Complexity of Regular Functions

Eric Allender and Ian Mertz

Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
allender@cs.rutgers.edu, iwmertz@gmail.com

Abstract. We give complexity bounds for various classes of functions computed by cost register automata.

Keywords: computational complexity, transducers, weighted automata

1 Introduction

We study various classes of *regular functions*, as defined in a recent series of papers by Alur *et al.* [8, 10, 9]. In those papers, the reader can find pointers to work describing the utility of regular functions in various applications in the field of computer-aided verification. Additional motivation for studying these functions comes from their connection to classical topics in theoretical computer science; we describe these connections now.

The class of functions computed by *two-way* deterministic finite transducers is well-known and widely-studied. Engelfriet and Hoogeboom studied this class [18] and gave it the name of *regular string transformations*. They also provided an alternative characterization of the class in terms of monadic second-order logic. It is easy to see that this is a strictly larger class than the class computed by *one-way* deterministic finite transducers, and thus it was of interest when Alur and Černý [5] provided a characterization in terms of a new class of *one-way* deterministic finite automata, known as *streaming string transducers*; see also [6]. Streaming string transducers are traditional deterministic finite automata, augmented with a finite number of *registers* that can be updated at each time step, as well as an output function for each state. Each register has an initial value in Γ^* for some alphabet Γ, and at each step receives a new value consisting of the concatenation of certain other registers and strings. (There are certain other syntactic restrictions, which will be discussed later, in Section 2.)

The model that has been studied in [8, 10, 9], known as *cost register automata* (CRAs), is a generalization of streaming string transducers, where the register update functions are not constrained to be the concatenation of strings, but instead may operate over several other algebraic structures such as monoids, groups and semirings. Stated another way, streaming string transducers are cost register automata that operate over the monoid $(\Gamma^*, \circ)$ where $\circ$ denotes concatenation. Another important example is given by the so-called "tropical semiring", where the additive operation is min and the multiplicative operation is $+$; CRAs

over $(\mathbb{N} \cup \{\infty\}, \min, +)$ can be used to give an alternative characterization of the class of functions computed by weighted automata with costs in $\mathbb{N}$ [8].

The cost register automaton model is the main machine model that was advocated by Alur *et al.* [8] as a tool for defining and investigating various classes of "regular functions" over different domains. Their definition of "regular functions" does not always coincide exactly with the CRA model, but does coincide in several important cases. In this paper, we will focus on the functions computed by (various types of) CRAs.

Although there have been papers examining the complexity of several decision problems dealing with some of these classes of regular functions, there has not previously been a study of the complexity of computing the functions themselves.[1] There was even a suggestion [4] that these functions might be difficult or impossible to compute efficiently in parallel. Our main contribution is to show that most of the classes of regular functions that have received attention lie in certain low levels of the NC hierarchy.

2 Preliminaries

The reader should be familiar with some common complexity classes, such as L (deterministic logspace), and P (deterministic polynomial time). Many of the complexity classes we deal with are defined in terms of families of circuits. A language $A \subseteq \{0,1\}^*$ is accepted by circuit family $\{C_n : n \in \mathbb{N}\}$ if $x \in A$ iff $C_{|x|}(x) = 1$. Our focus in this paper will be on *uniform* circuit families; by imposing an appropriate uniformity restriction (meaning that there is an algorithm that describes C_n, given n) circuit families satisfying certain size and depth restrictions correspond to complexity classes defined by certain classes of Turing machines.

Logspace uniformity (meaning that, on input 1^n, a logspace-bounded machine can produce a description of the circuit C_n) is a convenient and easy-to-define uniformity condition that is adequate to use when characterizing complexity classes in terms of circuit families, as long as the complexity class contains logspace. However, most of the classes that we consider in this paper are *subclasses* of logspace – and for these small classes, the issue of choosing the proper uniformity condition is more complicated. These small classes can all be described equivalently in terms of machine models (using variants of alternating Turing machines) or in terms of logic (because a language can be viewed as a class of finite structures that satisfy a given logical formula, and it turns out that many important complexity classes correspond to certain natural classes of logic formulae). In order to maintain equivalence with these machine-based and logic-based characterizations, there is now a general consensus that U_E-uniformity is the best notion to use. Informally, a family of polynomial-size circuits $\{C_n\}$ is U_E-uniform if there is a linear-time machine that takes inputs of the form

[1] It may be appropriate to emphasize that, in the prior work on decision problems, a CRA is given as part of the input. In the present investigation, the CRA computing a function f is *fixed*, and the task is to compute $f(x)$, given x.

(n, g, h, p) and determines if p encodes a path from gate h to gate g in C_n, and also determines what type of gate g and h are. The reader will not need to be concerned with the details of the uniformity condition, and we refer the reader to Section 4.5 of the excellent text by Vollmer [26] for a longer discussion of uniformity, as well as for more information about the following standard circuit complexity classes.

- $\mathsf{NC}^i = \{A : A$ is accepted by a U_E-uniform family of circuits of bounded fan-in AND, OR and NOT gates, having size $n^{O(1)}$ and depth $O(\log^i n)\}$.
- $\mathsf{AC}^i = \{A : A$ is accepted by a U_E-uniform family of circuits of unbounded fan-in AND, OR and NOT gates, having size $n^{O(1)}$ and depth $O(\log^i n)\}$.
- $\mathsf{TC}^i = \{A : A$ is accepted by a U_E-uniform family of circuits of unbounded fan-in MAJORITY gates, having size $n^{O(1)}$ and depth $O(\log^i n)\}$.

We remark that, for constant-depth classes such as AC^0 and TC^0, U_E-uniformity coincides with U_D-uniformity, which is also frequently called DLOGTIME-uniformity.

Following the standard convention, we also use these same names to refer to the associated classes of *functions* computed by the corresponding classes of circuits. For instance, the function f is said to be in NC^1 if there is U_E-uniform family of circuits $\{C_n\}$ of bounded fan-in AND, OR and NOT gates, having size $n^{O(1)}$ and depth $O(\log n)$, where C_n has several output gates, and on input x of length n, C_n outputs an encoding of $f(x)$. (We say that an "encoding" of the output is produced, to allow the possibility that there are strings x and y of length n, such that $f(x)$ and $f(y)$ have different lengths.) It is easy to observe that, if the length of $f(x)$ is polynomial in $|x|$, then f is in (say) NC^1 if and only if the language $\{(x, i, b) :$ the i-th symbol of $f(x)$ is $b\}$ is in NC^1.

We will have need to refer to an alternative characterization of NC^1 in terms of branching programs. A *branching program* is a directed acyclic graph with vertices partitioned into levels. The *width* of a branching program is the largest number of vertices that appears in any level. There is a source node in level zero, and edges connect only vertices in adjacent levels (going from level i to $i+1$). In the highest-numbered level, there are two sink nodes, labeled 1 and 0, respectively. Each edge is labeled with a predicate of the form "the i-th bit of x is b" or "true". A branching program *accepts* x if there is a path from the source node to the sink labeled 1, traversing only edges whose predicates are true on input x. A branching program is *deterministic* if, for every node v and every input x, there is at most one one edge leaving v on input x whose label evaluates to true. It is known that every problem in NC^1 is accepted by a uniform family of width-5 *deterministic* branching programs of polynomial size, and conversely every language accepted by a uniform family of constant-width *nondeterministic* branching programs of polynomial size lies in NC^1 [11].

We also need to refer to certain classes defined by families of *arithmetic* circuits. Let $(S, +, \times)$ be a semiring. An *arithmetic circuit* consists of input gates, $+$ gates, and $\times$ gates connected by directed edges (or "wires"). One gate is designated as an "output" gate. If a circuit has n input gates, then it computes

a function from $S^n \to S$ in the obvious way. In this paper, we consider only arithmetic circuits where all gates have bounded fan-in.

- $\#\mathsf{NC}^1{}_S$ is the class of functions $f : \bigcup_n S^n \to S$ for which there is a U_E-uniform family of arithmetic circuits $\{C_n\}$ of logarithmic depth, such that C_n computes f on S^n.
- By convention, when there is no subscript, $\#\mathsf{NC}^1$ denotes $\#\mathsf{NC}^1{}_{\mathbb{N}}$, with the additional restriction that the functions in $\#\mathsf{NC}^1$ are considered to have domain $\bigcup_n \{0,1\}^n$. That is, we restrict the inputs to the Boolean domain. (Boolean negation is also allowed at the input gates.)
- GapNC^1 is defined as $\#\mathsf{NC}^1 - \#\mathsf{NC}^1$; that is: the class of all functions that can be expressed as the difference of two $\#\mathsf{NC}^1$ functions. It is the same as $\#\mathsf{NC}^1{}_{\mathbb{Z}}$ restricted to the Boolean domain. See [26, 1] for more on $\#\mathsf{NC}^1$ and GapNC^1.

The following inclusions are known:

$$\mathsf{NC}^0 \subseteq \mathsf{AC}^0 \subseteq \mathsf{TC}^0 \subseteq \mathsf{NC}^1 \subseteq \#\mathsf{NC}^1 \subseteq \mathsf{GapNC}^1 \subseteq \mathsf{L} \subseteq \mathsf{AC}^1 \subseteq \mathsf{P}.$$

All inclusions are straightforward, except for $\mathsf{GapNC}^1 \subseteq \mathsf{L}$ [19].

2.1 Cost-register automata

A *cost-register automaton* (CRA) is a deterministic finite automaton (with a read-once input tape) augmented with a fixed finite set of *registers* that store elements of some algebraic domain $\mathcal{A}$. At each step in its computation, the machine

- consumes the next input symbol (call it a),
- moves to a new state (based on a and the current state (call it q)),
- based on q and a, updates each register r_i using updates of the form $r_i \leftarrow f(r_1, r_2, \ldots, r_k)$, where f is an expression built using the registers $r_1, \ldots, r_k$ using the operations of the algebra $\mathcal{A}$.

There is also an "output" function μ defined on the set of states; μ is a partial function – it is possible for $\mu(q)$ to be undefined. Otherwise, if $\mu(q)$ is defined, then $\mu(q)$ is some expression of the form $f(r_1, r_2, \ldots, r_k)$, and the output of the CRA on input x is $\mu(q)$ if the computation ends with the machine in state q.

More formally, here is the definition as presented by Alur *et al.* [8].

A cost-register automaton M is a tuple $(\Sigma, Q, q_0, X, \delta, \rho, \mu)$, where

- Σ is a finite input alphabet.
- Q is a finite set of states.
- $q_0 \in Q$ is the initial state.
- X is a finite set of *registers*.
- $\delta : Q \times \Sigma \to Q$ is the state-transition function.
- $\rho : Q \times \Sigma \times X \to E$ is the register update function (where E is a set of algebraic expressions over the domain $\mathcal{A}$ and variable names for the registers in X).

– $\mu : Q \to E$ is a (partial) final cost function.

A *configuration* of a CRA is a pair (q, ν), where ν maps each element of X to an algebraic expression over $\mathcal{A}$. The *initial configuration* is (q_0, ν_0), where ν_0 assigns the value 0 to each register (or some other "default" element of the underlying algebra). Given a string $w = a_1 \dots a_n$, the *run* of M on w is the sequence of configurations $(q_0, \nu_0), \dots (q_n, \nu_n)$ such that, for each $i \in \{1, \dots, n\}$ $\delta(q_{i-1}, a_i) = q_i$ and, for each $x \in X$, $\nu_i(x)$ is the result of composing the expression $\rho(q_{i-1}, a_i, x)$ to the expressions in ν_{i-1} (by substituting in the expression $\nu_{i-1}(y)$ for each occurrence of the variable $y \in X$ in $\rho(q_{i-1}, a_i, x)$). The output of M on w is undefined if $\mu(q_n)$ is undefined. Otherwise, it is the result of evaluating the expression $\mu(q_n)$ (by substituting in the expression $\nu_n(y)$ for each occurrence of the variable $y \in X$ in $\mu(q_n)$).

It is frequently useful to restrict the algebraic expressions that are allowed to appear in the transition function $\rho : Q \times \Sigma \times X \to E$. One restriction that is important in previous work [8] is the "copyless" restriction.

A CRA is *copyless* if, for every register $r \in X$, for each $q \in Q$ and each $a \in \Sigma$, the variable "r" appears at most once in the multiset $\{\rho(q, a, s) : s \in X\}$. In other words, for a given transition, no register can be used more than once in computing the new values for the registers. Following [9], we refer to copyless CRAs as CCRAs. Over many algebras, unless the copyless restriction is imposed, CRAs compute functions that can not be computed in polynomial time. For instance, CRAs that can concatenate string-valued registers and CRAs that can multiply integer-valued registers can perform "repeated squaring" and thereby obtain results that require exponentially-many symbols to write down.

3 CRAs over Monoids

In this section, we study CRAs operating over algebras with a single operation. We focus on two canonical examples:

– CRAs operating over the commutative monoid $(\mathbb{Z}, +)$.
– CRAs operating over the noncommutative monoid $(\Gamma^*, \circ)$.

3.1 CRAs over the integers

Additive CRAs (ACRAs) are CRAs that operate over commutative monoids. They have been studied in [8, 10, 9]; in [10] the ACRAs that were studied operated over $(\mathbb{Z}, +)$, and thus far no other commutative monoid has received much attention, in connection with CRAs.

Theorem 1. *All functions computable by CCRAs over $(\mathbb{Z}, +)$ are computable in NC^1. (This bound is tight, since there are regular sets that are complete for NC^1 under projections [11].)*

Proof. It was shown in [8] that CCRAs (over any commutative semiring) have equivalent power to CRAs that are not restricted to be copyless, but that have another restriction: the register update functions are all of the form $r \leftarrow r' + c$ for some register r' and some semiring element c. Thus assume that the function f is computed by a CRA M of this form. Let M have k registers $r_1, \ldots, r_k$.

It is straightforward to see that the following functions are computable in NC^1:

- $(x, i) \mapsto q$, such that M is in state q after reading the prefix of x of length i. (To see that this is computable in NC^1, recall first that every regular set is in NC^1. Thus, each of the $O(1)$ languages of the form L_q is in NC^1, where L_q is the regular set accepted by the finite automaton M, altered so that q is its only accepting state. Now consider the circuit that, on input (x, i) computes the length-i prefix of x and determines membership of the prefix in each L_q.)
- $(x, i) \mapsto G_i$, where G_i is a labeled directed bipartite graph on $[k] \times [k]$, with the property that there is an edge labeled c from j on the left-hand side to ℓ on the right hand side, if the register update operation that takes place when M consumes the i-th input symbol includes the update $r_\ell \leftarrow r_j + c$. If the register update operation includes the update $r_\ell \leftarrow c$, then vertex ℓ on the right hand side is labeled c. (To see that this is computable in NC^1, note that by the previous item, in NC^1 we can determine the state q that M is in as it consumes the i-th input symbol. Thus G_i is merely a graphical representation of the register update function corresponding to state q.) Note that the indegree of each vertex in G_i is at most one. (The *outdegree* of a vertex may be as high as k.)

Now consider the graph G that is obtained by concatenating the graphs G_i (by identifying the right-hand side of G_i with the left-hand side of G_{i+1} for each i). This graph shows how the registers at time $i + 1$ depend on the registers at time i. G is a constant-width graph, and it is known that reachability in constant-width graphs is computable in NC^1 [11, 12]. Note that we can determine in NC^1 the register that provides the output when the last symbol of x is read. By tracing the edges back from that vertex in G (following the unique path leading back toward the left, using the fact that each vertex has indegree at most one) we eventually encounter a vertex of indegree zero. In NC^1 we can determine which edges take part in this path, and add the labels that occur along that path. This yields the value of $f(x)$. □

We remark that the NC^1 upper bound holds for any commutative monoid where iterated addition of monoid elements can be computed in NC^1.

A related bound holds, when the copyless restriction is dropped:

Theorem 2. *All functions computable by CRAs over $(\mathbb{Z}, +)$ are computable in GapNC^1. (This bound is tight, since there is one such function that is hard for GapNC^1 under AC^0 reductions.)*

Proof. We use a similar approach as in the proof of the preceding theorem. We build a bipartite graph G_i that represents the register update function that is executed while consuming the i-th input symbol, as follows. Each register update operation is of the form $r_\ell \leftarrow a_0 + r_{i_1} + r_{i_2} + \ldots r_{i_m}$. Each register r_j appears, say, a_j times in this sum, for some nonnegative integer a_j. If $r_\ell \leftarrow a_0 + \sum_{j=1}^{k} a_j \cdot r_j$ is the update for r_ℓ at time i, then if $a_j > 0$, then G_i will have an edge labeled a_j from j on the left-hand side to ℓ on the right-hand side, along with an edge from 0 to ℓ labeled a_0, and an edge from 0 to 0. Let the graph G_i correspond to matrix M_i. An easy inductive argument shows that $(\sum_{j=0}^{k}(\prod_{i=1}^{t} M_i))_{j,\ell}$ gives the value of register ℓ after time t. The expressions $\prod_{i=1}^{t} M_i))_{j,\ell}$ are instances of the "iterated matrix multiplication" problem, and it is known that iterated multiplication of $O(1) \times O(1)$ integer matrices can be computed in GapNC^1 [16]. Thus the sum of such expressions also lies in GapNC^1, which establishes the upper bound.

For the lower bound, observe that it is shown in [16], building on [13], that computing the iterated product of 3×3 matrices with entries from $\{0, 1, -1\}$ is complete for GapNC^1. More precisely, taking a sequence of such matrices as input and outputting the (1,1) entry of the product is complete for GapNC^1. Consider the alphabet Γ consisting of such matrices. There is a CRA taking input from Γ^* and producing as output the contents of the $(1, 1)$ entry of the product of the matrices given as input. (The CRA simulates matrix multiplication in the obvious way.) □

3.2 CRAs over $(\Gamma^*, \circ)$

Unless we impose the copyless restriction, CRAs over this monoid can generate exponentially-long strings. Thus in this subsection we consider only CCRAs.

CCRAs operating over the algebraic structure $(\Gamma^*, \circ)$ are precisely the so-called *streaming string transducers* that were studied in [6], and shown there to compute precisely the functions computed by two-way deterministic finite transducers (2DFAs). This class of functions is very familiar, and it is perhaps folklore that such functions can be computed in NC^1, but we have found no mention of this in the literature. Thus we present the proof here.

Theorem 3. *All functions computable by CCRAs over $(\Gamma^*, \circ)$ are computable in NC^1. (This bound is tight, since there are regular sets that are complete for NC^1 under projections [11].)*

Proof. Let M be a 2DFA computing a (partial) function f, and let x be a string of length n. If $f(x)$ is defined, then M halts on input x, which means that M visits no position i of x more than k times, where k is the size of the state set of M.

Define the *visit sequence at i* to be the sequence $q_{(i,1)}, q_{(i,2)}, \ldots q_{(i,\ell_i)}$ of length $\ell_i \leq k$ such that $q_{(i,j)}$ is the state that M is in the j-th time that it visits position i. Denote this sequence by V_i.

We will show that the function $(x, i) \mapsto V_i$ is computable in NC^1. Assume for the moment that this is computable in NC^1; we will show how to compute f in NC^1.

Note that there is a planar directed graph G of width at most k having vertex set $\bigcup_i V_i$, where all edges adjacent to vertices V_i go to vertices in either V_{i-1} or V_{i+1}, as follows: Given V_{i-1}, V_i and V_{i+1}, for any $q_{(i,j)} \in V_i$, it is trivial to compute the pair (i', j') such that, when M is in state $q_{(i,j)}$ scanning the i-th symbol of the input, then at the next step it will be in state $q_{(i',j')}$ scanning the i'-th symbol of the input. (Since this depends on only $O(1)$ bits, it is computable in U_E-uniform NC^0.) The edge set of G consists of these "next move" edges from $q_{(i,j)}$ to $q_{(i',j')}$. It is immediate that no edges cross when embedded in the plane in the obvious way (with the vertex sets $V_1, V_2, \ldots$ arranged in vertical columns with V_1 at the left end, and V_{i+1} immediately to the right of V_i, and with the vertices $q_{(i,1)}, q_{(i,2)}, \ldots q_{(i,\ell_i)}$ arranged in order within the column for V_i).

Let us say that (i, j) *comes before* (i', j') if there is a path from $q_{(i,j)}$ to $q_{(i',j')}$ in G. Since reachability in constant-width planar graphs is computable in AC^0 [12], it follows that the "comes before" predicate is computable in AC^0.

Thus there is a TC^0 circuit family that first uses AC^0 circuitry to compute, for each tuple (i, j, i', j'), whether (i', j') comes before (i, j), and furthermore, for each tuple, the transition function of M determines whether M produces an output symbol when departing state $q_{i',j'}$. Using MAJORITY gates, it is trivial to compute the number of 1's in a vector, and hence, in TC^0, one can compute the size of the set $\{(i', j') : (i', j')$ comes before (i, j) and M produces an output symbol when moving from $q_{(i',j')}\}$. Call this number $m_{(i,j)}$. Hence, in TC^0 one can compute the function $(x, m) \mapsto (i, j)$ such that $m_{(i,j)} = m$. But this allows us to determine what symbol is the m-th symbol of $f(x)$. Hence, given the sequences V_i, $f(x)$ can be computed in $\mathsf{TC}^0 \subseteq \mathsf{NC}^1$.

It remains to show how to compute the sequences V_i.

It suffices to show that the set $B = \{(x, i, V) : V = V_i\} \in \mathsf{NC}^1$. To do this, we will present a nondeterministic constant-width branching program recognizing B; such branching programs recognize only sets in NC^1 [11]. Our branching program will guess each V_j in turn; note that each V_j can be described using only $O(k \log k) = O(1)$ bits, and thus there are only $O(1)$ choices possible at any step. When guessing V_{j+1}, the branching program rejects if V_{j+1} is inconsistent with V_j and the symbols being scanned at positions j and $j + 1$. When $i = j$ the branching program rejects if V is not equal to the guessed value of V_i. When $j = |x|$ the branching program halts and accepts if all of the guesses $V_1, \ldots, V_n$ have been consistent. It is straightforward to see that the algorithm is correct.

□

4 CRAs over Semirings

In this section, we begin the study of CRAs operating over algebras with two operations satisfying the semiring axioms. We focus on three such structures:

– CRAs operating over the commutative ring $(\mathbb{Z}, +, \times)$ (Section 4.1).

- CRAs operating over the commutative semiring $(\mathbb{Z} \cup \{\infty\}, \min, +)$: the so-called "tropical" semiring (Section 5).
- CRAs operating over the noncommutative semiring $(\Gamma^* \cup \{\bot\}, \max, \circ)$ (Section 6).

(Here, the max operation takes two strings x, y in Γ^* as input, and produces as output the lexicographically-larger of the two. For any $x, \max(x, \bot) = x$.) There is a large literature dealing with *weighted automata* operating over semirings. It is shown in [8] that the class of functions computed by weighted automata operating over a semiring $(S, +, \times)$ is exactly equal to the class of functions computed by CRAs operating over $(S, +, \times)$, where the only register operations involving $\times$ are of the form $r \leftarrow \sum_i r_i \times c_i$ for some registers r_i and some semiring elements c_i. Thus for each structure, we will also consider CRAs satisfying this restriction.

We should mention the close connection between iterated matrix product and weighted automata operating over commutative semirings. As in the proof of Theorem 2, when a CRA is processing the i-th input symbol, each register update function is of the form $r_\ell \leftarrow a_0 + \sum_{j=1}^{k} a_j \cdot r_j$, and thus the register updates for position i can be encoded as a matrix. Thus the computation of the machine on an input x can be encoded as an instance of iterated matrix multiplication. In fact, some treatments of weighted automata essentially *define* weighted automata in terms of iterated matrix product. (For instance, see [22, Section 3].) Thus, since iterated product of $k \times k$ matrices lies in $\#\mathsf{NC}^1{}_S$ for any commutative semiring S, the functions computed by weighted automata operating over S all lie in $\#\mathsf{NC}^1{}_S$. (For the case when $S = \mathbb{Z}$, iterated matrix product of $k \times k$ matrices is *complete* for GapNC^1 for all $k \geq 3$ [16, 13].)

4.1 CRAs over the integers.

First, we consider the copyless case:

Theorem 4. *All functions computable by CCRAs over $(\mathbb{Z}, +, \times)$ are computable in GapNC^1. (Some such functions are hard for NC^1, but we do not know if any are hard for GapNC^1.)*

Proof. Consider a CCRA M computing a function f, operating on input x. There is a function computable in NC^1 that maps x to an encoding of an arithmetic circuit that computes $f(x)$, constructed as follows: The circuit will have gates $r_{j,i}$ computing the value of register j at time i. The register update functions dictate which operations will be employed, in order to compute the value of $r_{j,i}$ from the gates $r_{j',i-1}$. Due to the copyless restriction, the outdegree of each gate is at most 1 (which guarantees that the circuit is a formula).

It follows from Lemma 5 below that $f \in \mathsf{GapNC}^1$. □

Lemma 5. *If there is a function computable in NC^1 that takes an input x and produces an encoding of an arithmetic formula that computes $f(x)$ when evaluated over the integers, then $f \in \mathsf{GapNC}^1$.*

Proof. Recall that an arithmetic formula takes a sequence of integers as input, and produces an integer as output. Thus a formula (together with an input sequence) can be viewed as a representation of the number that is produced as output. By [15], there is a logarithmic-depth arithmetic-Boolean formula over the integers, that takes as input a bitstring encoding a formula F (along with inputs to F) and outputs the integer represented by F. An arithmetic-Boolean formula is a formula with Boolean gates AND, OR and NOT, and arithmetic gates $+, \times$, as well as *test*[2] and *select* gates that provide an interface between the two types of gates. Actually, the construction given in [15] does not utilize any *test* gates [14], and thus we need not concern ourselves with them. (Note that this implies that there is no path in the circuit from an arithmetic gate to a Boolean gate.)

A *select* gate takes three inputs (y, x_0, x_1) and outputs x_0 if $y = 0$ and outputs x_1 otherwise. In the construction given in [15], *select* gates are only used when y is a Boolean value. When operating over the integers, then, $\mathit{select}(y, x_0, x_1)$ is equivalent to $y \times x_1 + (1 - y) \times x_0$. But since Boolean NC^1 is contained in $\#\mathsf{NC}^1 \subseteq \mathsf{GapNC}^1$ (see, e.g., [1]), the Boolean circuitry can all be replaced by arithmetic circuitry. (When operating over algebras other than $\mathbb{Z}$, it is not clear that such a replacement is possible.) □

We cannot entirely remove the copyless restriction while remaining in the realm of polynomial-time computation, since repeated squaring allows one to obtain numbers that require exponentially-many bits to represent in binary. However, as noted above, if the multiplicative register updates are all of the form $r \leftarrow r' \times c$, then again the GapNC^1 upper bound holds (and in this case, some of these CRA functions are complete for GapNC^1, just as was argued in the proof of Theorem 2).

5 CRAs over the tropical semiring.

In this section, we consider CRAs operating over the tropical semiring. We show that the functions computable by such CRAs have complexity bounded by the complexity of functions in $\#\mathsf{NC}^1$, and thus lie in L. In order to state a more precise bound on the complexity of these functions, we introduce the class $\#\mathsf{NC}^1_{\mathrm{trop}}$, and we prove some basic propositions about arithmetic circuits over the tropical semiring.

5.1 Arithmetic Circuit Preliminaries

Functions in $\#\mathsf{NC}^1_{\mathrm{trop}}$ have complexity in some sense intermediate between NC^1 and $\#\mathsf{NC}^1$. Proposition 6 shows that there are some functions in $\#\mathsf{NC}^1_{\mathrm{trop}}$ that are hard for NC^1, and Lemma 9 shows that, if the values at the input level of

[2] A *test* gate computes a unary function, taking as input a value from an arithmetic gate g, and producing as output the Boolean value of the predicate "g is equal to zero".

$\#\mathsf{NC}^1_{\text{trop}}$ circuits have binary representation of only $O(\log n)$ bits, then $\#\mathsf{NC}^1_{\text{trop}}$ circuits are no harder to evaluate than $\#\mathsf{NC}^1$ functions. (Without this restriction, the best known upper bound is AC^1; see, e.g. [2, Lemma 5.5].) It is worth remarking that it has been conjectured that $\#\mathsf{NC}^1$ consists of precisely the functions computable in NC^1; see [1]. Thus the lower and upper bounds of NC^1 and $\#\mathsf{NC}^1$ are not very far apart.

Recall that the accepted convention for $\#\mathsf{NC}^1$ is that inputs are restricted to be in $\{0,1\}$, and that for every Boolean input x_i the negated input $\neg x_i$ is also available. In order to simplify the statement of the following results, we allow $\#\mathsf{NC}^1_{\text{trop}}$ circuits to take arbitrary elements from $\mathbb{Z} \cup \{\infty\}$ as input (as in the standard setting for arithmetic circuit complexity). But sometimes it is also convenient to consider $\#\mathsf{NC}^1_{\text{trop}}$ as a class of *languages*, in which case we will follow the same convention as for $\#\mathsf{NC}^1$, and restrict the inputs to be in $\{0,1\}$, where for every Boolean input x_i the negated input $\neg x_i$ is also available.

Recall from Section 4 the result of [8], which states that the class of functions computed by weighted automata over the tropical semiring corresponds to the class of functions computed by CRAs operating over the tropical semiring (without the copyless restriction) where the only register operations involving addition are of the form $r \leftarrow r' + c$ for some register r' and some natural number c. As discussed in Section 4, these functions all lie in $\#\mathsf{NC}^1_{\text{trop}}$. In Theorem 10, we show nearly as good an upper bound for the class of functions computed by *copyless* CRAs *without this restriction* on addition. First, however, we present some basic results on the complexity of functions in $\#\mathsf{NC}^1_{\text{trop}}$, beginning with a lower bound.

Proposition 6. $\mathsf{NC}^1 \subseteq \#\mathsf{NC}^1_{\text{trop}}$.

Proof. Recall first that the inclusion $\mathsf{NC}^1 \subseteq \#\mathsf{NC}^1$ is proved by observing that NC^1 circuits can be assumed without loss of generality to be "unambiguous", in the sense that each OR gate that evaluates to one always has *exactly one* child that evaluates to one. (That is, $a \vee b$ is replaced by $(\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$; see, e.g., [1].) Thus consider any language $L \in \mathsf{NC}^1$, and consider the "unambiguous" NC^1 circuit family $\{C_n\}$ accepting L. If we simply replace each AND gate by min, and we replace each OR gate by $+$, then the resulting $\#\mathsf{NC}^1_{\text{trop}}$ circuit is equivalent to C_n. □

Now, we consider the problem of evaluating $\#\mathsf{NC}^1_{\text{trop}}$ circuits. We note first that determining if the output is ∞ can be accomplished in NC^1.

Proposition 7. *The problem of taking as input an arithmetic formula ϕ (with assignments to all of the input variables), and determining if ϕ evaluates to ∞ is in NC^1.*

Proof. Given ϕ, replace each finite input with 0, and replace each ∞ input with 1. Change each min gate to AND, and change each $+$ gate to OR. Call the resulting formula ϕ'; it is easy to see that ϕ' evaluates to 1 iff ϕ evaluates to ∞. Now, by [15], ϕ' can be evaluated in NC^1. □

Thus, if we want to evaluate a $\#\mathsf{NC}^1_{\text{trop}}$ formula, it suffices to focus on the case where the formula evaluates to a value other than ∞. A very powerful result by Elberfeld, Jakoby, and Tantau [17, Theorem 5] can be used to show that some closely-related problems reduce to the computation of $\#\mathsf{NC}^1$ functions, but we find that there are enough complications caused by the presence of ∞-inputs and negative inputs, so that it is simpler to present a direct argument rather than to invoke [17]. Thus our next lemma says that evaluating a $\#\mathsf{NC}^1_{\text{trop}}$ formula that takes on a finite value is no harder than evaluating a $\#\mathsf{NC}^1$ expression. The following definition and lemma make precise what is meant by "no harder than" in this context.

Definition 8. *Let x be a non-zero dyadic rational. That is, x can be expressed as $x = \sum_{i=-m}^{m} b_i 2^i$ for some m, where $b_i \in \{0,1\}$ for all i. Define* $\mathsf{low.order}(x)$ *to be the least $i \in \{-m, \ldots, m\}$ such that $b_i = 1$. If ϕ is an arithmetic formula, then* $\mathsf{low.order}(\phi)$ *is defined to be* $\mathsf{low.order}(z)$ *for the number z that is represented by ϕ.*

Observe that $\mathsf{low.order}(xy) = \mathsf{low.order}(x) + \mathsf{low.order}(y)$. Observe also that $\mathsf{low.order}(x+y) = \min\{\mathsf{low.order}(x), \mathsf{low.order}(y)\}$ if $\mathsf{low.order}(x) \neq \mathsf{low.order}(y)$, but if $\mathsf{low.order}(x) = \mathsf{low.order}(y)$, then it is not obvious how to obtain a useful bound on $\mathsf{low.order}(x+y)$. For this reason, in the following lemma, we will introduce the notion of "spread".

An informal interpretation of the following lemma is: In order to evaluate a $\#\mathsf{NC}^1_{\text{trop}}$ formula ϕ, one can evaluate a $\#\mathsf{NC}^1$ formula ϕ' and perform some trivial arithmetic on $\mathsf{low.order}(\phi')$. Thus $\#\mathsf{NC}^1_{\text{trop}}$ efficiently reduces to $\#\mathsf{NC}^1$.

Lemma 9. *Let c and ℓ be natural numbers. There is a function f computable in NC^1 that takes as input a $\#\mathsf{NC}^1_{\text{trop}}$ formula ϕ of depth $c \log n$, where each finite input to ϕ is in the range $[-n^\ell, n^\ell]$, and produces as output a $\#\mathsf{NC}^1$ formula ϕ' and numbers m, r such that, if ϕ evaluates to a finite value z, then $zr \leq \mathsf{low.order}(\phi') - m < (z+1)r$. (In other words, $z = \lfloor(\mathsf{low.order}(\phi') - m)/r\rfloor$.)*

Proof. The argument we present is very similar to a proof that is presented in [21] (where they are working over the $(\max, +)$ algebra, instead of $(\min, +)$).

Let the $\#\mathsf{NC}^1_{\text{trop}}$ formula ϕ be given, of depth $c \log n$, where each input that is not ∞ lies in the range $[-n^\ell, n^\ell]$. We first build an arithmetic formula ϕ_0 over the dyadic rationals, and then modify ϕ_0 to obtain the desired $\#\mathsf{NC}^1$ formula ϕ'.

We assume without loss of generality that ϕ is a complete binary tree, where all paths from input gates to the output have length $c \log n$, and we also assume that ϕ is composed of alternating layers of $+$ and min gates. (This normal form can be obtained by at most doubling the depth, by inserting dummy gates, using the rules $\min(x, x) = x$ and $x + 0 = x$; the modified formula can be obtained from ϕ in NC^1.) Thus ϕ has n^c input gates, each of which takes on a value in $[-n^\ell, n^\ell] \cup \{\infty\}$.

Let $r = (n^\ell + 1)n^{2c} + 1$. The formula ϕ_0 is obtained from ϕ by changing each $+$ gate of ϕ to a $\times$ gate, and changing each min gate of ϕ to a $+$ gate. At the

input level, each input of ϕ that has some finite value a is replaced by the value 2^{ra}. (Note, it is possible that $a < 0$.) Each input of ϕ that is labeled with ∞ is replaced by the value $2^{(n^\ell+1)n^c r}$.

First, we observe that each gate g of ϕ_0 evaluates to a dyadic rational in the range $[2^{-rn^\ell n^c}, 2^{r(n^\ell+1)n^{2c}}]$. This is because all inputs to ϕ_0 are positive. The output cannot be larger than the result of multiplying together n^c values of size $2^{(n^\ell+1)n^c r}$ (which is the value that replaces ∞), and it cannot be smaller than multiplying together n^c values of size 2^{-rn^ℓ}.

Before we proceed to our inductive argument showing that the output of ϕ_0 encodes the value of ϕ, it is necessary to prove some results showing how the values stored in the gates of ϕ_0 evolve as the computation progresses. Given a gate g_0 of ϕ_0 whose value is encoded in binary as $\sum_{i=-m}^{m} b_i 2^i$, define $\mathsf{spread}(g_0)$ to be the largest $j < r$ such that $b_{\lfloor \mathsf{low.order}(g_0)/r \rfloor r + j} = 1$. Here is some intuition about $\mathsf{spread}(g_0)$. Think of the binary representation of the value of g_0 as a bit string divided into subfields of length r. All of the fields to the right of $\mathsf{low.order}(g_0)$ are all zero. The field corresponding to positions

$$\lfloor \mathsf{low.order}(g_0)/r \rfloor r + (r-1), \ldots \lfloor \mathsf{low.order}(g_0)/r \rfloor r + 1, \lfloor \mathsf{low.order}(g_0)/r \rfloor r$$

is where the useful information is stored. If g_0 is an input gate, then this field is very "clean"; it is of the form $0^{r-1}1$. If g_0 appears at a higher depth in the circuit, this field can be a bit messy. However, the high-order bits of this field are all going to be 0, and the 1's can only appear in positions $\lfloor \mathsf{low.order}(g_0)/r \rfloor r + j$ for $0 \leq j \leq \mathsf{spread}(g_0)$.

Claim. A. If g_0 is a gate at depth d of ϕ_0, then $\mathsf{spread}(g_0) \leq 2^d$.

Note that, since $d = c \log n$, $2^d < r$.

Proof. The proof of the claim is by induction on d. When $d = 0$, $\mathsf{spread}(g_0) = 0 < 2^d$.

If g_0 is a + gate at depth d, say $g_0 = h_0 + k_0$, where the claim holds at h_0 and k_0, then either

$$\mathsf{spread}(g_0) = \max\{\mathsf{spread}(h_0), \mathsf{spread}(k_0)\},$$

or

$$\mathsf{spread}(g_0) = \max\{\mathsf{spread}(h_0), \mathsf{spread}(k_0)\} + 1.$$

In either case, by the induction hypothesis we have $\mathsf{spread}(g_0) \leq 2^{d-1} + 1 \leq 2^d$. So in either case the claim holds at g_0.

If g_0 is a $\times$ gate at depth d, say $g_0 = h_0 \times k_0$, where the claim holds at h_0 and k_0, then $\mathsf{spread}(g_0) = \mathsf{spread}(h_0) + \mathsf{spread}(k_0)$. (To see this, consider the binary representation of the product $h_0 \times k_0$ as divided up into fields of length r, and similarly divide h_0 and k_0 into fields of length r. Let x_h and x_k be the contents of the fields containing $\mathsf{low.order}(h_0)$ and $\mathsf{low.order}(k_0)$, respectively. Then the field containing $\mathsf{low.order}(h_0 \times k_0)$ consists of the low-order r bits of the product

$x_h \times x_k$. The length of the non-zero part of the product $x_h \times x_k$ is exactly $\mathsf{spread}(h_0) + \mathsf{spread}(k_0)$.)

By induction, $\mathsf{spread}(g_0) = \mathsf{spread}(h_0) + \mathsf{spread}(k_0) \leq 2^{d-1} + 2^{d-1} = 2^d$. □

Next, we claim that the value of ϕ can easily be extracted from the value of ϕ_0.

Claim. B. If ϕ evaluates to a finite value z, then $zr \leq \mathsf{low.order}(\phi_0) < z(r+1)$. Thus $z = \lfloor \mathsf{low.order}(\phi_0)/r \rfloor$ (since $z < r$).

Proof. This claim follows immediately from the following statement, which we prove by induction on d:

For all d, if gate g at depth d takes on a finite value z in ϕ, then $zr \leq \mathsf{low.order}(g_0) < zr + 2^d$ (where g_0 is the value that the gate corresponding to g takes on in ϕ_0), and if g (at depth d) takes on the value ∞ in ϕ, then $\mathsf{low.order}(g_0) \geq (n^\ell + 1)n^c r - dn^\ell$.

This suffices to prove the claim, since the output gate has depth $d = c \log n$ and thus $2^d = n^c < r$. The claim holds at the input level (where $d = 0$).

Now let g be a $+$ gate at depth d computing $h+k$, where the inductive hypothesis holds at h and k. If g takes on a finite value z, then both h and k take on finite values, call them z_h and z_k. By induction, we have $z = z_h + z_k$, and $g_0 = h_0 \times k_0$, where $z_h r \leq \mathsf{low.order}(h_0) < z_h r + 2^{d-1}$ and $z_k r \leq \mathsf{low.order}(k_0) < z_k r + 2^{d-1}$. Observe that $\mathsf{low.order}(g_0) = \mathsf{low.order}(h_0 \times k_0) = \mathsf{low.order}(h_0) + \mathsf{low.order}(k_0)$. Thus $zr = z_h r + z_k r \leq \mathsf{low.order}(h_0) + \mathsf{low.order}(k_0) = \mathsf{low.order}(g_0) < z_h r + 2^{d-1} + z_k r + 2^{d-1} = (z_h + z_k)r + 2^d = zr + 2^d$.

If g takes on the value ∞, then either h or k also takes on the value ∞. Assume without loss of generality that $h = \infty$. Then, by induction $\mathsf{low.order}(h_0) \geq (n^\ell + 1)n^c r - (d-1)n^\ell$. Thus $\mathsf{low.order}(g_0) = \mathsf{low.order}(h_0) + \mathsf{low.order}(k_0) \geq ((n^\ell + 1)n^c r - (d-1)n^\ell) + (-n^\ell) = (n^\ell + 1)n^c r - dn^\ell$.

Next let g be a min gate at depth d, computing $\min(h, k)$, where the inductive hypothesis holds at h and k. If g takes on a finite value z, then at least one of h and k takes on a finite value. Assume without loss of generality that h is the minimum, and that h takes the value z_h, and let z_k be the value of gate k. By induction, we have $z = z_h$, and $g_0 = h_0 \times k_0$, where $z_h r \leq \mathsf{low.order}(h_0) < z_h r + 2^{d-1}$. If z_k is finite, then $z_k r \leq \mathsf{low.order}(k_0) < z_k r + 2^{d-1}$, and otherwise $\mathsf{low.order}(k_0) \geq (n^\ell + 1)n^c r - (d-1)n^\ell$.

If $\mathsf{low.order}(h_0) \neq \mathsf{low.order}(k_0)$ (which is the case, in particular, if $k = \infty$), then $\mathsf{low.order}(g_0) = \mathsf{low.order}(h_0)$, and the inductive hypothesis holds at g_0. Thus assume that $\mathsf{low.order}(h_0) = \mathsf{low.order}(k_0)$. Thus $zr = z_h r \leq \mathsf{low.order}(h_0) \leq \mathsf{low.order}(g_0)$, and thus the first inequality of the claim holds at g_0.

Also $\mathsf{low.order}(g_0) \leq \lfloor \mathsf{low.order}(h_0)/r \rfloor r + \mathsf{spread}(h_0) + 1 \leq \lfloor \mathsf{low.order}(h_0)/r \rfloor r + 2^{d-1} + 1$ by Claim A. By induction, we have $\mathsf{low.order}(g_0) \leq \lfloor (z_h r + 2^{d-1})/r \rfloor r + 2^{d-1} + 1 = z_h r + 2^{d-1} + 1 = zr + 2^{d-1} + 1 < zr + 2^d$, as desired.

If g takes on the value ∞, then both h and k also evaluate to ∞. By the inductive hypothesis, $\mathsf{low.order}(h_0) \geq (n^\ell + 1)n^c r - (d-1)n^\ell$ and $\mathsf{low.order}(k_0) \geq (n^\ell + 1)n^c r - (d-1)n^\ell$. It follows that $\mathsf{low.order}(g_0) \geq \min\{\mathsf{low.order}(h_0), \mathsf{low.order}(k_0)\} \geq (n^\ell + 1)n^c r - (d-1)n^\ell > (n^\ell + 1)n^c r - dn^\ell$. This completes the proof of the

inductive step, and establishes how the value of ϕ can be obtained from the value of ϕ_0. □

However, ϕ_0 operates over the dyadic rationals, and it still remains for us to produce a formula ϕ' over $\mathbb{N}$.

Let q be the least natural number, such that no input to ϕ_0 has a label less than 2^{-qr}. Let ϕ' be ϕ_0, where each input x of ϕ_0 is replaced by $2^{qr}x$. Clearly, ϕ' operates over $\mathbb{N}$. Since ϕ was assumed to have alternating levels of $+$ and min gates, ϕ' has alternating levels of $\times$ and $+$ gates. At the input level, the value of each gate of ϕ_0 can be obtained by dividing the value of the corresponding gate of ϕ' by 2^{qr}. More generally, if g_0 is a gate of ϕ_0 such that paths from the input level to g_0 encounter d $\times$ gates, then the value of g_0 can be obtained by dividing the value of the corresponding gate of ϕ' by $2^{2^d qr}$.

The proof is completed, by setting m equal to $2^d qr$, where d is $\frac{c}{2}\log n$. □

5.2 Tropical CRAs

Having established the facts that we need about $\#\mathsf{NC}^1_{\text{trop}}$, we return to the task of giving a bound on the complexity of CRAs operating over the tropical semiring.

Again, we first consider the copyless case.

Theorem 10. *All functions computable by CCRAs over the tropical semiring are computable in* $\#\mathsf{NC}^1_{\text{trop}} \circ \mathsf{NC}^1$*, and are computable in* L.

Here, $\#\mathsf{NC}^1_{\text{trop}} \circ \mathsf{NC}^1$ refers to the class of functions expressible as $g(f(x))$ for some functions $f \in \mathsf{NC}^1$ and $g \in \#\mathsf{NC}^1_{\text{trop}}$. Thus this can be viewed as $\#\mathsf{NC}^1_{\text{trop}}$ with some minor NC^1 pre-processing. The reader might wonder how $\#\mathsf{NC}^1_{\text{trop}} \circ \mathsf{NC}^1$ differs from $\#\mathsf{NC}^1_{\text{trop}}$, in light of Proposition 6 (which states that NC^1 circuits can be replaced by $\#\mathsf{NC}^1_{\text{trop}}$ circuitry). The explanation is that our NC^1 function outputs encodings of ∞ (along with elements of $\mathbb{N}$), to feed into the $\#\mathsf{NC}^1_{\text{trop}}$ circuit.

Proof. The L upper bound follows easily, because the only operation that increases the value of a register is a $+$ operation, and because of the copyless restriction the value of a register after i computation steps can be expressed as a sum of $i^{O(1)}$ values that are present as constants in the program of the CRA. Thus, in particular, the value of a register at any point during the computation on input x can be represented using $O(\log |x|)$ bits. Thus a logspace machine can simply simulate a CRA directly, storing the value of each of the $O(1)$ registers, and computing the updates at each step.

Another way of obtaining the L upper bound follows from Lemma 9, because, when we establish the $\#\mathsf{NC}^1_{\text{trop}} \circ \mathsf{NC}^1$ upper bound, we use $\#\mathsf{NC}^1_{\text{trop}}$ circuits where all of the finite input values are small. Thus, not only are these functions computable in L, but they can easily be computed from functions in $\#\mathsf{NC}^1$.

For the $\#\mathsf{NC}^1_{\text{trop}} \circ \mathsf{NC}^1$ upper bound, first note that there is a function h computable in NC^1 that takes x as input, and outputs a description of an arithmetic

formula F over the tropical semiring that computes $f(x)$. This is exactly as in the first paragraph of the proof of Theorem 4.

Next, as in the proof of Lemma 5, recall that, by [15], there is a uniform family of *logarithmic-depth* arithmetic-Boolean formulae $\{C_n\}$ over the tropical semiring, that takes as input an encoding of a formula F and outputs the integer represented by F. Furthermore, each arithmetic-Boolean formula C_n has Boolean gates AND, OR and NOT, and arithmetic gates $\min, +$, as well as *select* gates, and there is no path in C_n from an arithmetic gate to a Boolean gate.

Let $\{D_n\}$ be the uniform family of arithmetic circuits, such that D_n is the connected subcircuit of C_n consisting only of arithmetic min and + gates. We now have the following situation: The NC^1 function h (which maps x to an encoding of a formula F having some length m) composed with the circuit C_m (which takes F as input and produces $f(x)$ as output) is identical with some NC^1 function h' (computed by the NC^1 circuitry in the composed hardware for $C_m(h(x))$) feeding into the arithmetic circuitry of D_m. Each *select* gate with inputs (y, x_0, x_1) can be simulated by the subcircuit $\min(x_0 + z(y), x_1 + z(\neg y))$ where $z(v)$ is the NC^1 function that takes the Boolean value v as input, and outputs 0 if $v = 0$, and outputs ∞ otherwise. This is precisely what is needed, in order to establish our claim that $f \in \#\mathsf{NC}^1_{\text{trop}} \circ \mathsf{NC}^1$. □

Unlike the case of CRAs operating over the integers, CRAs over the tropical semiring without the copyless restriction compute only functions that are computable in polynomial time (via a straightforward simulation). We know of no better upper bound than P in this case, and we also have no lower bounds.

As noted above at the beginning of Section 4, if the "multiplicative" register updates (i.e., + in the tropical semiring) are all of the form $r \leftarrow r' + c$, then even without the copyless restriction, the computation of a CRA function f reduces to iterated matrix multiplication of $O(1) \times O(1)$ matrices over the tropical semiring. Again, it follows easily that the contents of any register at any point in the computation can be represented using $O(\log n)$ bits. Thus the upper bound of L holds also in this case.

6 CRAs over the max-concat semiring.

As in Section 3.2, we consider only CCRAs.

Theorem 11. *All functions computable by CCRAs over* $(\Gamma^*, \max, \circ)$ *are computable in* AC^1.

Proof. Let f be computed by a CCRA M operating over $(\Gamma^*, \max, \circ)$.

We first present a logspace-computable function h with the property that $h(1^n)$ is a description of a circuit C_n computing f on inputs of length n. The input convention is slightly different for this circuit family. For each input symbol a and each $i \leq n$ there is an input gate $g_{i,a}$ that evaluates to λ (the empty string) if $x_i = a$, and evaluates to $\perp$ otherwise. (This provides an "arithmetical" answer to the Boolean query "is the i-th input symbol equal to a?")

Assume that there are gates $r_{1,i}, r_{2,i}, \ldots, r_{k,i}$ storing the values of each of the registers at time i. For $i = 0$ these gates are constants. For each input symbol a and each $j \leq k$, let $E_{a,j}(r_{1,i}, \ldots, r_{k,i})$ be the expression that describes how register j is updated if the $i+1$-st symbol is a. Then the value $r_{j,i+1} = \max_a\{g_{i,a} \circ E_{a,j}(r_{1,i}, \ldots, r_{k,i})\}$. This yields a very uniform circuit family, since the circuit for inputs of length n consists of n identical blocks of this form connected in series. That is, there is a function computable in NC^1 that takes 1^n as input, and produces an encoding of circuit C_n as output.

Although the depth of circuit C_n is linear in n, its *algebraic degree* is only polynomial in n. (Recall that the additive operation of the semiring is max and the multiplicative operation is $\circ$. Thus the degree of a max gate is the maximum of the degrees of the gates that feed into it, and the degree of a $\circ$ gate is the sum of the degrees of the gates that feed into it.) This degree bound follows from the copyless restriction. (Actually, the copyless restriction is required only for the $\circ$ gates; inputs to the max gates could be re-used without adversely affecting the degree.)

By [2, Proposition 5.2], arithmetic circuits of polynomial size and algebraic degree over $(\Gamma^*, \max, \circ)$ characterize exactly the complexity class $\mathsf{OptLogCFL}$. $\mathsf{OptLogCFL}$ was defined by Vinay [25] as follows: g is in $\mathsf{OptLogCFL}$ if there is a nondeterministic logspace-bounded auxiliary pushdown automaton M running in polynomial time, such that, on input x, $g(x)$ is the lexicographically largest string that appears on the output tape of M along any accepting computation path. The proof of Proposition 5.2 in [2], which shows how an auxiliary pushdown automaton can simulate the computation of a max-concat circuit, also makes it clear that an auxiliary pushdown machine, operating in polynomial time, can take a string x as input, use its logarithmic workspace to compute the bits of $h(1^{|x|})$ (i.e., to compute the description of the circuit $C_{|x|}$), and then to produce $C_{|x|}(x) = f(x)$ as the lexicographically-largest string that appears on its output tape along any accepting computation path. That is, we have $f \in \mathsf{OptLogCFL}$.

By [2, Lemma 5.5], $\mathsf{OptLogCFL} \subseteq \mathsf{AC}^1$, which completes the proof. □

7 Conclusion

Figure 1 summarizes our main results, and calls attention to some of our main open problems:

- Are there any CCRA functions over $(\mathbb{Z}, +, \times)$ that are complete for GapNC^1?
- Are there any CCRA functions over the tropical semiring that are hard for $\#\mathsf{NC}^1_{\text{trop}}$? (Note in particular that it is not known that iterated product of constant-dimension matrices is complete for $\#\mathsf{NC}^1_{\text{trop}}$.)
- The gap between the upper and lower bounds for CCRA functions over $(\Gamma^*, \max, \circ)$ is quite large (NC^1 versus $\mathsf{OptLogCFL} \subseteq \mathsf{AC}^1$). Can this be improved?
- Is there an NC upper bound for CRA functions (without the copyless restriction) over the tropical semiring?

Subsequent to our work, there have been some new developments relating to CRAs. Complexity results extending our work have been presented, that apply to automata similar to CRAs augmented with a "visible pushdown" [23]. And a logical characterization has been given for a subclass of CRA functions [24].

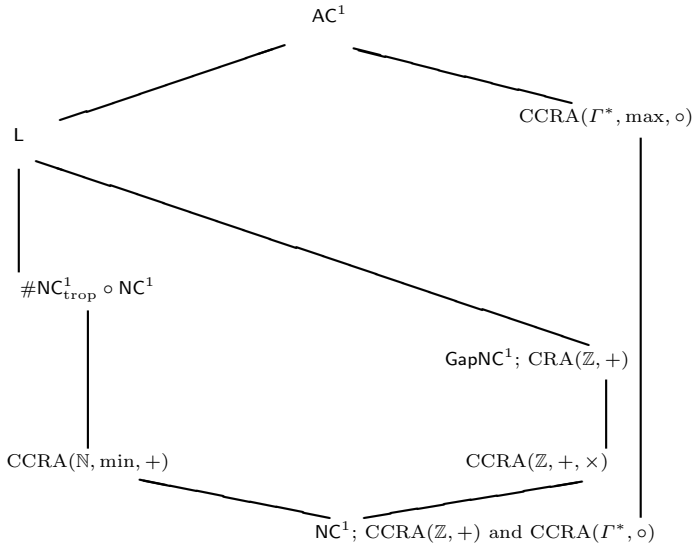

Fig. 1. Summary of Results. When a class of CRA functions and a complexity class appear together, it means that containment of the CRA class in the complexity class is tight, since some of the CRA functions are complete for the complexity class.

Acknowledgments

This work was supported by NSF grants CCF-1064785 and CCF-1555409 and an REU supplement. We thank Samir Datta for calling our attention to [21] and for his comments on an earlier version of this work [3]. We also thank Till Tantau and the anonymous referees for helpful comments.

References

1. Allender, E.: Arithmetic circuits and counting complexity classes. In: Krajíček, J. (ed.) Complexity of Computations and Proofs, Quaderni di Matematica, vol. 13, pp. 33–72. Seconda Università di Napoli (2004)
2. Allender, E., Jiao, J., Mahajan, M., Vinay, V.: Non-commutative arithmetic circuits: Depth reduction and size lower bounds. Theoretical Computer Science 209(1-2), 47–86 (1998)

3. Allender, E., Mertz, I.: Complexity of regular functions. In: Proc. 9th International Conference on Language and Automata Theory and Applications (LATA). pp. 449–460. No. 8977 in Lecture Notes in Computer Science, Springer (2015)
4. Alur, R.: Regular functions (2013), lecture presented at *Horizons in TCS: A Celebration of Mihalis Yannakakis's 60th Birthday*, Center for Computational Intractability, Princeton, NJ
5. Alur, R., Cerný, P.: Expressiveness of streaming string transducers. In: Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS). LIPIcs, vol. 8, pp. 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
6. Alur, R., Cerný, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 599–610 (2011)
7. Alur, R., D'Antoni, L., Deshmukh, J.V., Raghothaman, M., Yuan, Y.: Regular functions, cost register automata, and generalized min-cost problems. CoRR abs/1111.0670 (2011)
8. Alur, R., D'Antoni, L., Deshmukh, J.V., Raghothaman, M., Yuan, Y.: Regular functions and cost register automata. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 13–22 (2013), see also the expanded version, [7].
9. Alur, R., Freilich, A., Raghothaman, M.: Regular combinators for string transformations. In: Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science, (CSL-LICS). p. 9. ACM (2014)
10. Alur, R., Raghothaman, M.: Decision problems for additive regular functions. In: ICALP. pp. 37–48. No. 7966 in Lecture Notes in Computer Science, Springer (2013)
11. Barrington, D.A.: Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. Journal of Computer and System Sciences 38, 150–164 (1989)
12. Barrington, D.A.M., Lu, C.J., Miltersen, P.B., Skyum, S.: Searching constant width mazes captures the AC^0 hierarchy. In: 15th International Symposium on Theoretical Aspects of Computer Science (STACS). pp. 73–83. No. 1373 in Lecture Notes in Computer Science, Springer (1998)
13. Ben-Or, M., Cleve, R.: Computing algebraic formulas using a constant number of registers. SIAM Journal on Computing 21(1), 54–58 (1992)
14. Buss, S.: Comment on formula evaluation (2014), personal communication.
15. Buss, S.R., Cook, S., Gupta, A., Ramachandran, V.: An optimal parallel algorithm for formula evaluation. SIAM Journal on Computing 21(4), 755–780 (1992)
16. Caussinus, H., McKenzie, P., Thérien, D., Vollmer, H.: Nondeterministic NC^1 computation. Journal of Computer and System Sciences 57(2), 200–212 (1998)
17. Elberfeld, M., Jakoby, A., Tantau, T.: Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In: STACS'12 (29th Symposium on Theoretical Aspects of Computer Science). vol. 14, pp. 66–77. LIPIcs (2012)
18. Engelfriet, J., Hoogeboom, H.J.: MSO definable string transductions and two-way finite-state transducers. ACM Trans. Comput. Log. 2(2), 216–254 (2001)
19. Hesse, W., Allender, E., Barrington, D.A.M.: Uniform constant-depth threshold circuits for division and iterated multiplication. Journal of Computer and System Sciences 65, 695–716 (2002)
20. Jakoby, A., Tantau, T.: Computing shortest paths in series-parallel graphs in logarithmic space. In: Complexity of Boolean Functions, 12.03. - 17.03.2006. Dagstuhl

Seminar Proceedings, vol. 06111. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
21. Jakoby, A., Tantau, T.: Logspace algorithms for computing shortest and longest paths in series-parallel graphs. In: Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS). pp. 216–227. No. 4855 in Lecture Notes in Computer Science, Springer (2007), the proof of Lemma 4 can be found as the proof of Lemma 3.5 in [20].
22. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: On the complexity of equivalence and minimisation for Q-weighted automata. Logical Methods in Computer Science 9(1) (2013)
23. Krebs, A., Limaye, N., Ludwig, M.: Cost register automata for nested words. In: Proc. 22nd International Computing and Combinatorics Conference - (COCOON). Lecture Notes in Computer Science, Springer (2016), to appear
24. Mazowiecki, F., Riveros, C.: Maximal partition logic: towards a logical characterization of copyless cost register automata. In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 41. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
25. Vinay, V.: Counting auxiliary pushdown automata. In: Proceedings of the Sixth Annual Structure in Complexity Theory Conference, Chicago, Illinois, USA, June 30 - July 3, 1991. pp. 270–284 (1991), http://dx.doi.org/10.1109/SCT.1991.160269
26. Vollmer, H.: Introduction to Circuit Complexity: A Uniform Approach. Springer-Verlag New York Inc. (1999)