP-VPA in #BWBP.

Let $f \in$ #BP-VPA. There exists a uniform polynomial length branching program P over a VPA $M = (Q, \Delta, Q_{in}, \Gamma, \delta, Q_F)$. Let input w be projected to $P(w) = x \in \Delta^n$, where $\Delta = (\Delta_c, \Delta_r, \Delta_{int})$, $|x| = n$. So $f(w) = \#acc_M(x)$.

The strategy is as follows. We first construct an equivalent VPA M' that never needs to perform a pop on an empty stack. A TC^0 circuit transforms x to a string y over a larger alphabet, such that #$acc_M(x)$ = #$acc_{M'}(y)$. This latter quantity, #$acc_{M'}(y)$, is counted by paths in a BWBP G whose edges are labeled by NC^1 predicates involving M' and y. Thus each edge can be replaced by an equivalent BWBP, and the whole graph is still a BWBP.

The VPA $M' = (Q', \Delta'', Q'_{in}, \Gamma', \delta', Q'_F)$ is essentially the same as M. It has two new input symbols A, B, and a new stack symbol X. A is a push symbol on which X is pushed, and B is a pop symbol on which X is expected and popped. M' has a new state q' that is the only initial state. M' expects an input from $A^*\Delta^*B^*$. On the prefix of A's it pushes X's. When it sees the first letter from Δ, it starts behaving like M. The only exception is when M performs a pop move on $\perp$, M' can perform the same move on $\perp$ or on X. On the trailing suffix of B's it pops X's. It is straightforward to design δ' from δ.

The TC^0 circuit does the following. It counts the difference d between the number of push and pop symbols in $A^n x$. It then outputs $y = A^n x B^d$. By the way M' is constructed, it should be clear that #$acc_M(x)$ = #$acc_{M'}(y)$ and that M', on y, never pops on an empty stack. In fact y is *well-matched*, i.e. for every push there exists a corresponding pop and vice versa.

We now describe the layered directed acyclic graph $G = (V, E)$, with nodes s, t such that $\#_G s \rightsquigarrow t$ = #$acc_{M'}(y)$. It will be clear that G can be constructed in NC^1.

Let $V = \{(q, X, i) \mid q \in Q' \cup \{g\}, X \in \Gamma' \cup \{\perp\}, (g \notin Q'), 0 \le i \le (n+1)\}$. At layer 0 we need only the vertex labeled $s = (q', \perp, 0)$. Layer i, for $1 \le i \le n$, contains vertices of the form (q, X, i) $\forall q \in Q'$ and $\forall X \in \Gamma'$. At layer $n+1$, we keep only $t = (g, \perp, n+1)$. This describes the vertex set of G. Note that every layer has a constant number of vertices. The vertex labels are intended to denote *surface configurations* of M', i.e. state, top-of-stack, tape head position. Since VPAs have a one-way tape and no ϵ moves, the tape head position is also the time-stamp.

Now we describe the edge set of G. The edges should trace out computations of M'. Thus if $(q, Z') \in \delta'(p, y_i)$ for $y_i \in \Delta'_c$, then we put an edge from $(p, Z, i-1)$ to (q, Z', i) for each Z. Similarly, if $(q, Z) \in \delta'(p, y_i)$ for $y_i \in \Delta'_{int}$, then we put an edge from $(p, Z, i-1)$ to (q, Z, i) for each Z. The only problematic case is when $y_i \in \Delta'_r$. If $q \in \delta'(p, y_i, Z)$, then we want to put an edge from $(p, Z, i-1)$ to (q, Z', i). But we don't know Z'; it is the stack symbol that will be uncovered when Z is popped.

In TC^0, first find the matching symbol j, $j < i$, such that $y_j \in \Delta_c$ and the symbol Z pushed by M' while reading y_j is popped while reading y_i. Because of the padding of x to y, this matching symbol is uniquely defined. Note that the stack never dips below Z between $y_{j+1}...y_{i-1}$. M' can go from $(p, Z, i-1)$ to (q, Z', i) and hence we should put this edge in G if and only if for some $p', p" \in Q'$,

(a) $(p'', Z) \in \delta'(p', y_j)$ (and hence there is an edge from $(p', Z', j-1)$ to (p'', Z, j)),
(b) M' can move from (p'', Z) to (p, Z) on reading the string $y_{j+1}...y_{i-1}$ (and without dipping below Z on the stack), and
(c) $q \in \delta'(p, y_i, Z)$.
(d) M' can reach the configuration $(p', Z', j-1)$ starting from $s = (q', \perp, 0)$ and reading the string $y_1...y_{j-1}$.

(a) and (c) are determined by a simple lookup of δ'. (b) and (d) can be determined in NC^1, and hence by a deterministic BWBP, since the following is established in [15].

Proposition 1 ([15]). *Determining whether a pair of height-matched surface configurations of a* VPA *is realizable (one is reachable from the other without dipping below the given stack top) is in* NC^1.

(b) is already in the required form to use this result. To check (d), we need to pad the string $y_1...y_{j-1}$ with appropriate number of extra copies of B to get a well-matched string, and then check realizability. As argued above, this can be done in TC^0. Thus, the AND of the four conditions is recognised by a deterministic BWBP. We insert this BWBP in G, identifying its start and sink vertices with $(p, Z, i-1)$ and (q, Z', i).

Also put all the edges of the form $\langle (p, \perp, n),(g, \perp, n+1)\rangle$ provided $p \in F'$

This completely describes the graph G. We need to prove that the number of accepting paths in the VPA M equals the number of paths from s to t in G. This can be done through simple induction. □

4 Counting proof trees in (log width) formula

We show that the result of [17], asserting that log width formula capture NC^1, holds in the arithmetized setting as well. This result is crucially used in showing Theorem 4.

Definition 2. $\#\mathsf{F} = \left\{ f : \{0,1\}^n \to \mathbb{N} \mid \begin{array}{l} f \textit{ can be computed by a poly size formula} \\ \textit{over } \{+, \times, 1, 0, x_i, \overline{x_i}\}. \end{array} \right\}$

$\#\mathsf{LWF} = \left\{ f : \{0,1\}^n \to \mathbb{N} \mid \begin{array}{l} f \textit{ can be computed by a poly size } O(\log n) \textit{ width} \\ \textit{formula over } \{+, \times, 1, 0, x_i, \overline{x_i}\}. \end{array} \right\}$

Theorem 2. $\#\mathsf{LWF} = \#\mathsf{F} = \#\mathsf{NC}^1$

Proof. Clearly, $\#\mathsf{LWF} \subseteq \#\mathsf{F}$. It follows from [10] (see also [3]) that $\#\mathsf{F}$, and hence also $\#\mathsf{LWF}$, is in $\#\mathsf{NC}^1$. To show that $\#\mathsf{NC}^1$ is in $\#\mathsf{LWF}$, we observe that the construction of Lemma 2 in [17], establishing that $\mathsf{NC}^1 \subseteq \mathsf{LWF}$, preserves proof-trees. □

5 Polynomial degree small-width circuits and their arithmetization

We now consider arithmetization of SC. A straightforward arithmetization of any Boolean circuit class over $(\wedge, \vee, x_i, \overline{x_i}, 0, 1)$ is to replace each $\vee$ gate by a $+$ gate and each $\wedge$ gate by a $\times$ gate. In the case of SC^0 (SC^i in general), this enables the circuit to compute infeasible values (i.e exponential sized values), which makes the class uninteresting. Hence we propose bounded degree versions of these classes and then arithmetize them. The degree of a circuit is the maximum degree of any gate in it, where the degree of a leaf is 1, the degree of an $\vee$ or $+$ gate is the maximum of the degrees of its children, and the degree of a $\wedge$ or $\times$ gate is the sum of the degrees of its children.

Definition 3. sSC^i *is the class of languages accepted by Boolean circuits of polynomial size,* $O(\log^i n)$ *width and polynomial degree.*

$\#\mathsf{sSC}^i$ *is the class of functions computed by arithmetic circuits of polynomial size,* $O(\log^i n)$ *width and polynomial degree. Equivalently, it is the class of functions counting the number of proof trees in an* sSC^i *circuit.*

$$\mathsf{sSC} = \bigcup_{i \geq 0} \mathsf{sSC}^i \qquad\qquad \#\mathsf{sSC} = \bigcup_{i \geq 0} \#\mathsf{sSC}^i$$

Note that SC circuits can have internal NOT gates as well; moving the negations to the leaves only doubles the width. However, when we restrict degree as in sSC, we explicitly disallow internal negations. The circuits have only AND and OR gates, and constants and literals appear at leaves.

It is known that polynomial-size circuits of polynomial degree, irrespective of width or depth, characterize LogCFL, which is equivalent to semi-unbounded log depth circuits SAC^1, and hence is contained in NC^2 [24, 22, 25]. This equivalence also holds in the arithmetic settings for # and for Gap, see [26, 20, 4]. Thus

Proposition 2. *For all $i \geq 0$,*
(1) $\mathsf{sSC}^i \subseteq \mathsf{LogCFL}$. *(2)* $\#\mathsf{sSC}^i \subseteq \#\mathsf{LogCFL}$. *(3)* $\mathsf{GapsSC}^i \subseteq \mathsf{GapLogCFL}$

A branching program can be viewed as a skew circuit, and a skew circuit's degree is bounded by its size; so BWBP is contained in sSC^0. But SC^0 = BWBP = NC^1. Thus

Proposition 3. sSC^0 = SC^0 = NC^1.

We do not know whether such an equality (sSC^i = SC^i) holds at any other level. If it holds for any $i \geq 2$, it would bring a larger chunk of SC into the NC hierarchy.

We now show that the individual bits of each $\#\mathsf{sSC}^i$ function can be computed in polynomial time using $O(\log^{i+1})$ space. However, the Boolean circuits constructed may not have polynomial degree.

Theorem 3. *For all $i \geq 0$,* $\#\mathsf{sSC}^i \subseteq \mathsf{GapsSC}^i \subseteq \mathsf{SC}^{i+1}$

Proof. We show how to compute $\#\mathsf{sSC}^i$ in SC^{i+1}. The result for Diff and hence Gap follows since subtraction can be performed in SC^0.

Let $f \in \#\mathsf{sSC}^i$. Let d be the degree bound for f. Then the value of f can be represented using $d \in n^{O(1)}$ bits. By the Chinese Remainder Theorem, f can be computed exactly from its residues modulo the first $O(d^{O(1)})$ primes, each of which has $O(\log d) = O(\log n)$ bits. These primes are small enough that they can be found in log space. Further, due to [12], the computation of f from its residues can also be performed in L= SC^1; see also [2]. If the residues can be computed in SC^k, then the overall computation will also be in SC^k because we can think of composing the computations in a sequential machine with a simultaneous time-space bound.

It thus remains to compute $f \bmod p$ where p is a small prime. Consider a bottom-up evaluation of the $\#\mathsf{sSC}^i$ circuit, where we keep track of the values of all intermediate nodes modulo p. The space needed is $\log p$ times the width of the circuit, that is, $O(\log^{i+1} n)$ space, while the time is clearly polynomial. □

In particular, bits of an $\#\mathsf{sSC}^0$ function can be computed in SC^1, which equals L. On the other hand, similar to the discussion preceding Proposition 3, we know that #BWBP is contained in $\#\mathsf{sSC}^0$. Thus

Corollary 1. $\mathsf{FNC}^1 \subseteq \#\mathsf{BWBP} \subseteq \#\mathsf{sSC}^0 \subseteq \mathsf{FL}$.
$\mathsf{GapNC}^1 = \mathsf{GapBWBP} \subseteq \mathsf{GapsSC}^0 \subseteq \mathsf{FL}$.

We cannot establish any direct connection between #sSC^0 and #NC^1. Thus this is potentially a third arithmetization of the Boolean class NC^1, the other two being #BWBP and #NC^1.

We also do not know whether sSC^1 properly restricts SC^1=L. Even if it does, it cannot fall below NC^1, since NC^1 = sSC^0(Proposition 3). We note that this holds in the arithmetic setting as well:

Theorem 4. #$\mathsf{NC}^1 \subseteq$ #sSC^1.

Proof. ¿From Theorem 2, we know that #NC^1 equals #LWF. But an LWF has log width and has poly degree since it is a formula; hence #LWF is in #sSC^1. □

Since the levels of sSC are sandwiched between NC^1 and LogCFL, both of which are closed under complementation, it is natural to ask whether the levels of sSC are also closed under complement. While we are unable to show this, we show that for each i, co-sSC^i is contained in sSC^{2i}; thus sSC as a whole is closed under complement.

Theorem 5. *For each $i \geq 1$, co-sSC^i is contained in sSC^{2i}.*

Proof. Consider the proof of closure under complement for LogCFL, from [8]. This is shown by considering the characterization of LogCFL as semi-unbounded log depth circuits, and applying an inductive counting technique to such circuits. Our approach for complementing sSC^i is similar: use inductive counting as applied by [8]. However, one problem is that the construction of [8] uses NC^1 circuits for threshold internally, and if we use these directly, the degree will blow up. So for the thresholds, we use the construction from [27]. A careful analysis of the parameters then yields the result.

Let C_n be a boolean circuit of length l, width $w = O(\log^i n)$ and degree p. Without loss of generality, assume that C_n has only $\vee$ gates at odd levels and $\wedge$ gates at even levels. Also assume that all gates have fan in 2 or less. If an input literal is read by a gate at level k, the literal is counted as a gate at level $k-1$. We construct a boolean circuit C'_n, which computes $\bar{C_n}$. C'_n contains a copy of C_n. Besides, for each level k of C_n, C'_n contains the gates $cc(g|c)$ where g is a gate at level k of C_n and $0 \leq c \leq w$, and gates $count(c,k)$ for $0 \leq c \leq w$. These represent the conditional complement of g assuming the count at the previous level is c, and verifying that the count at level k is c, and are defined as follows:

$$cc(g|c) = \begin{cases} cc(a_1|c) \vee cc(a_2|c), & \text{if } g = a_1 \wedge a_2 \\ Th^c(b_1, \cdots, b_j), & \text{if } g = a_1 \vee a_2 \end{cases}$$

where $b_1, \cdots, b_j$ range over all gates at the previous level except a_1 and a_2.

$$count(c,k) = \begin{cases} Th1(c,k) \wedge \bigvee_{d=0}^{w}[count(d,k-1) \wedge Th0(c,k,d)] \text{ if } k > 0 \\ 1 \text{ if } k = 0, c = \text{\# of inputs with value 1 at level 0} \\ 0 \text{ otherwise} \end{cases}$$

Th^c is the c-threshold value of its inputs , $Th1(c,k) = Th^c$ of all original gates at current level, $Th0(c,k,d)$ is Th^{k-c} of all $cc(g|d)$ at the current level. Finally, the output gate of C'_n is $comp(g) = \bigvee_{c=0}^{w} Count(c, l-1) \wedge cc(g|c)$, where g is the output gate of C_n, at level l. Correctness follows from the analysis in [8].

A crucial observation, used also in [8], is that any root-to-leaf path goes through at most two threshold blocks.

To achieve small width and small degree, we have to be careful about how we implement the thresholds. Since the inputs to the threshold blocks are computed in the circuit, we need monotone constructions. We do not know whether monotone NC^1 is in monotone sSC^0. But for our purpose, the following is sufficient: Lemma 4.3 of [27] says that any threshold on K bits can be computed by a monotone branching program of width $O(K)$ and size $O(K^2)$ (hence degree $O(K^2)$). The thresholds we use have $K = O(w^2)$. The threshold blocks can be staggered so that the $O(w^2)$ extra width appears as an additive rather than multiplicative factor. Hence the width of C'_n is $O(w^2)$.

Let q be the degree of a threshold block; $q \in O(K^2) \in O(w^4)$. If the inputs to a threshold block come from computations of degree p, then the overall degree is pq. A $cc(g|c)$ gate is a threshold block applied to gates of C_n at the previous level, and these gates all have degree at most p. So the $cc(g|c)$ gate has degree at most pq. Also, the degree of a $count(c, k)$ gate is bounded by the sum of (1) the degree of a $count(c, k-1)$ gate, (2) the degree of a threshold block applied to gates of C_n, and (3) the degree of a threshold block applied to $cc(g|c)$ gates. Hence it is bounded by $p^{O(1)}w^{O(1)}l$, where l is the depth of C_n. Thus, the entire circuit has polynomial degree. $\square$

6 Extensions and Closure Properties

In this section, we show that some closure properties that hold for $\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$ also hold for $\#\mathsf{sSC}^0$. (Construction details are omitted due to space restrictions.) The simplest closures are under addition and multiplication, and it is straightforward to see that $\#\mathsf{sSC}^0$ is closed under these. The next are weak sum and weak product: add (or multiply) the value of a two-argument function over a polynomially large range of values for the second argument. (See [11, 28] for formal definitions.) A simple staggering of computations yields:

Lemma 3. *For each $i \geq 0$, $\#\mathsf{sSC}^i$ is closed under weak sum and weak product.*

$\#\mathsf{NC}^1$ and $\#\mathsf{BWBP}$ are known to be closed under decrement $f \ominus 1 = \max\{f-1, 0\}$ and under division by a constant $\lfloor \frac{f}{m} \rfloor$. ([1] credits Barrington with this observation for $\#\mathsf{NC}^1$.) We show that these closures hold for $\#\mathsf{sSC}^0$ as well. The following property will be useful.

Proposition 4. *For any f in $\#\mathsf{sSC}^0$ or $\#\mathsf{NC}^1$, and for any constant m, the value $f \bmod m$ is computable in FNC^1.*

Lemma 4. *$\#\mathsf{sSC}^0$ is closed under decrement and under division by a constant m.*

Another consequence of Proposition 4 can be seen as follows. We have three competing arithmetizations of the Boolean class NC^1. The most natural one is $\#\mathsf{NC}^1$, defined by arithmetic circuits. It contains $\#\mathsf{BWBP}$, which is contained in $\#\mathsf{sSC}^0$, though we do not know the relationship between $\#\mathsf{NC}^1$ and $\#\mathsf{sSC}^0$. Applying a "> 0?" test to any yields the same class, Boolean NC^1. We show here that applying a "$\equiv 0 \bmod p$?" test to any also yields the same language class, namely NC^1.

Definition 4. *For any function class $\#C$, let* Mod_pC *denote the class of languages L such that there is an $f \in \#C$ satisfying $\forall x \in \Sigma^* : \ x \in L \Longleftrightarrow f(x) \equiv 0 \mod p$.*

Theorem 6. *For any fixed p,* $\mathsf{Mod}_p\mathsf{BWBP} = \mathsf{Mod}_p\mathsf{sSC}^0 = \mathsf{Mod}_p\mathsf{NC}^1 = \mathsf{NC}^1$.

Proof. From Proposition 4, for $f \in \{\#\mathsf{sSC}^0, \#\mathsf{BWBP}, \#\mathsf{NC}^1\}$, and a constant m, the value $[f(x) \bmod m]$ can be computed in FNC^1. Hence the predicate $[f(x) \equiv 0 \bmod m]$ can be computed in NC^1. □

Another natural way to produce boolean circuits from arithmetic circuits is by allowing the circuit to perform test-for-zero operations. Such circuits, known as *Arithmetic-Boolean* circuits, were introduced by von zur Gathen, and have been studied in the literature; see e.g. [30, 29, 10, 3]. We extend this by looking at bounded width restrictions.

Definition 5. *Let $\mathcal{C}$ be any of the arithmetic circuit classes studied above. Then Arith-Bool $\mathcal{C}$ is defined to be the set of languages accepted by circuits from $\mathcal{C}$ with the following additional gates, and with Boolean output. (Here y is either a constant or a literal.)*

$$test(f) = \begin{cases} 0 & \text{if } f = 0 \\ 1 & \text{otherwise} \end{cases} \qquad select(f_0, f_1, y) = \begin{cases} f_0 & \text{if } y = 0 \\ f_1 & \text{if } y = 1 \end{cases}$$

Assigning $deg(select(f_0, f_1, y)) = 1 + \max\{deg(f_0), deg(f_1)\}$ and $deg(test(f)) = deg(f)$, we have the following,

Lemma 5. *1. Arith-Bool*$\#\mathsf{NC}^1 = \#\mathsf{NC}^1$.*[3]*
2. Arith-Bool$\#\mathsf{BWBP} = \#\mathsf{BWBP}$.
3. Arith-Bool$\#\mathsf{sSC}^0 = \#\mathsf{sSC}^0$

However, for the Gap classes, we do not have such a collapse. Analogous to the definitions of SPP and SPL, define a class SNC^1: it consists of those languages L for which the characteristic function χ_L is in GapNC^1. Then we have:

Lemma 6. *Arith-Bool*GapNC^1=GapNC^1 *if and only if* SNC^1=$\mathsf{C_=NC}^1$.

References

1. M. Agrawal, E. Allender, and S. Datta. On TC^0, AC^0, and arithmetic circuits. *Journal of Computer and System Sciences*, 60(2):395–421, 2000.
2. E. Allender. The division breakthroughs. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 74, 2001.
3. E. Allender. Arithmetic circuits and counting complexity classes. In J. Krajicek, editor, *Complexity of Computations and Proofs*, Quaderni di Matematica Vol. 13, pages 33–72. Seconda Universita di Napoli, 2004. An earlier version appeared in the Complexity Theory Column, SIGACT News 28, 4 (Dec. 1997) pp. 2-15.
4. E. Allender, J. Jiao, M. Mahajan, and V. Vinay. Non-commutative arithmetic circuits: depth reduction and size lower bounds. *Theoretical Computer Science*, 209:47–86, 1998.
5. R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *32nd International Colloquium on Automata, Languages, and Programming*, 2005.

6. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.
7. D. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *JCSS*, 38(1):150–164, 1989.
8. A. Borodin, S. Cook, P. Dymond, W. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal of Computation*, 18(3):559–578, 1989.
9. B. V. Braunmuhl and R. Verbeek. Input-driven languages are recognized in $\log$ n space. In *Proc. FCT Conference, LNCS*, pages 40–51, 1983.
10. S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM J. Comput.*, 21(4):755–780, 1992.
11. H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer. Nondeterministic NC^1 computation. *JCSS*, 57:200–212, 1998.
12. A. Chiu, G. Davida, and B. Litow. Division in logspace-uniform NC^1. *RAIRO Theoretical Informatics and Applications*, 35:259–276, 2001.
13. S. A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *STOC*, pages 338–345, 1979.
14. P. Dymond and S. Cook. Complexity theory of parallel time and hardware. *Information and computation*, 80:205–226, 1989.
15. P. W. Dymond. Input-driven languages are in $\log n$ depth. In *Information processing letters*, pages 26, 247–250, 1988.
16. H. Fernau, K.-J. Lange, and K. Reinhardt. Advocating ownership. In V. Chandru and V. Vinay, editors, *Proc. 16th FST&TCS, LNCS 1180*, pages 286–297. Springer, Dec. 1996.
17. S. Istrail and D. Zivkovic. Bounded width polynomial size Boolean formulas compute exactly those functions in AC^0. *Information Processing Letters*, 50:211–216, 1994.
18. D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 67–161. 1990.
19. K. Mehlhorn. Pebbling mountain ranges and its application to DCFL-recognition. In *Porc. 7th ICALP*, pages 422–432, 1980.
20. R. Niedermeier and P. Rossmanith. Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits. *Information and Computation*, 118(2):227–245, 1995.
21. N. Nisan. RL $\subseteq$ SC. *Computational Complexity*, 4(11):1–11, 1994.
22. W. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.
23. S.Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of Assoc. Comput. Mach.*, 18:4–18, 1971.
24. I. Sudborough. On the tape complexity of deterministic context-free language. *Journal of Association of Computing Machinery*, 25(3):405–414, 1978.
25. H. Venkateswaran. Properties that characterize LogCFL. *Journal of Computer and System Sciences*, 42:380–404, 1991.
26. V. Vinay. Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proceedings of 6th Structure in Complexity Theory Conference*, pages 270–284, 1991.
27. V. Vinay. Hierarchies of circuit classes that are closed under complement. In *CCC '96: Proceedings of the 11th Annual IEEE Conference on Computational Complexity*, pages 108–117, Washington, DC, USA, 1996. IEEE Computer Society.
28. H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York Inc., 1999.
29. J. von zur Gathen. Parallel linear algebra. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 573–617. Morgan Kaufmann, 1993.
30. J. von zur Gathen and G. Seroussi. Boolean circuits versus arithmetic circuits. *Information and Computation*, 91(1):142–154, 1991.