Complexity Theory Through a Spectrum of Computation Models

by

Ian Mertz

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Mathematics

at Rutgers University

May 2016

Author .....
Department of Computer Science
April 26, 2016

Certified by .....
Eric Allender
Distinguished Professor
Thesis Supervisor

Complexity Theory Through a Spectrum of Computation Models

by

Ian Mertz

Submitted to the Department of Computer Science
on April 26, 2016, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Mathematics

Abstract

The study of complexity theory is the study of whether or not a given problem can be solved by a restricted model of computation. The Turing machine model is often used to define the best-known natural complexity classes such as P and NP. However, there are other computational models besides Turing machines that are also worthy of study. In addition to giving an introduction to the overall study of complexity theory and its value in the field of computer science, we look at the basic space-bounded Turing machine model before jumping into two different models of computation, circuits and automata. For Turing machines we attempt to decide the complexity of a particular reachability problem that falls somewhere between L and NL. For circuit complexity we give more extensive characterizations of various classes of polynomial-size logarithmic-depth circuits. For automata we give various upper bounds on a type of automaton defined by Alur *et al.* called cost-register automata.

Thesis Supervisor: Eric Allender
Title: Distinguished Professor

Acknowledgments

I have had three incredible mentors I would be remiss not to thank. In high school it was Graciela Elia, who started me on computer science and connected me with various programs and topics, as well as remaining vigilant against my terribly lax coding and proof practices. Then it was Rajiv Gandhi, who motivated my choice to go into theory by pushing me beyond my comfort zone, both in my personal work and in the collaborative research setting. Finally in college it was Eric Allender, who made the crazy move of giving a research opportunity to a freshman who barely knew what complexity theory was. He gave me his time, energy, problems, and funding to devote three years and two summers to learning and researching. And of course, he oversaw the writing of this thesis, without which it may have never gotten done.

I want to thank my coauthor Anna Gál for showing up and telling us that our work in Chapter 3 could go much further than we had taken it. Thanks to Michael Saks, Swastik Kopparty, Shubhangi Saraf, and Bill Steiger for the fantastic courses that made my coursework challenging and engaging. And thanks to my classmates for helping me survive, particularly Amey Bhangale, Mrinal Kumar, and Abhishek Bhrushundi.

Thanks to all the conference friends and associates I made, which helped keep me sane where the coffee didn't. Thanks to all my undergraduate math and computer science friends, who kept me motivated to continue research by being at least as obsessed with scientific progress as I was. On the other side of the coin thanks to all my other undergraduate friends for giving me a break from those same people. I want to especially thank my roommate Stephen Hackler for four years of tolerance and distractions.

My most emphatic and heartfelt thanks belong to my parents, who have supported me and remained endlessly positive, throughout both the past four years and all eighteen before that. You gave me the opportunity to explore my passion for mathematics my whole life, and ultimately enabled me to meet with all the peers and mentors who made this possible. And of course, you were my best peers and mentors yourselves. This thesis is dedicated to you.

The author acknowledges the support of the Aresty Research Center for the work done on Chapter 2. Much of the material in Chapters 3 and 4 appears in [32, 8], for which the author also acknowledges the support of NSF grants CCF-0832787 and CCF-1064785, as well as an REU supplement.

Contents

Chapter 1

Introduction: what is complexity theory?

As technology grows more and more powerful, solving larger and more difficult problems faster than ever before, it is sometimes easy to lose sight of how a computer actually performs these functions under the hood. With the help of electrical engineers worldwide, hardware continues to develop every year and cut down the time it takes to run its computations. However, state-of-the-art hardware running at optimal capacity is useless without equally good software, built with straightforward sequential commands and tasked with solving millions of problems in the blink of an eye.

What is essential to remember about computers is that they operate on a set of simple instructions that perform the most basic functions given knowledge only of the physical electrical impulses feeding in, and not of the abstract problem at hand. It is up to electrical engineers to produce hardware to send and store bits, and it is up to programmers to use those bits as efficiently as possible. And while there are many methods to solve a problem, programmers must be efficient with using and moving bits around in devising their method of choice.

As an example of how crucial it is to choose the right method to solve a problem, consider the task of searching a phone book for a given name. A simple strategy is to start at the first page and work our way through the book sequentially, name by name and page by page, until we find the name. This works well for looking up Scott Aaronson's phone number, but go looking for Leslie Valient and this could take hours. We know the book is ordered alphabetically, but we let this crucial piece of information fall by the wayside.

Now consider the following approach. Open the book right in the center and look at the first name on the page. If that name matches our target, then we are in luck, but more often than not we will get a completely different name. But we know the book is ordered alphabetically, and we instantly know whether our target name appears in the book before this page or afterwards; if we open up to Richard Lipton for example, we know that Scott Aaronson comes earlier, while Leslie Valiant comes later. Now we can throw away the other half of the book, and in one simple step we have cut the problem in two! Open the relevant half of the book in the middle and repeat the procedure. You will find your target name in minutes.

While a computer can run billions of instructions per second, there are examples like the phone book where the wrong strategy can take seconds, minutes, or even hours, days, or years to solve, where a more clever solution can consistently answer queries in milliseconds. Thus we have the notion of an *algorithm*, a precise step-by-step procedure for carrying out a particular task or solving a particular problem using only the basic instructions available. In our example, it was "open book", "flip page", "check name", and "compare words alphabetically".

Ultimately the design and implementation of good algorithms is the most essential step in building computers that are as fast and efficient as the modern world demands. Programmers and software engineers are tasked with both creating newer better algorithms and coding them on the physical machine in the most efficient

ways, balancing their knowledge of software and hardware. But are there limits to this constant refinement? Increasing hardware efficiency may increase the number of instructions a computer can perform per second, but is there a software barrier that allows algorithms to perform only so well, no matter how clever or creative we may be? Here arises the notion of *complexity theory*: given a problem, how efficient of an algorithm could we possibly design to solve it?

1.1 Complexity theory

1.1.1 Measuring efficiency: the Turing machine model

The fundamental question when approaching a problem is how to measure "efficiency". Because we are attempting to solve these problems on a computer, it is natural to think in terms of time, possibly measuring the number of cycles the computer has to run for before the solution can be found, or in terms of space, the amount of RAM and/or memory needed to be set aside for the procedure. Even before computers were in circulation, in a seminal work Alan Turing [54] introduced a theoretical model on which algorithms could be implemented and analyzed.

Definition 1 *A Turing machine is a conceptual machine that solves a given problem L, and consists of three parts:*

- *three tapes that are each $1 \times \infty$ rectangles broken up into sequential 1×1 squares called cells, each of which holds a single bit of either 0 or 1. We label the tapes as follows:*

  1. *the input tape, where the input to L is written (read-only)*
  2. *the output tape, where we eventually write the solution (write-once)*
  3. *the work tape, where we perform the algorithm (read-write)*

- *a head for each tape, each pointing to a single cell*

- *a state machine, where at any current state there is a function that takes as input the value of the cells pointed to by the heads of the input and work tapes, and outputs a new value for the cells pointed to on the work and output tapes, a direction to move each of the heads (left or right), and the next state*

This model, while bizarre at first sight, actually contains all the keys we need to start talking about efficient algorithms. Time can be measured by the number of steps the head takes from the time it receives the input to the time it finishes writing the answer on the tape, while space can be measured by the number of cells needed in the work tape to run the algorithm. If we squint a bit and imagine the three infinite tapes as keyboard, memory, and screen, and imagine the head and state machine as the hardware, then it really starts to look like a computer. And in fact this crude abstract model is enough to analyze the way in which our modern computers operate. Though the model was proposed decades before the first "computer" would ever be built, time and space complexity of algorithms on Turing machines is a strong approximation of how those same algorithms will run on the latest hardware.

1.1.2 Formalizing time complexity

Now that we have a model of computation, we can begin to talk about complexity theory in earnest. Given an individual problem, we can try and come up with better algorithms to prove that it can be solved in a certain amount of time, which we define relative to the length of our input. To visualize why working relative to the input length is necessary, imagine our phone book example with a preschool directory instead of the YellowPages. Just because our sloppier first method of searching runs fast on the former does not imply it will also run fast on the latter. Hence it is useful to talk not about the absolute time required to solve a problem, but rather how much

time it will take given an input of some length, which we classically denote as n.

Given a problem, we come up with an algorithm that solves it in time $f(n)$, where f is some function. Our first algorithm searched through the phone book one entry at a time in order, which could take as little time as one step and as much time as the whole length of the book. Since we generally want to worry about the worst-case scenario, we say that this algorithm runs in time $f(n) = n$, or *linear time*, aka it takes time exactly proportional to the input length.

Our second algorithm by contrast was cutting the length of the input in half at every step. This algorithm can be seen to to run in $f(n) = \log(n)$ time, where $\log(n)$ is the binary logarithm function. To see the difference between $f(n) = n$ and $f(n) = \log(n)$, let us imagine that the length of our phone book is a million names long, and thus $n = 1000000$. In our first algorithm we will check at worst 1000000 names (no surprises there), whereas in the absolute worst case our second algorithm will check 20, an astonishing gap. Even more telling is what happens when n doubles in size: in the first algorithm we check 2000000 names instead of 1000000, but in the second we check 21 names instead of 20. Our time commitment barely changes.

So the problem of finding a name in a phone book is said to take at most $\log(n)$ time. This is an example of a problem that can be solved without even looking at the entire input. However, this only holds when we are allowed to access any specific part of the memory in one step, which would not work in the Turing machine model given that the head can only move right or left by one step. If we stay true to the Turing machine model, most useful computations require at least linear time.

Beyond linear time there is time n^c for a constant c, which we call *polynomial time*. If instead of an ordered phone book we are given a disorganized jumble of names, we cannot hope to beat n time. However, if we wanted to order the list alphabetically, there are simple algorithms to do it in time n^2, and only slightly more sophisticated ones that can do it in time $n \times \log(n)$.

Another good example is the problem of adding two binary numbers that are roughly the same size. The classic elementary school procedure is to take the least significant bits, add them together, and then move to next column, bringing with us the carry if there is one. If we repeat this up until the end of the numbers, we get the full answer. Note that our input length is n, and so each number is about $n/2$ bits long, so we take c steps per column we end up with $f(n) = \frac{c}{2}n$, which is linear time.

Note that n changes with different inputs, while c remains fixed. In practice the constant often depends less on the inherent structure of the problem and more on what instructions the computer does and does not allow, which is not true of functions of the input length. Also, if we had some $f(n) = n + \log(n)$, note that because $\log(n)$ is a very small additive quantity compared to n, and so $f(n) < 2n$ which is again linear. However, despite $\log(n)$ being much smaller than n we hold onto it when they are multiplied together, as it indicates a dependence on n, an example being the $n\log(n)$ time it takes to sort a list, which is between n and n^2.

Combining all these notions, when discussing time complexity we reduce $f(n)$ to its largest factor, stripped of any constant factors or smaller additive quantities. We define this notion formally before moving forward.

Definition 2 *We say that $f(n) = O(g(n))$ if there exist natural numbers c, n_0 such that for all $n > n_0$, $f(n) \leq cg(n)$.*

1.1.3 Complexity classes

Note that we have introduced two examples of problems solvable by algorithms running in linear time, as well as one that runs in time $n\log(n)$. It also makes sense to talk about the set of *all* problems that we can solve in linear time, as there are an infinite number of such examples. Likewise there is a set of problems that can be solved in $n\log(n)$ time, n^2 time, n^3 time, and any other polynomial f we can think of. For each such polynomial, we can assign that set a name: $\mathsf{DTIME}(f(n))$. These

are our first examples of *complexity classes*, which are simply sets of problems that are all solvable by algorithms that run under the same constraints (in this case the restraint is that the algorithm must finish in time $f(n)$).

One would be hard-pressed to walk into a computer science setting and not here buzz of the famous P vs NP problem, but we now have the tools to give a formal definition.

Definition 3 P *is the set of all problems that have an algorithm to solve them in polynomial time. In other words, if* $f(n) = O(n^c)$ *for some constant* c*, then the problem belongs in* P*.*

All of our examples thus far have been problems belonging to P. P is often denoted as $\cup_{c\in\mathbb{N}}\mathsf{DTIME}(O(n^c))$, or even as $\mathsf{DTIME}(n^{O(1)})$. Note that although $n\log(n)$ does not seem to fit this definition, it turns out that $n\log(n) = O(n^2)$ just by setting $c = 1$ and $n_0 = 1$, and so the sorting problem also belongs to P.

NP problems are a little trickier. The formal definition of NP involves *non-determinism*, which we refrain from defining formally but think of as "guessing". A problem is in NP iff it has an algorithm that runs on a Turing machine in polynomial time where the state machine is allowed to make guesses as to how to proceed, and in addition we assume that it makes the best possible guess whenever possible. Another equivalent definition is given as follows: a problem is in NP iff the problem of checking whether or not a proposed solution to the problem is correct is in P.

It is straightforward to see that $\mathsf{P} \subseteq \mathsf{NP}$, given that an NP algorithm is a P algorithm that is allowed but not required to guess during its execution. So every problem in P is also in NP. If it also turned out to be the case that $\mathsf{NP} \subseteq \mathsf{P}$, then because they would contain the exact same set of problems, they would be equal. However, it is widely believed that this is not the case; $\mathsf{P} = \mathsf{NP}$ would have startling ramifications in many areas of not just computer science, but society. For example the current standard scheme for data encryption is in NP, but if $\mathsf{P} = \mathsf{NP}$ then it could

be solved in polynomial time, which would imply that no electronic security system is safe.

From the P and NP example we see how complexity classes can be defined by placing restrictions on Turing machines and seeing what problems are solvable, and so we define a few more. L is the class of problems with an algorithm on a Turing machine to solve them that only uses $\log(n)$ cells on the work tape. We motivate this as a setting worth analyzing as follows: given an input of length n, $\log(n)$ cells are enough to write down the number n in binary, or any number between 1 and n. Therefore an algorithm in L can hold numbers that refer to a specific bit in the input, but not much more.

In an analog to P and NP, we define NL to be problems with an algorithm that is allowed only $\log(n)$ cells on the work tape, but once again can "guess" using non-determinism. Later we will see examples of both L and NL problems, but for now it can be understood that this is a distinction similar to P and NP revolving around space constraints rather than time constraints. Once again it holds trivially that $\mathsf{L} \subseteq \mathsf{NL}$. However it is not entirely clear that $\mathsf{L} \neq \mathsf{NL}$, and many sane people (myself included) seriously entertain the notion that they are equal, which while not as earth-shattering as $\mathsf{P} = \mathsf{NP}$ would be, would be a tremendous result.

1.1.4 Reducibility

For the sake of the next section we now define two more problems that will be very useful in the rest of the thesis, both of which rely on the notion of *graphs*.

Definition 4 *A graph G is a set of vertices V with edges E, where an edge $e = (u, v)$ for some vertices u and v. If there is an edge $e = (u, v)$, we say that v is adjacent to u. A path P is a set of edges $e_1 = (s, v_1), e_2 = (v_1, v_2), \ldots, e_k = (v_{k-1}, t)$ in the graph, and if there exists a path between vertices s and t we say they are connected.*

Note that edges are *ordered* pairs of vertices, and so it is possible that (u, v) is an edge in the graph while (v, u) is not. This is called a *directed graph*, referring to the direction of the edges. When the order does not matter, i.e. for every $e = (u, v)$ there is an $e' = (v, u)$ (which we will also label as e), the graph is called an *undirected graph*. These two models seem very similar, but in reality they are very different algorithmically. For example, note that in a directed graph it may be that u is connected to v but not the other way around, while in an undirected graph connectivity is symmetric.

It is a fundamental problem in all areas of computer science to test whether two vertices are connected. It has been known for a long time that st-connectivity is in NL [42], as given s and t we can start at s and then repeatedly guess the next edge in a path to t. Because we guess optimally we can assume that if s and t are connected, our algorithm will find a path connecting them. Additionally we need only store a pointer to the current vertex in our path, which takes logarithmic space as mentioned above.

However, a surprising result of Reingold showed that when the graph is undirected, st-connectivity is actually in L [46]. The algorithm involves transforming the input graph into a new graph where all vertices connected in the original graph are connected by a path of length at most $\log(n)$, and additionally where all vertices have only a small number of other vertices adjacent to them. Thus given s and t on the new graph we simply check every single possible path of length $\log(n)$ or less. Because there are only a polynomial number of such paths, in logspace we can keep a counter of how many paths we have already checked and thus check all of them in some canonical order. This transformation was shown to be possible in logspace on undirected graphs.

Now we will pretend that there exists an algorithm A that can solve st-connectivity on directed graphs. However, we are given an instance of st-connectivity on an undi-

rected graph G instead, and want to use A to solve our problem. After thinking for a moment, we realize that we can take G and produce a directed graph G', where the vertices of G and G' are the same and every edge $e = (u, v)$ in G is two separate directed edges, $e' = (u, v)$ and $e\text{“} = (v, u)$, in G'. Hence if we run A on G', it will tell us whether or not there is a path from s to t in G', which also tells us whether or not there is a path in G, and so we are done.

This is an example of a *reduction* from undirected reachability to directed reachability. We have transformed the input in some small way and fed it into a Turing machine that decides a completely different problem for us, and yet we get the correct answer as a result. In some sense, this shows that st-connectivity on directed graphs is at least as hard as the problem on undirected graphs, as solving the former also gives us the latter for free. We write this as "undirected st-connectivity $\leq_m$ directed st-connectivity".

Now we see the power and elegance of complexity classes. It turns out that for every problem $C \in \mathsf{NL}$, there is a reduction from C to directed st-connectivity, or in other words $C \leq_m$ directed st-connectivity. Intuitively directed st-connectivity is "harder" than every other problem C in NL, and so we say that directed st-connectivity is *hard* for NL, and because it is also contained in NL, it is NL-*complete* as well.

Likewise, it is the case that undirected st-connectivity is L-complete. The beauty of this framework speaks for itself; these two complexity classes, defined with regard to restrictions on Turing machines, can also be defined as the sets of problems reducible to two natural st-connectivity problems. Thus even if one were to make the claim that defining classes based on space and log functions is contrived, it turns out that the classes L and NL arise in a fundamental way regardless.

1.2 Other models of computation

1.2.1 Connections between time and space

The study of Turing machines and complexity classes gives us a very natural way to characterize problems according to their efficiency, and with it we can take important computational problems and try and fit them into the framework of *complexity theory*, either by: a) coming up with an algorithm that solves them under time/space restrictions; b) reducing them to problems for which efficient solutions are known (*upper bounds*); or c) by reducing problems for which no efficient solutions are known to them (*lower bounds*).

But how do we relate classes based on different restricted models, such as time-bounded Turing machines with no space restrictions and space-bounded Turing machines with no time restrictions? It turns out that complexity theory is incredibly flexible, and it is possible to achieve relations between very different complexity classes. We already showed that $\mathsf{L} \subseteq \mathsf{NL}$ and $\mathsf{P} \subseteq \mathsf{NP}$, but how do the time bounded classes relate to the space-bounded classes? It turns out that we can connect these two relations:

Theorem 1 $\mathsf{NL} \subseteq \mathsf{P}$

Proof: Since every problem in NL efficiently reduces to directed st-connectivity, we need only find a polynomial time algorithm for st-connectivity to prove the theorem. An example of such an algorithm is given here:

1. Define $C, D = \{s\}$, $v = s$ (recall that we are trying to find a path from s to t)
2. If there is an edge (v, t) then the answer is "true"
3. Otherwise, for all $u \notin D$ such that $(v, u) \in E$, add u to C and D
4. If C empty, then the answer is "false"

5. Otherwise set $v = u$ for some $u \in C$ and remove u from C

6. Return to step 2

Note that our general procedure is to keep a list of all vertices that s has a path to (D), and then expand that list by checking vertices adjacent to those we have already discovered but not yet checked (C). □

1.2.2 Beyond Turing machines

Just as we have compared different Turing machine restrictions, it turns out we can also analyze models of computation other than Turing machines. We will focus on two particular models in this thesis, namely *Boolean circuits* and *automata*, which we define in brief now.

Definition 5 *A Boolean circuit is a directed graph where each vertex v is labeled with a symbol σ_e that is either c, x_k, or f, where $c \in \{0, 1\}$, x_k is the kth bit of the input (here we take the input to be boolean), and f is a function that takes in boolean values and outputs either 0 or 1. We also have the following restrictions:*

- *if $\sigma_v = c$ or x_k for any k, then there are no edges $e = (u, v)$*
- *if $\sigma_v = f$ for some function f that takes i inputs, there are exactly i unique edges $e = (u, v)$*
- *there exist no vertices v such that v has a path back to v*
- *there is a special vertex o called the "output gate" with the property that there are no edges $e = (o, v)$*

Typically the functions f are the basic boolean functions AND ($\wedge$), OR ($\vee$), and NOT ($\neg$), which, respectively, return 1 if and only if every input bit is 1, return 0 if and

only if every input bit is 0, and return the opposite of the single input bit. We refer to vertices labeled with functions as gates, and edges are referred to as wires.

Because there are no vertices with paths back to themselves, and o has no edges going out while vertices labeled with constants or input symbols have no vertices going in, we can draw the graph by putting the vertices in rows such that o is alone in the top row, every vertex v which o is adjacent to is in a row below o, every vertex v' that v is adjacent to is on a row below v, and so on until we reach the bottom, where there are only vertices labeled c or x_k. If we then evaluate all fs starting from the lowest level and working our way up and so on until the function at o is evaluated, which gives us either 0 or 1, and we return "true" iff the function at o evaluates to 1.

Definition 6 *An automaton is a directed graph with the following properties:*

- *each edge e is labeled with a symbol σ_e from a finite alphabet Σ such that there do not exist edges $e_1 = (u, v_1), e_2 = (u, v_2)$ such that $\sigma_{e_1} = \sigma_{e_2}$*
- *for every $u \in V, \sigma \in \Sigma$, there exists an edge $e = (u, v)$ such that $\sigma_e = \sigma$*
- *there is a set of vertices A in the graph that are called "accept states", as well as a special vertex s called the "start state"*
- *there is a pointer that points to a specific vertex v in the graph, initially pointing to $v = s$*

On a given input $x = x_1x_2x_3 \ldots x_n$, we consume the first symbol of the input x_1 and move the pointer from s to the vertex v such that $e = (s, v)$ is an edge in the graph with $\sigma_e = x_1$. We repeat this procedure by consuming x_2 and moving along the appropriate edge on v, and continue likewise until the entire input has been used up, at which point we return "true" iff the current vertex we are pointing to is in A.

Intuitively, an automaton is a state machine where we read each symbol of the input sequentially and transition to another state according to the symbol, returning "true" if and only if we end in an accept state.

While automata represent a simplistic decision process where we do not rely on saving any values for later, only considering the current input symbol and all the ones we have seen up until now, circuits mirror the process inside a computer where bits are sequentially put through logic gates until there is only a single bit to indicate whether or not the circuit returns "true" or "false". Of course there are natural ways to model these settings on a Turing machine–indeed the decision process by which a Turing machine chooses what to do at a given time step is an automaton, and circuits can be evaluated in polynomial time one piece at a time as described above–they do represent different restrictions on the Turing machine model, not in terms of time or space but in terms of how the computation is carried out.

Depending on how these models are restricted, we get very natural complexity classes that fit into the framework of complexity theory. One such important connection again relies on the notion of a reduction, though we omit the proof.

Definition 7 *The Circuit Evaluation Problem is defined as follows: given a circuit C and an input string x, return the value of C evaluated on x as described above. This problem is in* P *as noted in the above paragraph.*

Theorem 2 *Every problem in* P *reduces to the Circuit Evaluation Problem.*

Once again we see a natural problem being used as an alternate characterization for an equally natural complexity class. It is clear from this statement that evaluating circuits is inherently a Turing machine problem, but from the theorem we also see that every problem in the class of Turing machine problems P can be rewritten as a circuit. Hopefully this provides some motivation as to why we deem it useful to study these alternate models of computation.

1.3 Our results

For the rest of the thesis we will dive into various problems across these three models of computation. For Turing machines we turn our attention to *gridgraph reachability*, a highly structured instance of directed st-connectivity that we hoped to put in L. For circuits we add a host of new characterizations to well-known Boolean circuit complexity classes in terms of so-called *arithmetic circuits*, as well as proving stronger results and even degree reductions on polynomials as a consequence. For automata we analyze a model introduced by Alur *et al.* in 2010 called *cost-register automata*, which run like normal automata but additionally evaluate some algebraic function along the way. We reduce the problem over various settings to Turing-machine complexity classes as well as circuit complexity classes.

With this I hope to survey a number of different computational models in complexity theory, to hopefully motivate the notion that these models are interconnected and worthy of study. I find it incredible that finding algorithms in one setting can also affect unrelated problems via reductions and the interconnected structure of complexity classes, and this has shaped my approach to research for these past four years.

For a more in-depth introduction to the field of complexity theory, we refer the reader to an excellent textbook by Sipser [48].

Chapter 2

Gridgraph reachability

2.1 Introduction

We begin our study into complexity theory with the Turing machine model. In the gap between L and NL there are many other classes, one of which is UL or "unambiguous logspace", the class for which there exist NL algorithms that have only one solution. Following up on the work done by Allender *et. al* [33] and Soltys [50] we delve into a problem known as Shuffle. The Shuffle problem is the following: given three words a, b, c, we want to find out if there is a way of interleaving the letters of a and b *without changing their original order* to get c. For example, if $a = 01$ and $b = 10$, then Shuffle will accept if $c = 0110$ or $c = 1010$, but not if $c = 0011$.

The Shuffle problem has been discussed in standard texts such as [41], and thanks to Soltys we know that it can be decided in NL, by using a simple algorithm that at time t chooses whether to take the next letter from a or from b, thus only having to maintain pointers to the first letter of each we have not yet taken. However, this uses the guessing power of non-determinism, because if a and b both match up with the next letter of c at time t then trying to work deterministically can either make us screw up or require us to use more than logarithmic space.

We can put Shuffle into the slightly more restricted class UL using a reduction [33] from Shuffle to a particular directed reachability problem called layered gridgraph. A *gridgraph* is a graph where the vertices are the points on an $m \times k$ grid, with edges only possible between vertices that are adjacent on the grid in the four cardinal directions (aka no diagonals). We define the vertex in the top left corner of the grid to be s and the vertex in the bottom right to be t. The *layered* restriction states that every edge must point either to the right or downwards, or in other words, no edge can point away from the s-to-t direction. Because planar reachability is in UL [28], it holds that layered gridgraph–a special case of planar graphs–is also in UL.

Given an instance of layered gridgraph, it is natural to try to transform the graph into a state where we can use the considerable power of Reingold's connectivity algorithm. One naive approach is to simply replace the directed edges in our instance with undirected edges, which fails in fairly trivial cases. Another approach would be to try and remove paths that go "backwards", away from t and back towards s, but in general layered gridgraphs this is a difficult task. However because we are reducing Shuffle to the more general layered gridgraph problem, the graphs will be built in a highly organized fashion that restrict them far beyond normal instances of layered gridgraph. We hoped to use this additional structure to find an algorithm that can remove backwards paths and thus be valid instances of undirected reachability, which would put Shuffle in L. While we did not succeed, we discovered a few of these structural properties that can hopefully be exploited in future works.

2.2 Preliminaries

Definition 8 (Shuffle) *For $a, b, c \in \Sigma^*$ such that $|c| = |a| + |b|$, SHUFFLE($a = a_1 \dots a_m, b = b_1 \dots b_k, c = c_1 \dots c_{m+k}$) accepts iff there is some binary string $s = \{0,1\}^{m+k}$ such that*

1. *s has exactly m bits equal to 0*

2. *$\forall i : 1 \leq i \leq m + k$, $c_i = a_j$ if s_i is the jth 0 in s, or $c_i = b_j$ if s_i is the jth 1 in s*

We think of this string s as a roadmap for how to interleave a and b to create c, noting that we cannot change the internal ordering of a or b.

Definition 9 (Layered gridgraph) *A graph $G = \{V, E\}$ is a layered gridgraph if there exist $m, k \in \mathbb{N}$ such that:*

1. *For each $v \in V$, v is labeled with some coordinate in $[m] \times [k]$ (we say $v = (i, j)$ for some $(i, j) \in [m] \times [k]$). Furthermore, for every coordinate in $[m] \times [k]$, there is exactly one vertex $v \in V$ labeled with that coordinate.*

2. *For every edge $e \in E$ from vertex $u = (u_1, u_2)$ to vertex $v = (v_1, v_2)$, either $v_1 = u_1 + 1$ or $u_2 = v_2 + 1$, but not both.*

Definition 10 (Layered gridgraph) *Let G be a layered gridgraph with parameters m, k. We define $GG(G)$ to accept iff there is a path P from $(1, 1)$ to (m, k).*

Now that we have defined our two problems, we need to make good on our promise from the introduction.

Theorem 3 ([50]) *$SHUFFLE \leq_m GG$*

Because GG is a special case of directed reachability it holds that $GG \in \mathsf{NL}$, also implying that $SHUFFLE \in \mathsf{NL}$.

2.3 Our proposed algorithm

Our goal now is to build an instance of undirected st-connectivity from layered grid-graph. Given $GG(G)$, we build graph $G' = (V', E')$ to have $V' = [m] \times [k]$, and $E' = \{(u, v), (v, u), \forall (u, v) \in E\}$. For an instance of GG we can build G' in logspace, and as discussed in our introduction there exists an algorithm in L to find a path from $(1, 1)$ to (m, k). However, this path does not necessarily correspond to a directed path in the original GG instance, as the path could go from a point (i, j) to a new point $(i, j - 1)$ or $(i - 1, j)$, which would not be possible in GG.

We now propose the following algorithm to solve $GG(G)$:

Algorithm 1 *First we build G' according to our steps above. Now we build G'' as follows: for each node $v = (i, j)$ in the graph:*

1. *Set $G_{v,(1,1)}$ to be G' restricted to only vertices $v' = (i', j')$ such that $i' \leq i$ and $j' \leq j$*

2. *Run Reingold's undirected st-connectivity algorithm from $(1, 1)$ to v on $G_{v,(1,1)}$, and if it rejects we "mark" v*

3. *Set $G_{v,(m,k)}$ to be G' restricted to only vertices $v' = (i', j')$ such that $i' \geq i$ and $j' \geq j$*

4. *Run Reingold's undirected st-connectivity from v to (m, k) on $G'_{v,(m,k)}$, and if it rejects we "mark" v*

We run Reingold's undirected st-connectivity on G'' from $(1, 1)$ to (m, k), but ignoring edges (u, v) where either u or v was marked by our algorithm. Our algorithm then accepts iff this st-connectivity accepts.

We now analyze the space requirements of our algorithm. Our first procedure is to build $G_{v,(1,1)}$, which we do by simply storing i and j. When we run Reingold's

algorithm on $G_{v,(1,1)}$, we do so by ignoring any edges that are on a vertex which have any coordinates not in $[i] \times [j]$, which just involves checking each new vertex before proceeding with the algorithm. We repeat these procedures for $G_{v,(m,k)}$. Our final logspace procedure is the last run of Reingold's algorithm, which runs in logspace if we have access to the list of vertices marked by our algorithm. While we cannot store this list, we can simply compute whether or not a vertex is marked when it is considered by the algorithm, and thus we need not store more than a logarithmic amount of memory. Thus our algorithm runs in logspace.

Now we check its correctness. If the instance of Shuffle is valid, then there is a directed path from $(1,1)$ to (m,k). For every vertex $v = (i,j)$ along that path there will be a path from $(1,1)$ to v in $G_{v,(1,1)}$ and to (m,k) in $G_{v,(m,k)}$. Therefore that path is left intact after vertex checking and removal occurs, and so our algorithm accepts. Likewise, if it is an invalid instance of Shuffle where there is no undirected path in G', there cannot possibly be a path after the removal step, and so our algorithm rejects. The last case to consider–the one case that keeps us from just running undirected reachability on G'–is the case where Shuffle rejects, implying that there is no directed path in GG, but there is an undirected path P in G'. We will show that unfortunately there is a particular circumstance in which our algorithm does not remove this path.

2.4 Where our algorithm breaks down

We consider this final case, where there is no layered path in the instance of GG but there is an undirected path P_{bad} in G'. We now attempt to look at what P_{bad} could look like by attempting to construct a valid path P:

1. initialize $P = \emptyset$, $v = (1,1)$
2. Add v to the path P, and guess a neighbor v' adjacent to v that was not marked by our algorithm to be the next node in P

3. If v' is a layered edge from v, set $v' = v$ and return to step 2

4. If v' is a non-layered edge from v, then because it was not eliminated from our algorithm, there must be a path $P_{v',(1,1)}$ from $(1,1)$ to v in $G_{v',(1,1)}$, and so if that path is only made up of layered edges, set $P = P_{v',(1,1)}$ and return to step 2

5. If this new path $P_{v',(1,1)}$ is not completely made up of layered edges, ERROR

By this process, P is a layered path at each step that does not return an error, assuming we guessed optimally in step 2 every time. Now we consider when this procedure returns ERROR, at vertex $v = (i,j), v' = (i',j')$. Because our procedure has run as normal up until this point, we know that there is a layered path $P_{forward}$ from $(1,1)$ to v. However, because we got ERROR it also must be that there is no layered path that reaches v', and yet it is unmarked by our algorithm. So there must be an unlayered path $P_{backward}$ to v' in $G_{v',(1,1)}$.

Assume without loss of generality that $i = i'+1$. Consider the following gridgraph G_{bad}: $m = 2i-1$ and $k = 2j$. We have two paths, $P_{forward} = \{e_1, e_2, e_3 \dots e_c\}$ and $P_{backward} = \{f_1, f_2, f_3 \dots f_d\}$, both identical to the ones we considered in the previous paragraph. Additionally we have an edge from $(i-1,j)$ to (i,j). Now we include in our graph paths $P'_{forward}$ and $P'_{backward}$ defined as follows: for every (undirected) edge $e_t = ((u_1,u_2),(v_1,v_2)) \in P_{forward}$, set e_{c-t} in $P'_{forward}$ to be $((m-u_1,k-u_2),(m-v_1,k-v_2))$, and analogously for every edge $f_t \in P_{backward}$. What we can think of is "rotating" our original paths by 180° around $(m/2,k/2)$. The end result is that $P_{forward}$ is a layered path from $(1,1)$ to (i,j), while $P'_{forward}$ is a layered path from $(i-1,j)$ to (m,k). Likewise $P_{backward}$ is an unlayered path from $(1,1)$ to $(i-1,j)$, while $P'_{backward}$ is an unlayered path from (i,j) to (m,k).

To see why this is a counterexample, consider running Algorithm 1 on G_{bad}. We will mark some vertices along both $P_{backward}$ and $P'_{backward}$, but leave $P_{forward}$ and

$P'_{forward}$ untouched, as well as leaving the edge $((i-1,j),(i,j))$ untouched due to the presence of $P_{backward}$ and $P'_{backward}$. Thus our last execution of Reingold's algorithm will take $P_{forward}$ from $(1,1)$ to (i,j), follow the edge back to $(i-1,j)$, and then take $P'_{forward}$ from $(i-1,j)$ to (m,k). Note that there was no layered path in G_{bad}, and yet our algorithm will return *true*.

In the general layered gridgraph problem, such a counterexample would instantly break our algorithm. However, when constructing the layered gridgraph for Shuffle, there are many more restrictions that have to be followed.

Theorem 4 *The following equalities hold in any gridgraph constructed according to the reduction in [50] from Shuffle:*

- *if there exist two edges* $e = ((i,j),(i+1,j)), e' = ((i,j'),(i+1,j'))$, *then* $c_{i+j} = a_i = c_{i+j'}$
- *if there exist two edges* $e = ((i,j),(i,j+1)), e' = ((i',j),(i',j+1))$, *then* $c_{i+j} = b_j = c_{i'+j}$
- *if there exist two edges* $e = ((i,j),(i+1,j)), e' = ((i',j'),(i',j'+1))$ *such that* $i+j = i'+j'$, *then* $a_i = c_{i+j} = c_{i'+j'} = b_{j'}$

This theorem follows as a consequence of how the graph was originally constructed. Thus immediately by attempting to construct a counterexample like the one listed above, we note that we have a directed path and an undirected path that do not intersect and have the same opposite corners, meaning we have a closed polygon shape in our grid. But in an enclosed area, using the three rules established in the theorem we can draw equalities between all the various regions on the perimeter of the polygon, and see where equalities between various regions in a, b, and c also force more edges on the inside of the region into existence. By doing so we had hoped to show that all the vertices on the unlayered path also have layered paths, thus rendering the counterexample harmless.

Unfortunately we did discover such a counterexample using the following values:

a : 0100000100000000000000000000010000010000000000010000

00001000000000010000010000000000000000000010000010

b : 0100000100000000000000000001000001000000000001000

00001000000000010000010000000000000000000010000010

c : 0100000100000000000000000001000001000000000100000

10000010000000000000000000010000010000000000010000

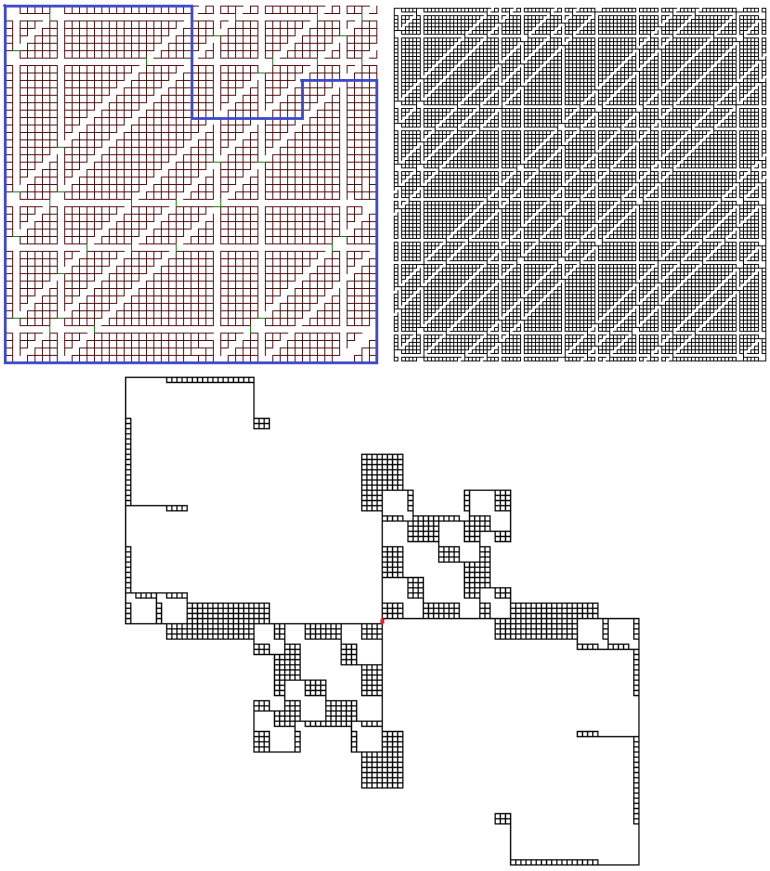
00010000000000100000100000000000000000000001000001

00000100000000001000001000000000000000000010000010

We visualized this example using Java as seen in Fig. 2-1. Note that there seem to be a very dense structure within the initial polygon in (a), and so doing the rotation to create (b) gave us hope that some other layered path from the start to the goal would emerge. However, as we can see in (c), after we run Algorithm 1 the only path left is exactly the unlayered path we predicted when defining these polygons. This was the smallest counterexample we could find via computer-assisted search, so it is clear that due to the edges that are forced to appear with our equalities such examples are rarer than not, but still our algorithm at the current time does not work.

In light of this example, our next step is to test the power of iterating the deletion step of Algorithm 1. It turns out that running it twice in succession solves this example correctly, and as long as we limit ourselves to a constant number of iterations we can perform the calculations in logspace. However, it may be possible to nest these counterexamples *within each other*, which would remove any hope of our algorithm working. Because we have not analyzed the restrictions on Shuffle graphs to the

Figure 2-1: (a) a bad polygon, outlined in blue with necessary edges in black; (b) the full counterexample from before; (c) the result of running our algorithm on the counterexample, with the backwards edge colored red



fullest, it is hard to say at the current time whether or not this will work.

2.5 Conclusion

We have given Shuffle as an example of a problem that fits into the framework of complexity theory. We show a reduction to a reachability problem that is solvable in a class contained in NP but is not known to be as hard as directed reachability nor as easy as undirected reachability. We examine one algorithm that seemed to get us close to proving the latter, and discussed where it breaks down. In the process we give some theorems pertaining to the structure of the layered gridgraphs in question, which can hopefully be analyzed in the future along with possible adjustments to our

algorithm that could lead to putting Shuffle into L.

Chapter 3

Circuit complexity

3.1 Introduction

The next setting we will analyze is the circuit model of complexity theory, which has strong ties to the basic digital logic used in modern computers. We introduced the model at the beginning of this work in relation to graphs which have nodes representing either input variables, the constants 0 and 1, or the basic Boolean functions $\wedge$, $\vee$, and $\neg$. Furthermore we gave a visualization of these nodes as being organized into layers where the top layer contains the single function returning the output of the circuit, and every gate takes inputs only from layers below it.

With this organization, we can define properties of the circuit that will be useful to restrict in our computation models. The *fan-in* of a gate is the number of inputs it takes, and while some gates such as the NOT gate can only ever take one input, AND and OR gates can take any positive number of inputs unless restricted by the model. For the purposes of this work we define the *size* of a circuit to be the number of nodes in the graph, although we could have also chosen the number of wires as well. Finally, the *depth* of a circuit is the number of layers in the graph, which is equivalent to the length of the longest path from a node labeled with an input variable or constant to

the output gate.

Our focus in this section will be on circuits that have polynomial size and logarithmic depth in relation to the number of input variables to the circuit, n. There are many important circuit classes that have different restrictions than this model, such as constant depth circuits, but while many of our results carry over to different classes we will only mention such extensions in passing, leaving it to the more interested reader to find such places as they see fit.

There are two other gates we will use throughout this section that have not been discussed thus far. The MOD_m gate takes as input any positive number of Boolean values and returns 1 iff the number of 1-valued inputs is a multiple of m. For example, the MOD_2 gate, more commonly known as the PARITY gate ($\oplus$), returns 1 iff there are an even number of 1-valued inputs. The other gate is the MAJORITY gate, which again takes any positive number of Boolean inputs and returns 1 iff there are at least as many 1-inputs as 0-inputs, or in other words, returns whether the majority of the inputs are 1 or 0.

An important study in circuit complexity is that of *arithmetic circuits*, circuits whose underlying expression is a polynomial over a field rather than a Boolean formula. The changes from Boolean circuits are that wires now carry numbers rather than just Boolean bits, and gates are labeled by arithmetic functions such as $+$ or $\times$ rather than Boolean functions such as $\wedge$ and $\vee$. In the settings we will analyze, there is an underlying ring restricting the circuit, and so all computations are carried out within that field. Furthermore in this work we generally restrict ourselves to finite fields or rings, namely $\mathbb{F}_p$ and $\mathbb{Z}_m$. For completeness sake the basic field identities used in this work will be included in the preliminaries in brief.

Perhaps the most famous class of arithmetic circuits is $\mathsf{VP}(\mathbb{F}_p)$. This class was originally introduced by Valiant [55] as the class of polynomials f over $\mathbb{F}_p$ that have polynomial degree and can be represented by poly-size arithmetic circuits, which is

equivalent to the class of poly-size log-depth arithmetic circuits over $\mathbb{F}_p$ where $+$ gates have unbounded fan-in and $\times$ gates have fan-in two [56, 2]. Though the arithmetic circuits tend to receive less attention than Boolean circuits, we will see that this class, along with other arithmetic circuit models under consideration, can be seen as an "arithmetic version" of natural Boolean circuit classes. Thus we will attempt to better characterize arithmetic circuit complexity classes in the classical framework of Boolean circuit complexity and vice versa.

Our other goal besides characterizing these lesser-known classes will be fan-in reductions, seeing how much power is lost–if any–by restricting the fan-in of certain gates in a given model. Note that in the arithmetic complexity setting, restricting the fan-in of $\times$ gates is the same as restricting the degree of the underlying polynomials, which mathematics tells us should yield a strictly weaker class of circuits. Yet over $\mathbb{F}_2$ we show that the problem of querying a polynomial of degree $n^{\log n}$ reduces to the problem of querying a polynomial of degree $n^{\log \log n}$.

3.2 Preliminaries

Before introducing our main circuit classes, we will make good on our promise from before and introduce a few equations necessary for understanding our reductions, as well as DeMorgan's Laws, a staple in Boolean circuit complexity.

Definition 11 (DeMorgan's Laws) *For Boolean variables $x_1 \dots x_m$, $\neg(\wedge(x_1 \dots x_m)) = \vee(\overline{x_1} \dots \overline{x_m})$ and $\neg(\vee(x_1 \dots x_m)) = \wedge(\overline{x_1} \dots \overline{x_n})$, where $\overline{x_i}$ denotes $\neg x_i$.*

Definition 12 (Fermat's Little Theorem) *For all prime p and $x \in \mathbb{Z}$, $(x^{p-1} \equiv 1) mod\ p$. Equivalently, $x^{p-1} = 1$ over $\mathbb{F}_p$.*

Although we do not assume that the reader is familiar with Boolean circuit complexity classes such as AC^0 and ACC^0, we recommend an excellent text by Vollmer

[58], which sums up relevant circuit classes, commonly used notation in the field, and important results in circuit complexity.

Definition 13

- *A family of circuits* $\{C_n \mid n \in \mathbb{N}\}$ *is logspace uniform if it can be defined by a logspace algorithm* A *that maps* 1^n *to* C_n*, the unique circuit in the family that takes* n *inputs.*

- AC^i *is the class of languages accepted by logspace uniform circuit families of polynomial size and depth* $O(\log^i n)$*, consisting of unbounded fan-in* AND*, and* OR *gates, along with* NOT *gates.*

- $\mathsf{AC}^i[m]$ *is defined as* AC^i*, but in addition unbounded fan-in* MOD_m *gates are allowed, which output 1 iff the number of input wires carrying a value of 1 is a multiple of* m*.*

- *For any finite set* $S \subset \mathbb{N}$*,* $\mathsf{AC}^i[S]$ *is defined analogously to* $\mathsf{AC}^i[m]$*, but now the circuit families are allowed to use* MOD_r *gates for any* $r \in S$*. It is known that, for any* $m \in \mathbb{N}, \mathsf{AC}^i[m] = \mathsf{AC}^i[\mathsf{Supp}(m)]$*, where – following the notation of [29] –* $\mathsf{Supp}(m) = \{p : p \text{ is prime and } p \text{ divides } m\}$ *[49]. Thus, in particular* $\mathsf{AC}^i[6] = \mathsf{AC}^i[2,3]$ *and* $\mathsf{AC}^i = \mathsf{AC}^i[\emptyset]$*. (When it will not cause confusion, we omit unnecessary brackets, writing for instance* $\mathsf{AC}^i[2,3]$ *instead of* $\mathsf{AC}^i[\{2,3\}]$*.)*

- $\mathsf{ACC}^i = \bigcup_m \mathsf{AC}^i[m]$.

- TC^i *is the class of languages accepted by Dlogtime-uniform circuit families of polynomial size and depth* $O(\log^i n)$*, consisting of unbounded fan-in* MAJORITY *gates, along with* NOT *gates.*

- SAC^i *is the class of languages accepted by Dlogtime-uniform circuit families of polynomial size and depth* $O(\log^i n)$*, consisting of unbounded fan-in* OR *gates and bounded fan-in* AND *gates, along with* NOT *gates at (some of) the leaves.*

Note that the restriction that NOT gates appear only at the leaves in SAC^i circuits is essential; if NOT gates were allowed to appear everywhere, then using DeMorgan's Laws we could get unbounded fan-in gates of both types, and these classes would coincide with AC^i. Similarly, note that we do not bother to define a complexity class $\mathsf{SAC}^i[m]$, since a MOD_m gate with a single input wire is equivalent to a NOT gate, and thus $\mathsf{SAC}^i[m]$ would be the same as $\mathsf{AC}^i[m]$.

In this paper, we focus on uniform circuit families, and thus we use the notation $\mathsf{VP}(R)$ to denote the families of polynomials that result when we impose a logspace-uniformity condition on the circuit families. The algebraic complexity classes $\mathsf{VP}(R)$ for various algebraic structures R were originally defined [55] in the context of nonuniform circuit complexity, as classes of families of n-variate polynomials of degree $n^{O(1)}$ that can be represented by polynomial-size arithmetic circuits over R. (For more on VP, see, e.g. [23, 22, 34, 44].) In the original nonuniform setting, it was shown by [56] that the circuits defining polynomials in $\mathsf{VP}(R)$ can be assumed to have small depth. Later [2] a slightly improved characterization was provided, that works also in the context of uniform circuit complexity:

Theorem 5 *[2] For any commutative semiring R, $\mathsf{VP}(R)$ coincides with the class of families of polynomials over R represented by logspace-uniform circuit families of polynomial size and logarithmic depth with unbounded fan-in $+$ gates, and fan-in two $\times$ gates.*

Note that over $\mathbb{F}_p$, many different polynomials yield the same function. For example, since $x^3 = x$ in $\mathbb{F}_3$, every function on n variables has a polynomial of degree at most $2n$. Very likely there are functions represented by polynomials in $\mathsf{VP}(\mathbb{F}_3)$ of degree, say, n^5, but not by any VP polynomial of degree $2n$. On the other hand, there is a case to be made for focusing on the *functions* in these classes, rather than focusing on the *polynomials* that represent those functions. For instance, one bold conjecture posed by Immerman and Landau posits that TC^1 is reducible to problems in $\mathsf{VP}(\mathbb{Q})$,

and it would suffice for every *function* in $\mathsf{TC}^1 = \#\mathsf{AC}^1(\mathbb{F}_{p_n})$ to have a representation in $\mathsf{VP}(\mathbb{Q})$, even though the *polynomials* represented by $\#\mathsf{AC}^1(\mathbb{F}_{p_n})$ circuits have large degree, and thus cannot be in any VP class.

In the literature on VP classes, one standard way to focus on the *functions* represented by polynomials in VP is to consider what is called the *Boolean Part* of $\mathsf{VP}(R)$, which is the set of *languages* $A \subseteq \{0,1\}^*$ such that, for some sequence of polynomials (Q_n), for $x \in A$ we have $Q_{|x|}(x) = 1$, and for $x \in \{0,1\}^*$ such that $x \notin A$ we have $Q_{|x|}(x) = 0$.

When the algebra R is a finite field, considering the Boolean part of $\mathsf{VP}(R)$ captures the relevant complexity aspects, since the computation of any function represented by a polynomial in $\mathsf{VP}(R)$ (with inputs and outputs coming from R) is logspace-Turing reducible to some language in the Boolean Part of $\mathsf{VP}(R)$.

In this paper, we will be concerned exclusively with the "Boolean Part" of various arithmetic classes. For notational convenience, we will just refer to these classes using the "VP*" notation, rather than constantly repeating the phrase "Boolean Part".*

Following the standard naming conventions of [58], for any Boolean circuit complexity class $\mathcal{C}$ defined in terms of circuits with AND and OR gates, we define the class $\#\mathcal{C}(R)$ to be the class of functions represented by arithmetic circuits defined over the algebra R, where AND is replaced by $\times$, and OR is replaced by $+$ (and NOT gates at the leaves are applied to the $\{0,1\}$ inputs).[1] In particular, we will be concerned with the following two classes:

Definition 14 *Let R be any suitable semiring.*[2] *Then*

- $\#\mathsf{AC}^1(R)$ *is the class of functions $f : \{0,1\}^* \to R$ represented by families*

[1] The classes $\#\mathsf{L}$, $\#\mathsf{P}$ and $\#\mathsf{LogCFL}$ also fit into this naming scheme, using established connections between Turing machines and circuits.

[2] Our primary focus in this paper is on *finite* semirings, as well as countable semirings such as $\mathbb{Q}$, where we use the standard binary representation of constants (say, as a numerator and denominator) when a logspace uniformity machine makes use of constants in the description of a circuit. It is not clear to us which definition would be most useful in describing a class such as $\#\mathsf{AC}^1(\mathbb{R})$, and so for now we consider such semirings to be "unsuitable".

of logspace-uniform circuits of unbounded fan-in $+$ and $\times$ gates having depth $O(\log n)$ and polynomial size.

- $\#\mathsf{SAC}^1(R)$ *is the class of functions $f : \{0,1\}^* \to R$ represented by families of logspace-uniform circuits of unbounded fan-in $+$ gates and $\times$ gates of fan-in two, having depth $O(\log n)$ and polynomial size.*

Input variables may be negated. Where no confusion will result, the notation $\#\mathcal{C}(R)$ will also be used to refer to the class of languages whose characteristic functions lie in the given class.

Hence from Theorem 5 we obtain:

Proposition 1 *Let p be a prime power. Then $\mathsf{VP}(\mathbb{F}_p) = \#\mathsf{SAC}^1(\mathbb{F}_p)$.*

Proof: The inclusion $\mathsf{VP}(\mathbb{F}_p) \subseteq \#\mathsf{SAC}^1(\mathbb{F}_p)$ is immediate from Theorem 5. The $\#\mathsf{SAC}^1(\mathbb{F}_p)$ circuit that is created for a $\mathsf{VP}(\mathbb{F}_p)$ circuit has no NOT gates. For the converse inclusion, given a $\#\mathsf{SAC}^1(\mathbb{F}_p)$ circuit family, each NOT gate at a leaf, connected to input x_i can be replaced by $(x_i + (p-1))^2$. □

3.2.1 New Definitions: Λ-classes

In this section, we introduce and define classes that are dual to the $\#\mathsf{SAC}^1(R)$ classes discussed above. Define $\#\mathsf{SAC}^{1,*}(R)$ to be the class of functions $f : \{0,1\}^* \to R$ represented by families of logspace-uniform circuits of unbounded fan-in $\times$ gates and $+$ gates of fan-in two, having depth $O(\log n)$ and polynomial size. Proposition 1 highlights the connection between VP and $\#\mathsf{SAC}^1$; thus we will utilize the convenient notation $\Lambda\mathsf{P}(R)$ to denote the dual notation, rather than the more cumbersome $\#\mathsf{SAC}^{1,*}(R)$.

Of course, the set of formal polynomials represented by $\Lambda\mathsf{P}$ circuits is not contained in any VP class, because $\Lambda\mathsf{P}$ contains polynomials of degree $n^{O(\log n)}$. However, as

discussed in the previous section, we are considering the "Boolean Part" of these classes. More formally:

Definition 15 *Let p be a prime power. $\Lambda\mathsf{P}(\mathbb{F}_p)$ is the class of all languages $A \subseteq \{0,1\}^*$ with the property that there is a logspace-uniform family of circuits $\{C_n : n \in \mathbb{N}\}$ such that*

- *The depth of C_n is $O(\log n)$.*
- *Each C_n consists of input gates, $+$ gates, and $\times$ gates.*
- *Each $+$ gate has fan-in two, whereas there is no bound on the fan-in of the $\times$ gates.*[3]
- *For each string x of length n, x is in A if and only if $C_n(x)$ evaluates to 1, when the $+$ and $\times$ gates are evaluated over $\mathbb{F}_p$. Furthermore, if $x \notin A$, then $C_n(x)$ evaluates to 0.*

Another way of relating arithmetic classes (such as VP and $\Lambda\mathsf{P}$) to complexity classes of languages would be to consider the languages that are logspace-Turing reducible to the polynomials in $\mathsf{VP}(R)$ or $\Lambda\mathsf{P}(R)$, via a machine M with a polynomial p as an oracle, which obtains the value of $p(x_1, \ldots, x_n)$ when M writes $x_1, \ldots x_n$ on a query tape. It is worth mentioning that (the Boolean parts of) both $\mathsf{VP}(\mathbb{F}_p)$ and $\Lambda\mathsf{P}(\mathbb{F}_p)$ are closed under logspace-Turing reductions, although this is still open for classes over $\mathbb{Z}_m$ when m is not prime.

We mention that VP classes over different fields of the same characteristic define the same class of languages. This seems to be one way that the VP and $\Lambda\mathsf{P}$ classes differ; see Corollary 2.

Proposition 2 *Let p be a prime, and let $k \geq 1$. Then $\mathsf{VP}(\mathbb{F}_p) = \mathsf{VP}(\mathbb{F}_{p^k})$.*

[3]The uniformity condition imposes an implicit polynomial bound on the fan-in of any gate.

Proof: One inclusion follows immediately since $\mathbb{F}_p$ is a subfield of $\mathbb{F}_{p_k}$. For the other direction, observe that the finite field of size p^k is a vector space of dimension k over the field of size p, and thus can be represented by $k \times k$ matrices over $\mathbb{F}_p$, as described in [35]. (See also [59].) Thus each + and $\times$ gate of a $\Lambda\mathsf{P}(\mathbb{F}_{p^k})$ circuit can be replaced by subcircuits implementing matrix sum and product over $\mathbb{F}_p$. (Unbounded fan-in matrix sum corresponds to unbounded fan-in sum of each component. Fan-in two multiplication is implemented by a depth-two subcircuit, with fan-in two $\times$ gates, and with addition gates of fan-in $O(1)$.) The resulting circuit is a $\mathsf{VP}(\mathbb{F}_p)$ circuit. □

It is also appropriate to use the VP and ΛP notation when referring to the classes defined by Boolean semiunbounded fan-in circuits with negation gates allowed at the inputs. With this notation, $\mathsf{VP}(B_2)$ corresponds to the Boolean class SAC^1, and $\Lambda\mathsf{P}(B_2)$ corresponds to the complement of SAC^1 (with bounded fan-in OR gates, unbounded fan-in AND gates and negation gates allowed at the inputs). It has been shown by [20] that SAC^1 is closed under complement. Thus we close this section with the equality that serves as a springboard for investigating the ΛP classes.

Theorem 6 *[20]* $\mathsf{VP}(B_2) = \Lambda\mathsf{P}(B_2) (= \mathsf{SAC}^1 = \mathsf{LogCFL})$.

We do not believe that $\mathsf{VP}(\mathbb{F}_p) = \Lambda\mathsf{P}(\mathbb{F}_p)$ for any prime p; see further related discussion in Section 3.5.

3.3 Subclasses of ACC^1

In this section, we present our characterizations of ACC^1 in terms of the $\Lambda\mathsf{P}(\mathbb{F}_{p^k})$ classes.

Theorem 7 *For any prime p and any $k \geq 1$, $\Lambda\mathsf{P}(\mathbb{F}_{p^k}) = \mathsf{AC}^1[\mathsf{Supp}(p^k - 1)]$. (Recall that $\mathsf{Supp}(m)$ is defined in Definition 13.)*

Proof: ($\subseteq$): Consider a $\Lambda\mathsf{P}(\mathbb{F}_{p^k})$ circuit C. We will create a circuit C' that has subcircuits computing the Boolean value $[g = a]$ for each gate g in C and for each $a \in \mathbb{F}_{p^k}$. (We will use the notation "$[B]$" to refer to the truth-value of predicate B.) If g is the output gate of C, then the output gate of C' is the gate $[g = 1]$. Since the input gates of C take on only binary values (by our definition of $\Lambda\mathsf{P}(\mathbb{F}_{p^k})$), if g is an input gate of C, then the subcircuit $[g = 1]$ is just g, and the subcircuit for $[g = 0]$ is $\neg g$. If g is a constant gate, set to the value $a \in \mathbb{F}_{p^k}$, then $[g = a]$ is set to the constant 1, and $[g = a']$ is set to the constant 0, for each $a' \neq a$.

If g is a + gate of C (of fan-in 2), then any gate $[g = a]$ can be simulated with NC^0 circuitry using the $O(1)$ Boolean gates of the form $[g' = a']$, where g' feeds into g in C.

Now consider a $\times$ gate g of C, having unbounded fan-in: $g = \prod_i h_i$. The value $[g = 0]$ is obtained by simply checking if there is some i such that $h_i = 0$.

Now we show how to compute $[g = a]$ for $a \neq 0$. Let $p^k - 1 = \prod_{j=1}^{\ell} q_j^{e_j}$ where $\mathsf{Supp}(p^k - 1) = \{q_1, \ldots, q_\ell\}$. Let σ be a generator of the multiplicative group of $\mathbb{F}_{p^k}$. Then $g = \prod_i h_i = \prod_i \sigma^{\log h_i} = \sigma^{\sum_i \log h_i}$ where "$\log b$" denotes the unique element of $\mathbb{F}_{p^k}$ such that $\sigma^{\log b} = b$. Hence the value $[g = a]$ is equivalent to $[\log a \equiv \sum_i \log h_i \bmod (p^k - 1)]$, which in turn is equivalent to the AND of the values $[\log a \equiv \sum_i \log h_i \bmod (q_j^{e_j})]$.

If $e_j = 1$ then the value $[\log a \equiv \sum_i \log h_i \bmod (q_j)]$ is easy to compute with a MOD_{q_j} gate, as follows. Using NC^0 circuitry, for each i, find the unique b such that $[h_i = b]$ holds (and for simplicity, let us refer to this value as h_i). Then, for each i, compute the string $x_i = 1^{\log h_i} 0^{p - \log h_i}$. (Note that the mapping from gates of the form $[h_i = b]$ to x_i is computable in logspace-uniform NC^0.) Let X_a be the string that results from concatenating the string $1^{(p-1)-\log a}$ and all of the strings x_i. Now observe that feeding X_a into a MOD_{q_j} gate computes the value $[\log a \equiv \sum_i \log h_i \bmod (q_j)]$.

If $e_j > 1$, then first observe that $[b \equiv 0 \bmod q_j^{e_j}]$ can be computed by checking if each of $b, \binom{b}{q_j}, \binom{b}{q_j^2}, \ldots \binom{b}{q_j^{e_j-1}}$ is equivalent to 0 mod q_j. (See, e.g. [18, Fact 2.2].) Observe also that $\binom{b}{d}$ can be represented as the number of different AND gates of fan-in d that evaluate to 1, taking inputs from the string X_b. Thus all of these conditions can be checked in constant depth with MOD_{q_j} gates and bounded fan-in AND gates.

Since C has depth $O(\log n)$, and C' consists of layers of constant-depth circuitry to replace each layer of gates in C, this completes the proof of this direction.

($\supseteq$): Given an $\mathsf{AC}^1[\mathsf{Supp}(p^k-1)]$ circuit C, we show how to construct an arithmetic circuit C' that is equivalent to C. Each gate g of C will have an equivalent gate g in C'. The input gates of C and of C' are exactly the same.

If g is a NOT gate in C, say $g = \neg h$, then in C' we will have $g = (h + (p-1)) \times (h + (p-1))$.

If g is an AND gate (say, $g = \wedge_i h_i$), then in C' we will have $g = \prod_i h_i$. OR gates will be handled the same way, using De Morgan's Laws.

Now consider the case when g is a MOD_{q_j} gate with inputs h_i. Thus g computes the value $[\sum_i h_i \equiv 0 \bmod q_j]$. Let σ be a generator of the multiplicative cyclic subgroup of size q_j. First map each h_i to the value $h'_i = 1 + ((\sigma + (p^k - 1)) \times h_i$, and observe that $h'_i = \sigma^{h_i}$ for all $h_i \in \{0, 1\}$. Observe that $1 - \prod_i h'_i = 1 - \sigma^{\sum_i h_i}$ is equal to 0 if $\sum_i h_i$ is a multiple of q_j, and is non-zero otherwise. Thus $1 - (1 - \prod_i h'_i)^{p^k-1}$ is equal to the Boolean value $[\sum_i h_i \equiv 0 \bmod q_j]$.

It is easy to verify that C' has logarithmic depth, and uses only bounded fan-in + gates, as well as unbounded fan-in $\times$ gates. □

Corollary 1 $\mathsf{ACC}^1 = \bigcup_p \Lambda\mathsf{P}(\mathbb{F}_p)$.

Proof: Let $A \in \mathsf{ACC}^1$. Thus $A \in \mathsf{AC}^1[m]$ for some modulus m.

By Dirichlet's Theorem, the arithmetic progression $m+1, 2m+1, \ldots$ contains some prime p. Thus $\mathsf{AC}^1[m] \subseteq \mathsf{AC}^1[\mathsf{Supp}(p-1)] = \Lambda\mathsf{P}(\mathbb{F}_p)$. □

Note also that several of the $\Lambda\mathsf{P}(\mathbb{F}_p)$ classes coincide. This is neither known nor believed to happen with the $\mathsf{VP}(\mathbb{F}_p)$ classes.

Corollary 2

- $\Lambda\mathsf{P}(\mathbb{F}_2) = \mathsf{AC}^1$, *whereas* $\Lambda\mathsf{P}(\mathbb{F}_4) = \mathsf{AC}^1[3]$. *Note that this contrasts with the equality* $\mathsf{VP}(\mathbb{F}_2) = \mathsf{VP}(\mathbb{F}_4)$ *given by Proposition 2.*
- *If p is a Fermat prime (that is, $p-1$ is a power of 2, such as $p \in \{3, 5, 17, 257, 65{,}537\}$), then* $\Lambda\mathsf{P}(\mathbb{F}_p) = \mathsf{AC}^1[2]$.
- $\Lambda\mathsf{P}(\mathbb{F}_7) = \Lambda\mathsf{P}(\mathbb{F}_{13}) = \Lambda\mathsf{P}(\mathbb{F}_{19})$.
- *More generally,* $\mathsf{Supp}(p-1) = \mathsf{Supp}(q-1)$ *implies* $\Lambda\mathsf{P}(\mathbb{F}_p) = \Lambda\mathsf{P}(\mathbb{F}_q)$.

Augmenting the $\Lambda\mathsf{P}(\mathbb{F}_p)$ classes with unbounded fan-in addition gates increases their computation power only by adding MOD_p gates, as the following theorem demonstrates.

Theorem 8 *For each prime p and each $k \geq 1$,* $\#\mathsf{AC}^1(\mathbb{F}_{p^k}) = \mathsf{AC}^1[\{p\} \cup \mathsf{Supp}(p^k - 1)]$.

Proof: ($\subseteq$): Again, we use a gate-by-gate simulation, with subcircuits recording the value of $[g = a]$ for each gate g and each $a \in \mathbb{F}_p$. Multiplication gates are handled as in the proof of Theorem 7. Consider now the case of an addition gate $g = \sum_i h_i$.

Using NC^0 circuitry, one can use the gates $[h_i = b]$ to compute the string $y_i = 1^{h_i}0^{p-h_i}$ (as in the proof of Theorem 7). Let Y_a be the string 1^{p-a} concatenated with all of the strings y_i. Feeding Y_a into a MOD_p gate computes the Boolean value $[g = a]$.

($\supseteq$): As in Theorem 7, we carry out a gate-by-gate simulation, whereby each gate g in a $\mathsf{AC}^1[\{p\} \cup \mathsf{Supp}(p^k - 1)]$ circuit C is equivalent to a gate (also called g) in a

$\#\mathsf{AC}^1(\mathbb{F}_p)$ circuit C'. We only need to consider the case where g is a MOD_p gate with Boolean inputs h_i. In this case, note that $g = 1 + ((\sum_i h_i)^{p^k-1} \times (p-1))$. □

Corollary 3 $\mathsf{ACC}^1 = \bigcup_p \Lambda\mathsf{P}(\mathbb{F}_p) = \bigcup_p \#\mathsf{AC}^1(\mathbb{F}_p) = \bigcup_m \#\mathsf{AC}^1(\mathbb{Z}_m)$.

Proof: All inclusions are immediate from Theorems 7 and 8, except for $\#\mathsf{AC}^1(\mathbb{Z}_m) \subseteq \mathsf{ACC}^1$. Consider a circuit C for some function in $\#\mathsf{AC}^1(\mathbb{Z}_m)$. Again, we will build an ACC^1 circuit C' with gates of the form $[g = a]$ for each gate g in C and each $a \in \mathbb{Z}_m$. Addition is handled as in the proof of Theorem 8. Thus consider a multiplication gate $g = \prod_i h_i = \prod_j a_j^{e_j}$, where $e_j = |\{i : [h_i = a_j]\}|$. The sequence $(a_j^0, a_j^1, a_j^2, \ldots)$ (where the product is interpreted in $\mathbb{Z}_m$) is ultimately periodic with a period less than m, and thus the value of $[a_j^{e_j} = b]$ can be computed using AC^0 circuitry and a MOD gate, using inputs of the form $[h_i = a_j]$ for various values of i. Then $[g = a]$ can be computed in NC^0 using the $O(1)$ gates of the form $[a_j^{e_j} = b]$. □

Corollary 4 *For any prime p there is a prime q such that $\#\mathsf{AC}^1(\mathbb{F}_p) \subseteq \Lambda\mathsf{P}(\mathbb{F}_q)$.*

Proof: By Dirichlet's Theorem, there is a prime q such that $q-1$ is a multiple of $p(p-1)$. The claim now follows immediately from Theorems 8 and 7. □

It will be useful to bear in mind that $\mathsf{VP}(\mathbb{F}_p)$ also has a simple characterization in terms of Boolean circuits. In order to present this characterization, we present a more general definition, which will be needed later.

Definition 16 *Let $m \in \mathbb{N}$, and let g be any function on $\mathbb{N}$. Define g-$\mathsf{AC}^i[m]$ to be the class of languages with logspace-uniform circuits of polynomial size and depth $O(\log^i n)$, consisting of unbounded-fan-in MOD_m gates, along with AND gates of fan-in $O(g(n))$. Clearly g-$\mathsf{AC}^i[m] \subseteq \mathsf{AC}^i[m]$.*

When $g(n) = O(1)$, the class g-$\mathsf{AC}^i[m]$ coincides with the class $\mathsf{CC}^i[m]$, which was defined by Straubing [51, p. 141] for the special case $i = 0$, and which has been studied

subsequently in e.g. [37, 52, 36]. If $m > 2$, then no AND *or* OR *gates are needed at all [51, Chapter VIII, Exercise 9]. Thus some authors define* $\mathsf{CC}^i[m]$ *in terms of circuits consisting only of* MOD_m *gates, but the original definition is more convenient for our purposes. The class* CC^i *is defined to be* $\bigcup_m \mathsf{CC}^i[m]$*, analogously to* ACC^i*.*

Observe that, since a MOD_m gate can simulate a NOT gate, g-$\mathsf{AC}^1[m]$ remains the same if OR gates of fan-in $O(g)$ are also allowed.

Corollary 5 *For every prime p, $\mathsf{VP}(\mathbb{F}_p) = \mathsf{CC}^1[p] \subseteq \mathsf{AC}^1[p]$.*

Proof: Recall that $\mathsf{VP}(\mathbb{F}_p) = \#\mathsf{SAC}^1(\mathbb{F}_p)$. Thus we need only show how to simulate bounded fan-in $\times$ gates and unbounded fan-in $+$ gates. Bounded fan-in $\times$ gates can be simulated in $O(1)$ depth using AND and OR gates of fan-in two (since the values being multiplied are of size $O(1)$). Unbounded fan-in $+$ gates can be simulated using MOD_p gates, as in the proof of Theorem 7.

For the converse inclusion, consider a $\mathsf{CC}^1[p]$ circuit. Since a unary MOD_p gate is equivalent to a NOT gate, we can assume that the circuit has only fan-in two AND gates and unbounded fan-in MOD_p gates. Thus each Boolean AND gate can be simulated by a fan-in two multiplication gate, and the MOD_p gates can be simulated as in the proof of Theorem 8. □

We remark that the same proof shows that, for any $m \in \mathbb{N}$, $\mathsf{VP}(\mathbb{Z}_m) \subseteq \mathsf{CC}^1[m]$. However, the converse inclusion is not known, unless m is prime.

We remark that the proofs of Theorems 7 and 8 carry over also for depths $\log^i n$ for every $i \geq 0$. (Related results for constant-depth unbounded-fan-in circuits can be found already in [49, 1].)

Corollary 6 *For any prime p and for every $i \geq 0$, $\#\mathsf{SAC}^{i,*}(\mathbb{F}_p) = \mathsf{AC}^i[\mathsf{Supp}(p-1)]$ and $\#\mathsf{AC}^i(\mathbb{F}_p) = \mathsf{AC}^i[p \cup \mathsf{Supp}(p-1)]$. In particular, $\mathsf{ACC}^i = \bigcup_p \#\mathsf{SAC}^{i,*}(\mathbb{F}_p)$.*

3.3.1 Comparing $\Lambda\mathsf{P}$ and VP.

How do the $\Lambda\mathsf{P}$ and VP classes compare to each other?

As a consequence of Corollary 5 and Theorem 7, $\mathsf{VP}(\mathbb{F}_p) \subseteq \Lambda\mathsf{P}(\mathbb{F}_q)$ whenever p divides $q-1$. In particular, $\mathsf{VP}(\mathbb{F}_2) \subseteq \Lambda\mathsf{P}(\mathbb{F}_q)$ for any prime $q > 2$. No inclusion of any $\Lambda\mathsf{P}$ class in any VP class is known unconditionally, although $\Lambda\mathsf{P}(B_2)(= \mathsf{SAC}^1)$ is contained in every $\mathsf{VP}(\mathbb{F}_p)$ class in the nonuniform setting [34, 47], and this holds also in the uniform setting under a plausible derandomization hypothesis [5].

No $\Lambda\mathsf{P}(\mathbb{F}_q)$ class can be contained in $\mathsf{VP}(\mathbb{F}_p)$ unless $\mathsf{AC}^1 \subseteq \mathsf{VP}(\mathbb{F}_p)$, since $\mathsf{AC}^1 = \Lambda\mathsf{P}(\mathbb{F}_2) \subseteq \Lambda\mathsf{P}(\mathbb{F}_3) \subseteq \Lambda\mathsf{P}(\mathbb{F}_q)$ for every prime $q \geq 3$. AC^1 is not known to be contained in any VP class, although we return to this topic again in Section 3.4

3.4 Threshold circuits and small degree

The inspiration for the results in this section comes from the following theorem of Reif and Tate [45] (as re-stated by Buhrman *et al.* [21]):

Theorem 9 $\mathsf{TC}^1 = \#\mathsf{AC}^1(\mathbb{F}_{p_n})$.

Here, the class $\#\mathsf{AC}^1(\mathbb{F}_{p_n})$ consists of the languages whose (Boolean) characteristic functions are computed by logspace-uniform families of arithmetic circuits of logarithmic depth with unbounded fan-in $+$ and $\times$ gates, where the arithmetic operations of the circuit C_n are interpreted over $\mathbb{F}_{p_n}$, where $p_1, p_2, p_3, \ldots$ is the sequence of all primes $2, 3, 5, \ldots$ That is, circuits for inputs of length n use the n-th prime to define the algebraic structure.

This class is closed under logspace-Turing reductions – but when we consider *other* circuit complexity classes defined using $\mathbb{F}_{p_n}$, it is *not* clear that these other classes are closed under logspace-Turing reductions.

As an important example, we mention $\mathsf{VP}(\mathbb{F}_{p_n})$. As we show below, this class has an important connection to $\mathsf{VP}(\mathbb{Q})$, which is perhaps the canonical example of a VP

class. Vinay [57] proved that $\mathsf{VP}(\mathbb{Q})$ has essentially the same computational power as $\#\mathsf{LogCFL}$ (which counts among its complete problems the problem of determining how many distinct parse trees a string x has in a certain context-free language). Here, we mention one more alternative characterization of the computational power of $\mathsf{VP}(\mathbb{Q})$.

Proposition 3 $\mathsf{L}^{\mathsf{VP}(\mathbb{F}_{p_n})} = \mathsf{L}^{\mathsf{VP}(\mathbb{Q})} = \mathsf{L}^{\#\mathsf{LogCFL}}$.

Proof: Consider the first equality. If one wants to compute the value of a $\mathsf{VP}(\mathbb{F}_{p_n})$ circuit on a given input of length n, in logspace one can first compute the value of p_n. Then one can use a $\mathsf{VP}(\mathbb{Q})$ oracle to evaluate the $\mathsf{VP}(\mathbb{F}_{p_n})$ circuit over the rationals instead of over $\mathbb{F}_{p_n}$, obtaining an integer result. Then one can divide the result by p_n and obtain the remainder, which is the value of the circuit in $\mathbb{F}_{p_n}$, using the fact that division is computable in logspace [27, 38].

Conversely, if one wants to evaluate a $\mathsf{VP}(\mathbb{Q})$ circuit on a given n-tuple of rationals, one can use the standard technique of computing the numerator and denominator separately; the circuits for these functions are also in $\mathsf{VP}(\mathbb{Q})$. Thus our task boils down to evaluating an integer-valued arithmetic circuit C_n. To do this, we use Chinese remaindering, and evaluate circuits (with some dummy variables) over the primes $p_n, p_{n+1}, \ldots, p_{n+n^c}$ for some constant c. Converting between Chinese remainder representation and binary representation can be accomplished in logspace [27, 38], which completes the proof of the first equality.

For the second equality, we similarly use the fact that $\mathsf{VP}(\mathbb{Q})$ circuits with integer coefficients and inputs can be evaluated in $\#\mathsf{LogCFL}$, and appeal to [57]. □

When we consider arithmetic circuits of superpolynomial algebraic degree (such as the $\Lambda\mathsf{P}$ classes), evaluating the circuits over the integers can produce outputs that require a superpolynomial number of bits to express in binary. Thus, when we

consider such classes, it will always be in the context of structures (such as $\mathbb{F}_{p_n}$) where the output can always be represented in a polynomial number of bits.

Our first new result in this section, is to improve Theorem 9.

Theorem 10 $\mathsf{TC}^1 = \#\mathsf{AC}^1(\mathbb{F}_{p_n}) = \mathsf{L}^{\Lambda\mathsf{P}(\mathbb{F}_{p_n})}$.

Proof: The first equality is due to [45]. The inclusion of $\mathsf{L}^{\Lambda\mathsf{P}(\mathbb{F}_{p_n})}$ in TC^1 follows since $\Lambda\mathsf{P}(\mathbb{F}_{p_n})$ is a subclass of $\#\mathsf{AC}^1(\mathbb{F}_{p_n})$ and TC^1 is closed under logspace-Turing reducibility.

We will show how to simulate a $\#\mathsf{AC}^1(\mathbb{F}_{p_n})$ circuit C, by making calls to an appropriate function in $\Lambda\mathsf{P}(\mathbb{F}_{p_n})$. The first step is to find a prime q that is not too much larger than p_n, such that $q-1$ is a multiple of $p_n(p_n-1)$. Xylouris [60] has shown that the sequence $1+p_n(p_n-1), 1+2p_n(p_n-1), 1+3p_n(p_n-1)\ldots$ contains a prime of size $O(p_n^{10.4})$. Thus our logspace procedure will begin by enumerating the elements of this sequence, and is guaranteed to find some such prime q. We will create an arithmetic circuit C' operating over $\mathbb{F}_q$ that will allow us to simulate C. (The logspace-uniform $\Lambda\mathsf{P}(\mathbb{F}_{p_n})$ family will actually have a circuit C'_q for each prime q in a polynomially-large range. The logspace oracle machine will thus first find the appropriate q, and then pick the appropriate length for its queries, so that it will obtain the values of the appropriate C'_q.)

For each gate g of C and each $a \in \mathbb{F}_{p_n}$, C' will have a gate computing the Boolean value $[g=a]$. If g is an input gate, our logspace procedure will compute the value of each $[g=a]$ and provide these as the inputs to the circuit C'.

Let us now consider the case when g is a + gate, $g = \sum_i h_i$. Let γ be a generator of the cyclic subgroup of the multiplicative group of $\mathbb{F}_q$ of order p_n. Our circuit C' will have gates $h_{i,a}$ computing the value

$$h_{i,a} = ([h_i = a] \times (\gamma^a - 1) + 1).$$

Observe that $\prod_a h_{i,a}$ is equal to γ^{h_i}. C' will have a gate g' computing the value $g' = \prod_{i,a} h_{i,a}$. Note that g' is equal to $\gamma^{\sum_i h_i} = \gamma^g$ (since γ has order p_n). The value of the gate $[g = b]$ (for a given $b \in \mathbb{F}_{p_n}$) is thus $c_b^{-1} \times \prod_{\ell \neq b}(\gamma^\ell - g')$, where the constant $c_b = \prod_{\ell \neq b}(\gamma^\ell - b)$ can be computed in logspace and is thus available as a constant in C'.

It remains only to deal with the case when g is a $\times$ gate, $g = \prod_i h_i$. In C', the gate $[g = 0]$ is $1 - \prod_i (1 - [h_i = 0])$.

Let μ be a generator of the multiplicative group of $\mathbb{F}_{p_n}$, and let α be a generator of the subgroup of the multiplicative group of $\mathbb{F}_q$ of order $p_n - 1$. If g does not evaluate to 0, then g is equal to μ^b for some b. Our circuit C' will have gates $h_{i,\ell}$ computing the values

$$h_{i,\ell} = ([h_i = \mu^\ell] \times (\alpha^\ell - 1) + 1).$$

Our circuit C' will have gates h'_i computing the value $h'_i = \prod_\ell h_{i,\ell}$. Observe that h'_i is equal to α^a if $h_i = \mu^a$, and h'_i is equal to 1 if $h_i = 0$.

In C', there will be a gate g' that computes the following value: $g' = (1 - [g = 0]) \prod_i h'_i = ([g \neq 0]) \prod_i \alpha^{\log_\mu h_i} = ([g \neq 0])\alpha^{\sum_i \log_\mu h_i} = ([g \neq 0])\alpha^{\log_\mu g}$. Observe that, if $g \neq 0$, then $g = \mu^b$ for some b, and in this case g' evaluates to α^b. The value of the gate $[g = \mu^b]$ (for a given $b \in \mathbb{F}_{p_n}$) is thus $c_b^{-1} \times \prod_{\ell \neq b}(\alpha^\ell - g')$, where the constant $c_b = \prod_{\ell \neq b}(\alpha^\ell - \mu^b)$ can be computed in logspace and is thus available as a constant in C'. □

For completeness, we add two more relevant characterizations of TC^1. (Recall the definition of g-$\mathsf{AC}^1[m]$ from Definition 16.)

Theorem 11 $\mathsf{TC}^1 = \#\mathsf{AC}^1(\mathbb{F}_{p_n}) = \mathsf{L}^{\Lambda\mathsf{P}(\mathbb{F}_{p_n})} = \mathsf{AC}^1[p_n] = \mathsf{CC}^1[p_n]$.

Proof: We need only consider the last two equalities.

($\supseteq$): MAJORITY gates can simulate AND, OR, and MOD_{p_n} gates in constant depth; thus this direction is easy.

($\subseteq$): Let ϵ be chosen so that $2n^\epsilon < p_n$ for every n. Any MAJORITY gate (of fan-in n^k) can be simulated by an AC^0-reduction to MAJORITY gates having fan-in n^ϵ [4]. Thus if $A \in \mathsf{TC}^1$, then A is accepted by a family of circuits of AND, OR, and MAJORITY gates, where the MAJORITY gates have fan-in at most n^ϵ. It suffices to show how to simulate a MAJORITY gate with inputs $h_1, \ldots, h_\ell$. Note that $\mathrm{MOD}_{p_n}(h_1, \ldots, h_\ell, 1^{p_n - b})$ computes the value $[b = \sum_i h_i]$. Thus the MAJORITY of the h_i is simply the OR, over all $b > \ell/2$ of the subcircuits computing $[b = \sum_i h_i]$.

For the final equality, first note any AND or OR gate with fan-in at least p_n can be replaced by a constant-depth tree of AND and OR gates of fan-in strictly less than p_n. Next, use DeMorgan's laws to remove all of the AND gates. Thus the circuit has only MOD gates and small fan-in OR gates. But note that if we feed the wires from an OR gate into a MOD_{p_n} gate, then the result is the NOR of the inputs (since if all of the wires are zero, the MOD gate outputs 1, and otherwise the number of wires that are one is less than p_n, and thus the MOD gate outputs zero. Negating each such NOR (again using a MOD gate) completes the proof. □

We also mention that Theorem 11 generalizes to other depths, in a way analogous to Corollary 6:

Corollary 7 $\mathsf{TC}^i = \#\mathsf{AC}^i(\mathbb{F}_{p_n}) = \mathsf{AC}^i[p_n] = \mathsf{CC}^i[p_n]$.

For $i \geq 1$ the equality $\mathsf{TC}^i = \mathsf{L}^{\#\mathsf{SAC}^{i,*}(\mathbb{F}_{p_n})}$ also holds, but for $i = 0$ a more careful argument is needed, using AC^0-Turing reducibility in place of logspace-Turing reducibility.

In order to set the context for the results of the next section, it is necessary to consider an extension of Theorem 10, involving arithmetic circuits over certain *rings*. Thus we require the following definition.

Definition 17 *Let (m_n) be any sequence of natural numbers (where each $m_n > 1$) such that the mapping $1^n \mapsto m_n$ is computable in logspace. We use the notation*

$\#\mathsf{AC}^1(\mathbb{Z}_{m_n})$ to denote the class of functions f with domain $\{0,1\}^*$ such that there is a logspace-uniform family of arithmetic circuits $\{C_n\}$ of logarithmic depth with unbounded fan-in $+$ and $\times$ gates, where the arithmetic operations of the circuit C_n are interpreted over $\mathbb{Z}_{m_n}$, and for any input x of length n, $f(x) = C_n(x)$. We use the notation $\#\mathsf{AC}^1(\mathbb{Z}_{\mathsf{L}})$ to denote the union, over all logspace-computable sequences of moduli (m_n), of $\#\mathsf{AC}^1(\mathbb{Z}_{m_n})$.

Since the sequence of primes (p_n) is logspace-computable, $\mathsf{TC}^1 (= \#\mathsf{AC}^1(\mathbb{F}_{p_n}))$ is clearly contained in $\#\mathsf{AC}^1(\mathbb{Z}_{\mathsf{L}})$. Conversely, all of the functions in $\#\mathsf{AC}^1(\mathbb{Z}_{\mathsf{L}})$ are computable in TC^1. To see this, consider a function $f \in \#\mathsf{AC}^1(\mathbb{Z}_{\mathsf{L}})$. To evaluate $f(x)$ for an input of length n, first we compute the modulus m_n and the circuit C_n. To evaluate each gate g of C_n (in binary), first we compute the sum or product of the values that feed into g (which can be done in constant depth using threshold circuits) and then we reduce the result modulo m_n (which involves division, which can also be computed in constant depth). Thus, arithmetic circuits over the integers mod m_n for reasonable sequences of moduli m_n give yet another arithmetic characterization of TC^1.

3.4.1 Degree Reduction

The results of Sections 3.3 and 3.4 gave examples of fan-in reduction for arithmetic circuits (showing that ACC^1 and TC^1 can be characterized either in terms of unbounded fan-in or semiunbounded fan-in arithmetic circuits). However, those theorems showed only how to reduce the fan-in of addition gates; thus they did not involve decreasing the algebraic degree of the circuits under consideration. Degree reduction is the topic to which we turn now.

In this section, we introduce a class of circuits that is intermediate between the unbounded fan-in circuit model and the semiunbounded fan-in model, for the purposes of investigating when arithmetic circuits of superpolynomial algebraic degree can be

simulated by arithmetic circuits (possibly over a different algebra) with much smaller algebraic degree.

The starting point for this section is Theorem 4.3 in [2], which states that every problem in AC^1 is reducible to a function computable by polynomial-size arithmetic circuits of degree $n^{O(\log\log n)}$. In this section, we refine the result of [2], and put it in context with the theorems about TC^1 that were presented in the previous section. Those results show that TC^1 reduces to semiunbounded fan-in arithmetic circuits in the $\Lambda\mathsf{P}(\mathbb{F}_{p_n})$ model, but leave open the question of whether TC^1 also reduces to semiunbounded fan-in arithmetic circuits in the $\mathsf{VP}(\mathbb{F}_{p_n})$ model (which coincides with $\mathsf{VP}(\mathbb{Q})$). We are unable to answer this question, but we do show that some interesting inclusions can be demonstrated if we relax the VP model, by imposing a less-stringent restriction on the fan-in of the $\times$ gates.

Definition 18 *Let (m_n) be any sequence of natural numbers (where each $m_n > 1$) such that the mapping $1^n \mapsto m_n$ is computable in logspace. $\#\mathsf{WSAC}^1(\mathbb{Z}_{m_n})$ is the class of functions represented by logspace-uniform arithmetic circuit families $\{C_n\}$, where C_n is interpreted over $\mathbb{Z}_{m_n}$, where each C_n has size polynomial in n, and depth $O(\log n)$, and where the $+$ gates have unbounded fan-in, and the $\times$ gates have fan-in $O(\log n)$. Thus these circuits are not semiunbounded, but have a "weak" form of the semiunbounded fan-in restriction. We use the notation $\#\mathsf{WSAC}^1(\mathbb{Z}_{\mathsf{L}})$ to denote the union, over all logspace-computable sequences of moduli (m_n), of $\#\mathsf{WSAC}^1(\mathbb{Z}_{m_n})$. In the special case when $m_n = p$ for all n, we obtain the class $\#\mathsf{WSAC}^1(\mathbb{F}_p)$.*

We refrain from defining a weakly semiunbounded analog of the $\Lambda\mathsf{P}$ classes, because it is easy to show that they are equivalent to the $\Lambda\mathsf{P}$ classes, since AC^0 circuits can add logarithmically-many numbers, given in binary.

We improve on [2, Theorem 4.3] by showing AC^1 is contained in $\#\mathsf{WSAC}^1(\mathbb{F}_2)$; note that all polynomials in $\#\mathsf{WSAC}^1(\mathbb{F}_p)$ have degree $n^{O(\log\log n)}$, and note also that the class of functions considered in [2] is not obviously even in TC^1. In addition, we

improve on [2] by reducing not merely AC^1, but also $\mathsf{AC}^1[p]$ for any prime p. This includes $\Lambda\mathsf{P}(\mathbb{F}_p)$ for any p such that $\mathsf{Supp}(p-1) \subseteq \{2\}$. Also, we obtain an exact characterization of $\mathsf{AC}^1[p]$, whereas [2] presented merely an inclusion.

Theorem 12 *Let p be any prime. Then $\mathsf{AC}^1[p] = \#\mathsf{WSAC}^1(\mathbb{F}_p)$.*

Proof: The inclusion $\#\mathsf{WSAC}^1(\mathbb{F}_p) \subseteq \mathsf{AC}^1[p]$ is straightforward. The proof of Corollary 5 shows how to simulate semiunbounded fan-in circuits over $\mathbb{F}_p$ by $\mathsf{AC}^1[p]$ circuits. We merely need to add to that construction, to show how to handle multiplication gates of logarithmic fan-in. Let g be a multiplication gate computing the product of the gates $h_1, \ldots, h_{c \log n}$. As in the proof of Corollary 5, the simulating $\mathsf{AC}^1[p]$ circuit will have gates of the form $[h_i = b]$ for all $b \in \mathbb{F}_p$. Thus the value of g depends on only $O(\log n)$ binary bits of the simulating circuit, and the value of $[g = a]$ can be computed by a logspace-uniform DNF expression. This yields the desired $\mathsf{AC}^1[p]$ circuit.

For the proof of the converse inclusion, the main technical ingredient involved is the following lemma from [2]. (In [2] the lemma is stated only for MOD_2, but the proof carries over to any MOD_m gate with only trivial changes. (See also the very similar result of [37, Proposition 3.4].)

Lemma 1 *[2] Let m be any natural number, $m > 1$. For each $\ell \in \mathbb{N}$, there is a family of constant-depth, polynomial-size, probabilistic circuits consisting of unbounded-fan-in* MOD_m *gates,* AND *gates of fan-in $O(\log n)$, and $O(\log n)$ probabilistic bits, computing the* OR *of n bits, with error probability $< 1/n^\ell$.*

Now we follow closely the proof of [2, Theorem 4.3].

Take an $\mathsf{AC}^1[p]$ circuit, replace all AND gates by OR and MOD_p gates (using DeMorgan's laws), and then replace each OR gate in the resulting circuit with the sub-circuit guaranteed by Lemma 1 (for ℓ chosen so that n^ℓ is much larger than the

size of the original circuit), with the *same* $O(\log n)$ probabilistic bits re-used in each replacement circuit. The result is a probabilistic, polynomial-size circuit that, with high probability, provides the same output as the original circuit.

Note that replacing AND gates by $\times$ and replacing each MOD_p gate g having wires from h_i with a subcircuit of the form $1 - (\sum_i h_i)^{p-1}$, one obtains an arithmetic circuit over the integers, whose value mod p is equal to the output of the original $\mathsf{AC}^1[p]$ circuit with high probability. (This is one place where we use the fact that p is prime.) The circuit has depth $O(\log n)$, and has unbounded fan-in $+$ gates, and all $\times$ gates have fan-in $O(\log n)$, and thus it is a weakly semiunbounded fan-in circuit.

Create $n^{O(1)}$ copies of this probabilistic circuit, one copy for each sequence of probabilistic bits; call these circuits $D_1, D_2, \ldots, D_{n^c}$. Note that each D_i computes a value in $\{0, 1\}$. Note also that $1 - D_i$ is also computable in $\#\mathsf{WSAC}^1(\mathbb{F}_p)$. Thus we can feed these values into an arithmetic NC^1 circuit computing MAJORITY (using the fact that all functions in NC^1 are in $\#\mathsf{NC}^1$ [26]). The resulting circuit is equivalent to our original $\mathsf{AC}^1[p]$ circuit. □

We especially call attention to the following corollary, which shows that, over $\mathbb{F}_2$, polynomial size logarithmic depth arithmetic circuits of degree $n^{O(\log n)}$ and of degree $n^{O(\log \log n)}$ represent precisely the same functions!

Corollary 8 $\#\mathsf{WSAC}^1(\mathbb{F}_2) = \#\mathsf{AC}^1(\mathbb{F}_2) = \mathsf{AC}^1[2] = \Lambda\mathsf{P}(\mathbb{F}_3)$.

Proof: The containment $\#\mathsf{WSAC}^1(\mathbb{F}_2) \subseteq \#\mathsf{AC}^1(\mathbb{F}_2)$ is immediate from the definition (since $\#\mathsf{WSAC}^1(\mathbb{F}_2)$ circuits are a restricted form of $\#\mathsf{AC}^1(\mathbb{F}_2)$ circuits). The second equality is from Theorem 8. The equality $\mathsf{AC}^1[2] = \Lambda\mathsf{P}(\mathbb{F}_3)$ is from Theorem 7. The inclusion $\mathsf{AC}^1[2] \subseteq \#\mathsf{WSAC}^1(\mathbb{F}_2)$ is from Theorem 12. □

If we focus on the Boolean classes, rather than on the arithmetic classes, then we obtain a remarkable collapse.

Theorem 13 *Let $m \in \mathbb{N}$. Then $\mathsf{AC}^1[m] = \log\text{-}\mathsf{AC}^1[m]$.*

Proof: The proof of Theorem 12 begins with the statement of Lemma 1, which holds for any modulus m. The proof then uses Lemma 1 to replace a general $\mathsf{AC}^1[m]$ circuit by an equivalent probabilistic circuit with unbounded fan-in MOD_m gates and AND gates with logarithmic fan-in, using only $O(\log n)$ probabilistic bits.

The proof of Theorem 12 proceeds to modify this to obtain an arithmetic circuit. Instead, we simply make polynomially-many copies of this Boolean circuit (one copy for each probabilistic sequence), and take the majority vote of these copies. □

Using Theorem 8 it follows that arithmetic AC^1 circuits over any finite field $\mathbb{F}_p$ can be simulated by Boolean circuits with MOD gates and small fan-in AND gates. It remains open whether this in turn leads to small-degree arithmetic circuits over $\mathbb{F}_p$ when $p > 2$, and also whether the fan-in of the AND gates can be sublogarithmic, without loss of power.

When m is composite, Theorem 13 can be improved to obtain an even more striking collapse, by invoking the work of Hansen and Koucký [37].

Theorem 14 *Let m not be a prime power. Then $\mathsf{AC}^1[m] = \mathsf{CC}^1[m]$.*

Proof:

Let $p \neq q$ where $\{p, q\} \subseteq \mathsf{Supp}(m)$. It suffices to show how to construct a family of $\mathsf{CC}^1[m]$ circuits to simulate a given $\mathsf{AC}^1[m]$ circuit family.

Hansen and Koucký showed [37, Lemma 3.5] that, for every $c > 1$ there is a constant-depth probabilistic circuit composed of MOD_{pq} gates that computes the OR of n variables, using only $O(\log n)$ probabilistic bits, and having error probability less than $1/n^c$. Thus we can replace each unbounded fan-in AND and OR gate in the $\mathsf{AC}^1[m]$ circuit with the corresponding circuit (possibly with negation gates) guaranteed by [37]. The MOD_{pq} gates can be replaced with MOD_m gates via standard

techniques, as in the proof of Theorem 7. By choosing a suitably large value for c, the resulting probabilistic circuit simulates the original circuit with small error probability.

Now, as in the proof of Theorem 13 we can make polynomially-many copies of the probabilistic circuit, hardwiring in different values for the probabilistic bits, and take the majority vote. □

Corollary 9 $\mathsf{ACC}^1 = \bigcup_p \Lambda\mathsf{P}(\mathbb{F}_p) = \bigcup_p \#\mathsf{AC}^1(\mathbb{F}_p) = \bigcup_m \#\mathsf{AC}^1(\mathbb{Z}_m) = \mathsf{CC}^1$.

Corollary 10 $\mathsf{ACC}^i = \mathsf{CC}^i$ *for all* $i \geq 1$.

This equality is still open for the case $i = 0$, although Hansen and Koucký show that the probabilistic versions of ACC^0 and CC^0 coincide [37].

Note that

$$\bigcup_{p \text{ prime}} \mathsf{CC}^1[p] = \bigcup_{p \geq 2} \mathsf{VP}(\mathbb{F}_p) \subseteq \bigcup_m \mathsf{L}^{\mathsf{VP}(\mathbb{Z}_m)} \subseteq \mathsf{ACC}^1 = \mathsf{CC}^1.$$

The right-most class corresponds to uniform families of MOD_m gates (for *composite* m), and to arithmetic circuits of degree $n^{O(\log n)}$. The left-most class consists of uniform families of MOD_p gates for *prime* p, and to arithmetic circuits of degree $n^{O(1)}$. The intermediate class corresponds to arithmetic circuits of polynomial degree, but having access to composite moduli. It is natural to wonder how much the composite moduli can help, in simulating higher-degree arithmetic circuits using small degree.

It might be useful to have additional examples of algebras, where some degree reduction can be accomplished. Thus we also offer the following theorem:

Theorem 15 *Let* p *be any prime. Then* $\mathsf{AC}^1[p] \subseteq \mathsf{L}^{\#\mathsf{WSAC}^1(\mathbb{Z}_\mathsf{L})}$.

Proof: We start with the sequence of circuits $D_1, D_2, \ldots, D_{n^c}$ created in the proof of Theorem 12. We now make use of the "Toda polynomials" introduced in [53]. For

example, there is an explicit construction in [18] of a polynomial P_k of degree $2k-1$ such that $(y \bmod p) \in \{0,1\}$ implies $P_k(y) \bmod p^k = y \bmod p$. It is observed in [3] that, for $k = O(\log n)$, the polynomial P_k can be implemented via logspace-uniform constant-depth circuits over the integers. Thus, by replacing each multiplication gate with a tree of fan-in two, the polynomial can be implemented by a semiunbounded fan-in circuit of logarithmic depth. Applying this polynomial to the output of each circuit D_i, we obtain a $\#\mathsf{WSAC}^1(\mathbb{Z})$ circuit whose value mod p is the same as the output of the original $\mathsf{AC}^1[p]$ circuit with high probability, and with the additional property that the output of the circuit, when represented in p-ary notation, has all of the $c \log n$ low-order symbols of the result equal to zero (except possibly the lowest-order symbol). We will choose c to be the constant such that there are $c \log n$ probabilistic bits). Call the resulting circuit E_i.

Now create a circuit whose output gate computes $\sum_i E_i$. The output gate of the resulting $\#\mathsf{WSAC}^1(\mathbb{Z})$ circuit records a number whose low-order $c \log n$ positions (in p-ary notation) records the number of the n^c copies that output 1. If this number is greater than $n^c/2$, then the original circuit accepted its input; otherwise it rejected its input.

In order to compute this number using $\#\mathsf{WSAC}^1(\mathbb{Z}_{\mathsf{L}})$ instead of $\#\mathsf{WSAC}^1(\mathbb{Z})$, we use this logspace-computable sequence of moduli: $m_n = p^n$. Evaluating the arithmetic over $\mathbb{Z}_{p^n}$ gives the number represented by the low-order n positions of the result, in p-ary notation. A logspace oracle machine, upon being given this number (say, in binary notation) can compute the value of this number modulo $p^{1+c \log n}$ and determine if that number is greater than $n^c/2$, and can thereby determine if the original circuit accepted its input. □

It is natural to wonder whether this theorem can be extended, to allow composite moduli. A direct application of the techniques of [3, 18, 61] requires multiple applications of the Toda polynomials, and this in turn results in circuits of superlogarithmic

depth.

Using Theorems 7 and 8 we obtain the following.

Corollary 11 *If p is a Fermat prime, then* $\Lambda\mathsf{P}(\mathbb{F}_p) \subseteq \mathsf{L}^{\#\mathsf{WSAC}^1(\mathbb{Z}_\mathsf{L})}$.

3.5 Conclusions, Discussion, and Open Problems

We have introduced the complexity classes $\Lambda\mathsf{P}(R)$ for various algebraic structures R, and have shown that they provide alternative characterizations of well-known complexity classes. Furthermore, we have shown that arithmetic circuit complexity classes corresponding to polynomials of degree $n^{O(\log\log n)}$ also yield new characterizations of complexity classes, such as the equality

$$\mathsf{AC}^1[p] = \log\text{-}\mathsf{AC}^1[p] = \#\mathsf{WSAC}^1(\mathbb{F}_p).$$

Furthermore, in the case when $p = 2$, we obtain the additional collapse

$$\#\mathsf{AC}^1(\mathbb{F}_2) = \mathsf{AC}^1[2] = \log\text{-}\mathsf{AC}^1[2] = \#\mathsf{WSAC}^1(\mathbb{F}_2),$$

showing that algebraic degree $n^{O(\log n)}$ and $n^{O(\log\log n)}$ have equivalent expressive power, in this setting.

We have obtained new characterizations of ACC^1 in terms of restricted fan-in:

$$\mathsf{ACC}^1 = \bigcup_p \#\mathsf{AC}^1(\mathbb{F}_p) = \bigcup_p \Lambda\mathsf{P}(\mathbb{F}_p) = \mathsf{CC}^1.$$

That is, although ACC^1 corresponds to unbounded fan-in arithmetic circuits of logarithmic depth, and to unbounded fan-in Boolean circuits with modular counting gates, no power is lost if the addition gates have bounded fan-in (in the arithmetic case) or if only the modular counting gates have unbounded fan-in (in the Boolean

case). It remains unknown if every problem in ACC^1 is reducible to a problem in $\bigcup_m \mathsf{VP}(\mathbb{Z}_m)$, although we believe that our theorems suggest that this is likely. It would be highly interesting to see such a connection between ACC^1 and VP.

We believe that it is fairly likely that several of our theorems can be improved. For instance:

- Perhaps Theorems 13 and 14 can be improved, to show that for all m, $\mathsf{AC}^1[m] = \mathsf{CC}^1[m]$. Note that this is already known to hold if m is not a prime power. By Corollary 5 this would show that $\mathsf{VP}(\mathbb{F}_p) = \mathsf{AC}^1[p]$ for all primes p. It would also show that $\#\mathsf{AC}^1(\mathbb{F}_2) = \mathsf{VP}(\mathbb{F}_2) = \Lambda\mathsf{P}(\mathbb{F}_p)$ for every Fermat prime p. (We should point out that this would imply that $\mathsf{AC}^1 \subseteq \mathsf{VP}(\mathbb{F}_p)$ for every prime p, whereas even the weaker inclusion $\mathsf{SAC}^1 \subseteq \mathsf{VP}(\mathbb{F}_p)$ is only known to hold non-uniformly [34].)

- Can Corollary 11 be improved to hold for all primes p, or even for $\Lambda\mathsf{P}(\mathbb{F}_{p_n})$? The latter improvement would show that $\mathsf{TC}^1 \subseteq \mathsf{L}^{\#\mathsf{WSAC}^1(\mathbb{Z}_\mathsf{L})}$.

- Perhaps one can improve Theorem 15, to achieve a simulation of degree $n^{O(1)}$. Why should $n^{O(\log\log n)}$ be optimal? Perhaps this could also be improved to hold for composite moduli?

- If some combinations of the preceding improvements are possible, TC^1 would reduce to $\mathsf{VP}(\mathbb{Q})$, which would be a significant step toward the Immerman-Landau conjecture.

We began this investigation, wondering if the equality $\mathsf{VP}(B_2) = \Lambda\mathsf{P}(B_2)$ could carry over to any other algebraic structure. We think that it appears as if $\mathsf{VP}(\mathbb{F}_p)$ and $\Lambda\mathsf{P}(\mathbb{F}_p)$ are incomparable for every non-Fermat prime $p > 2$, since $\mathsf{VP}(\mathbb{F}_p) = \mathsf{CC}^1[p]$ and $\Lambda\mathsf{P}(\mathbb{F}_p) = \mathsf{CC}^1[\mathsf{Supp}(p-1)]$. That is, these classes correspond to circuits consisting of modular counting gates for completely different sets of primes. For Fermat primes we have $\Lambda\mathsf{P}(\mathbb{F}_p) = \text{log-}\mathsf{AC}^1[2]$ and again the VP and $\Lambda\mathsf{P}$ classes seem incomparable.

For the special case of $p = 2$, we have $\mathsf{VP}(\mathbb{F}_2) = \mathsf{CC}^1[2]$ and $\Lambda\mathsf{P}(\mathbb{F}_2) = \mathsf{AC}^1$. We hold out some hope that $\mathsf{VP}(\mathbb{F}_2) = \mathsf{AC}^1[2]$, in which case it would appear that the VP class could be *more* powerful than the $\Lambda\mathsf{P}$ class – but based on current knowledge it also appears possible that the VP and $\Lambda\mathsf{P}$ classes are incomparable even for $p = 2$.

Some of our theorems overcome various hurdles that would appear to stand in the way of a proof of our conjecture that $\mathsf{ACC}^1 = \bigcup_m \mathsf{L}^{\mathsf{VP}(\mathbb{Z}_m)}$.[4] First, recall that $\mathsf{VP}(\mathbb{Z}_m) \subseteq \mathsf{CC}^1[m]$ (Corollary 5). Thus, if the conjecture is correct, then *unbounded* fan-in AND and OR gates would have to be simulated efficiently with *bounded* fan-in AND and OR gates (which in turn can be replaced by MOD gates). But this is true in this context: $\mathsf{AC}^1[m] = \mathsf{CC}^1[m]$, if m is not a prime power (Theorem 14). If m is a prime power, then the fan-in can be reduced to $\log n$ (Theorem 13). If the fan-in can be reduced to $O(1)$ also in the case of prime power moduli, then $\mathsf{AC}^1[p] = \mathsf{CC}^1[p] = \mathsf{VP}(\mathbb{F}_p)$. If CC^1 circuits can be simulated using an oracle for functions in $\mathsf{VP}(\mathbb{Z}_{m'})$ for some m', then the conjecture holds. (The latter simulation is possible if the MOD gates in the CC^1 circuits are for a prime modulus; see Corollary 5.)

A second objection that might be raised against the conjecture deals with algebraic degree. ACC^1 corresponds precisely to polynomial-size logarithmic depth unbounded fan-in arithmetic circuits over finite fields (Corollary 3). Such circuits represent polynomials of degree $n^{O(\log n)}$, whereas VP circuits represent polynomials of degree only $n^{O(1)}$. One might assume that there are languages represented by polynomial-size log-depth arithmetic circuits of degree $n^{O(\log n)}$ that actually *require* such large degree in order to be represented by arithmetic circuits of small size and depth.

Our degree-reduction theorem (Corollary 8) shows that this assumption is incorrect. Every Boolean function that can be represented by an arithmetic AC^1 circuit over $\mathbb{F}_2$ (with algebraic degree $n^{O(\log n)}$) can be represented by an arithmetic AC^1 circuit over $\mathbb{F}_2$ where the multiplication gates have fan-in $O(\log n)$ (and thus the

[4] Here, "$\mathsf{VP}(\mathbb{Z}_m)$" refers to the class of *functions* defined on $\mathbb{Z}_m$ that are represented by VP circuits, rather than to a class of *languages*. The distinction is significant, as is discussed in [7].

arithmetic circuit has algebraic degree $n^{O(\log \log n)}$).

Chapter 4

Cost-register automata

4.1 Introduction

We study various classes of *regular functions*, as defined in a recent series of papers by Alur *et al.* [13, 15, 14]. 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 [31] 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 [10] provided a characterization in terms of a new class of *one-way* deterministic finite automata, known as *streaming string transducers*; see also [11]. 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 4.2.)

The model that has been studied in [13, 15, 14], 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 $(\mathbb{Z} \cup \{\infty\}, \min, +)$ can be used to give an alternative characterization of the class of functions computed by weighted automata [13].

The cost register automaton model is the main machine model that was advocated by Alur *et al.* [13] 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. There was even a suggestion [9] 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.

4.2 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.

For more detailed definitions about the following standard circuit complexity classes (as well as for motivation concerning the standard choice of the U_E-uniformity), we refer the reader to [58, Section 4.5].

- $\mathsf{NC}^i = \{A : A$ 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.) We use these same names to refer to the associated classes of *functions* computed by the corresponding classes of circuits.

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 [58, 6] 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 $\mathsf{GapNC}^1 \subseteq \mathsf{L}$ [38].

4.2.1 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.* [13].

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 $w = a_1 \ldots a_n$, the *run* of M on w is the sequence of configurations $(q_0, \nu_0), \ldots (q_n, \nu_n)$ such that, for each $i \in \{1, \ldots, n\}$ $\delta(q_{i-1}, a_i) = q_i$ 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 [13] 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 [14], 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.

4.3 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 $(\Gamma^*, \circ)$.

4.3.1 CRAs over the integers

Additive CRAs (ACRAs) are CRAs that operate over commutative monoids. They have been studied in [13, 15, 14]; in [15] 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 16 *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 [16].)*

Proof: It was shown in [13] 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.
- $(x, i) \mapsto G_i$, where G_i is a labeled 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. 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 $f(x)$. $\square$ 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 17 *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 upper bound now follows since iterated multiplication of $O(1) \times O(1)$ integer matrices can be computed in GapNC^1 [26].

For the lower bound, observe that it is shown in [26], building on [19], 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.) □

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

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

CCRAs operating over the algebraic structure $(\Gamma^*, \circ)$ are precisely the so-called *streaming string transducers* that were studied in [11], 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 18 *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 [16].)*

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 [17], it follows that the "comes before" predicate is computable in AC^0.

Thus, 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 [16]. 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. □

4.4 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 4.4.1).
- CRAs operating over the commutative semiring $(\mathbb{Z} \cup \{\infty\}, \min, +)$: the so-called "tropical" semiring (Section 4.5).
- CRAs operating over the noncommutative semiring $(\Gamma^* \cup \{\perp\}, \max, \circ)$ (Section 4.6).

There is a large literature dealing with *weighted automata* operating over semirings. It is shown in [13] 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 $r \leftarrow r' \times c$ for some register r' and some semiring element c. 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 17, 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 [43, 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 $k \geq 3$ [26, 19].)

4.4.1 CRAs over the integers.

First, we consider the copyless case:

Theorem 19 *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 2 below that $f \in \mathsf{GapNC}^1$. □

Lemma 2 *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: By [25], there is a logarithmic-depth arithmetic-Boolean formula over the integers, that takes as input an encoding of a formula 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* and *select* gates that

provide an interface between the two types of gates. Actually, the construction given in [25] does not utilize any *test* gates [24], 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 [25], *select* gates are only used when y is a Boolean value. When operating over the integers, then, $select(y, x_0, x_1)$ 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., [6]), 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 17).

4.5 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 $\#\mathsf{NC}^1_{\text{trop}}$, and we prove some basic propositions about arithmetic circuits over the tropical semiring.

4.5.1 Arithmetic Circuit Preliminaries

Functions in $\#\mathsf{NC}^1_{\text{trop}}$ have complexity in some sense intermediate between NC^1 and $\#\mathsf{NC}^1$. Proposition 4 shows that there are some functions in $\#\mathsf{NC}^1_{\text{trop}}$ that are hard for NC^1, and Lemma 3 shows that, if the values at the input level of $\#\mathsf{NC}^1_{\text{trop}}$ 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 [6]. 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.

Proposition 4 $\mathsf{NC}^1 \subseteq \#\mathsf{NC}^1_{\text{trop}}$.

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., [6].) 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 5 *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 [25], 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 [30, 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 [30]. 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 makes precise what is meant by "no harder than" in this context.

Definition 19 *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".

Lemma 3 *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 [40] (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 en-

codes 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 1 *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 circuit the value of ϕ can easily be extracted from the value of ϕ_0.

Claim 2 *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 1. 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 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 $\frac{c}{2}\log n$. □

4.5.2 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 20 *All functions computable by CCRAs over the tropical semiring are computable in* $\mathsf{NC}^1(\#\mathsf{NC}^1_{\text{trop}})$*, and are computable in* L*.*

Here, $\mathsf{NC}^1(\#\mathsf{NC}^1_{\text{trop}})$ refers to the class of functions expressible as $g(f(x))$ for some functions $f \in \mathsf{NC}^1$ and $g \in \#\mathsf{NC}^1_{\text{trop}}$.

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 3, because, when we establish the $\mathsf{NC}^1(\#\mathsf{NC}^1_{\text{trop}})$ 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 $\mathsf{NC}^1(\#\mathsf{NC}^1_{\text{trop}})$ 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 19.

Next, as in the proof of Lemma 2, recall that, by [25], 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 $f \in \mathsf{NC}^1(\#\mathsf{NC}^1_{\text{trop}})$.

□

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 4.4, 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.

4.6 CRAs over the max-concat semiring.

As in Section 4.3.2, we consider only CCRAs.

Theorem 21 *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 OptLogCFL. OptLogCFL was defined by Vinay [57] as follows: f is in OptLogCFL if there is a nondeterministic logspace-bounded auxiliary pushdown automaton M running in polynomial time, such that, on input x, $f(x)$ is the lexicographically largest string that appears on the output tape of M along any 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 computation path. That is, we have $f \in$ OptLogCFL.

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

Chapter 5

Conclusion: future directions

We have surveyed a number of different settings, from Turing machines to circuits to automata, and shown how the notion of reductions can create interplay between these seemingly incomparable models. We have attempted to design an algorithm that uses undirected reachability to solve the Shuffle problem in L, showing some of the structure of such graphs along the way. We then gave a number of new characterizations to the circuit families in between AC^1 and TC^1 using our new classes $\Lambda\mathsf{P}$ and $\#\mathsf{WSAC}^1(\mathbb{Z}_{m_n})$, showing fan-in reductions for many classes in ACC^1 as well as degree reductions of polynomials of degree $n^{O(\log n)}$. Finally we showed the complexity of evaluating cost-register automata over a couple of different semirings, as well as what power is lost with the copyless restriction. The map of complexity classes and problems surveyed in this work can be found in the appendix.

The work in chapter 2 leaves much to be desired, although there seems to be evidence that the structure of gridgraphs corresponding to Shuffle instances is regular enough that a logspace algorithm could solve them. While our theorems do not carry over to more general gridgraphs, we hold out hope that a bit more work on analyzing Shuffle could give us the results that were desired.

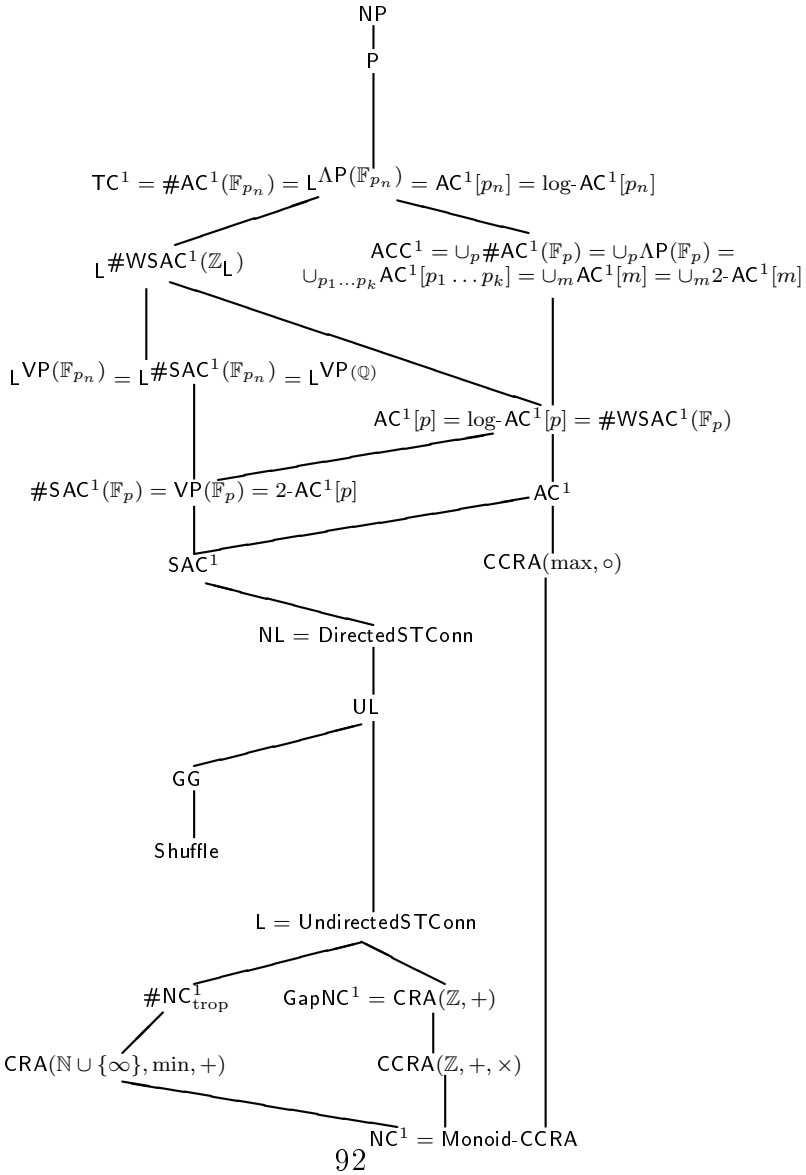
Our work on circuits in chapter 3 is also nowhere close to being done. Our new

characterizations of TC^1 and $\mathsf{VP}(\mathbb{Q})$ bring us hope that the Immerman-Landau conjecture is indeed true, refuting in part many of the degree arguments previously made to the contrary. Additionally there is no reason to think that our degree reductions or characterizations are optimal, and while it seems unlikely to us that $\mathsf{VP}(\mathbb{F}_p)$ could ever fully capture ACC^1, there may be a closer connection between these two classes than is currently known.

A natural extension of our results in chapter 4 could determine the complexity of solving cost-register automata over more generalized semirings, which would give us a greater understanding of how this arithmetically-motivated automata model fits into classes such as L and AC^1. Also, is it possible that such automata models are even complete for their respective classes? We can use the cost-register augmentation to evaluate arithmetic functions over the course of the automaton's execution, while most of the upper bounds we have given are circuit complexity classes, so there is no immediate reason to suspect completeness.

Appendix A

Charts of relevant complexity classes



Bibliography

[1] M. Agrawal, E. Allender, and S. Datta. On TC^0, AC^0, and arithmetic circuits. *Journal of Computer and System Sciences*, 60:395–421, 2000.

[2] Allender, J. Jiao, M. Mahajan, and V. Vinay. Non-commutative arithmetic circuits: Depth reduction and size lower bounds. *Theoret. Comp. Sci.*, 209:47–86, 1998.

[3] E. Allender and V. Gore. A uniform circuit lower bound for the permanent. *SIAM J. Comput.*, 23:1026–49, 1994.

[4] E. Allender and M. Koucký. Amplifying lower bounds by means of self-reducibility. *Journal of the ACM*, 57:14:1 – 14:36, 2010.

[5] E. Allender, K. Reinhardt, and S. Zhou. Isolation, matching, and counting: Uniform and nonuniform upper bounds. *Journal of Computer and System Sciences*, 59(2):164–181, 1999.

[6] Eric Allender. Arithmetic circuits and counting complexity classes. In J. Krajíček, editor, *Complexity of Computations and Proofs*, volume 13 of *Quaderni di Matematica*, pages 33–72. Seconda Università di Napoli, 2004.

[7] Eric Allender and Asa Goodwillie. Arithmetic circuit classes over $\mathbb{Z}_m$. Technical Report 15-145, Electronic Colloquium on Computational Complexity (ECCC), 2015.

[8] Eric Allender and Ian Mertz. Complexity of regular functions. In *Proc. 9th International Conference on Language and Automata Theory and Applications (LATA)*, number 8977 in Lecture Notes in Computer Science, pages 449–460. Springer, 2015.

[9] R. Alur. Regular functions. Lecture presented at *Horizons in TCS: A Celebration of Mihalis Yannakakis's 60th Birthday*, Center for Computational Intractability, Princeton, NJ, 2013.

[10] Rajeev Alur and Pavol Cerný. Expressiveness of streaming string transducers. In *Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[11] Rajeev Alur and Pavol Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 599–610, 2011.

[12] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions, cost register automata, and generalized min-cost problems. *CoRR*, abs/1111.0670, 2011.

[13] Rajeev Alur, Loris D'Antoni, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 13–22, 2013. See also the expanded version, [12].

[14] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. 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)*, page 9. ACM, 2014.

[15] Rajeev Alur and Mukund Raghothaman. Decision problems for additive regular functions. In *International Conference on Automata, Languages, and Programming (ICALP)*, number 7966 in Lecture Notes in Computer Science, pages 37–48. Springer, 2013.

[16] D. A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[17] D. A. M. Barrington, C.-J. Lu, P. B. Miltersen, and S. Skyum. Searching constant width mazes captures the AC^0 hierarchy. In *15th International Symposium on Theoretical Aspects of Computer Science (STACS)*, number 1373 in Lecture Notes in Computer Science, pages 73–83. Springer, 1998.

[18] R. Beigel and J. Tarui. On ACC. *Computational Complexity*, 4:350–366, 1994.

[19] Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM Journal on Computing*, 21(1):54–58, 1992.

[20] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal on Computing*, 18:559–578, 1989. See Erratum in SIAM J. Comput. 18, 1283.

[21] H. Buhrman, R. Cleve, M. Koucký, B. Loff, and F. Speelman. Computing with a full memory: catalytic space. In *STOC*, pages 857–866, 2014.

[22] P. Bürgisser. Cook's versus Valiant's hypothesis. *Theoret. Comp. Sci.*, 235:71–88, 2000.

[23] Peter Bürgisser. On the structure of Valiant's complexity classes. *Discrete Mathematics & Theoretical Computer Science*, 3(3):73–94, 1999.

[24] Samuel Buss. Comment on formula evaluation. Personal communication., 2014.

[25] Samuel R. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, 1992.

[26] H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer. Nondeterministic NC^1 computation. *Journal of Computer and System Sciences*, 57(2):200–212, 1998.

[27] A. Chiu, G.I. Davida, and B. Litow. Division in logspace-uniform NC^1. *RAIRO Theoretical Informatics and Applications*, 35:259–276, 2001.

[28] N. V. Vinodchandran Chris Bourke, Raghunath Tewari. Directed planar reachability is in unambiguous log-space. *ACM Transactions on Computation Theory*, 1(1):4:1–4:17, 2003.

[29] C. Corrales-Rodrigáñez and R. Schoof. The support problem and its elliptic analogue. *Journal of Number Theory*, 64(2):276–290, 1997.

[30] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In *STACS'12 (29th Symposium on Theoretical Aspects of Computer Science)*, volume 14, pages 66–77. LIPIcs, 2012.

[31] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Trans. Comput. Log.*, 2(2):216–254, 2001.

[32] Anna Gál Eric Allender and Ian Mertz. Dual vp classes. In *Proc. 40th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, number 9235 in Lecture Notes in Computer Science, pages 14–25. Springer Berlin Heidelberg, 2015.

[33] Tanmoy Chakraborty Samir Datta Sambuddha Roy Eric Allender, David A. Mix Barrington. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45(4).

[34] A. Gál and A. Wigderson. Boolean complexity classes vs. their arithmetic analogs. *Random Struct. Algorithms*, 9(1-2):99–111, 1996.

[35] J. von zur Gathen. Parallel linear algebra. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 574–615. Morgan Kaufmann, 1993.

[36] Michael Hahn, Andreas Krebs, Klaus-Jörn Lange, and Michael Ludwig. Visibly counter languages and the structure of NC^1. In *Symposium on Mathematical Foundations of Computer Science (MFCS)*, number 9235 in Lecture Notes in Computer Science, pages 384–394. Springer, 2015.

[37] K. Arnsfelt Hansen and M. Koucký. A new characterization of ACC^0 and probabilistic CC^0. *Computational Complexity*, 19(2):211–234, 2010.

[38] W. Hesse, E. Allender, and D. A. M. Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *J. Comp. and System Sci.*, 65:695–716, 2002.

[39] Andreas Jakoby and Till Tantau. Computing shortest paths in series-parallel graphs in logarithmic space. In *Complexity of Boolean Functions, 12.03. - 17.03.2006*, volume 06111 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[40] Andreas Jakoby and Till Tantau. 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)*, number 4855 in Lecture Notes in Computer Science, pages 216–227. Springer, 2007. The proof of Lemma 4 can be found as the proof of Lemma 3.5 in [39].

[41] JD Ullman JE Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[42] Neil D. Jones. Space-bounded reducibility among combinatorial problems. In *J Comput. Syst. Sci.*, volume 11, pages 68–85, 1975.

[43] Stefan Kiefer, Andrzej S. Murawski, Joël Ouaknine, Björn Wachter, and James Worrell. On the complexity of equivalence and minimisation for Q-weighted automata. *Logical Methods in Computer Science*, 9(1), 2013.

[44] P. Koiran and S. Perifel. Interpolation in Valiant's theory. *Comput. Complexity*, 20:1–20, 2011.

[45] J. Reif and S. Tate. On threshold circuits and polynomial computation. *SIAM Journal on Computing*, 21:896–908, 1992.

[46] Omer Reingold. Undirected st-connectivity in log-space. In *ACM Symposium on Theory of Computing (STOC)*, volume 37, pages 376–385, 2005.

[47] K. Reinhardt and E. Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29:1118–1131, 2000.

[48] Michael Sipser. *Introduction to the Theory of Computation, Second Edition*. Thomson Course Technology, 2006.

[49] R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *STOC*, pages 77–82, 1987.

[50] Michael Soltys. Circuit complexity of shuffle. *Combinatorial Algorithms*, pages 402–411, 2013.

[51] H. Straubing. *Finite Automata, Formal Logic, and Circuit Complexity.* Birkhäuser, Boston, 1994.

[52] Denis Thérien. Circuits constructed with MOD_q gates cannot compute "AND" in sublinear size. *Computational Complexity*, 4(4):383–388, 1994.

[53] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comp.*, 20:865–877, 1991.

[54] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *London Math. Soc.*, volume 42, pages 230–265, 1937.

[55] L.G. Valiant. Completeness classes in algebra. In *Proc. 11th ACM STOC*, pages 249–261, 1979.

[56] L.G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Comput.*, 12(4):641–644, 1983.

[57] V Vinay. Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proceedings of 6th Structure in Complexity Theory Conference*, pages 270–284, 1991.

[58] H. Vollmer. *Intro. to Circuit Complexity: A Uniform Approach.* Springer, 1999.

[59] William P Wardlaw. Matrix representation of finite fields. *Mathematics Magazine*, pages 289–293, 1994.

[60] T. Xylouris. On the least prime in an arithmetic progression and estimates for the zeros of Dirichlet L-functions. *Acta Arithmetica*, 150:65–91, 2011.

[61] A. C.-C. Yao. On ACC and threshold circuits. In *FOCS*, pages 619–627, 1990.